RESEARCH-ARTICLE

FaaSGraph: Enabling Scalable, Efficient, and Cost-Effective Graph Processing with Serverless Computing

YUSHI LIU, Shanghai Jiao Tong University, Shanghai, China

SHIXUAN SUN, Shanghai Jiao Tong University, Shanghai, China

ZIJUN LI, Shanghai Jiao Tong University, Shanghai, China

QUAN CHEN, Shanghai Jiao Tong University, Shanghai, China

SEN GAO, National University of Singapore, Singapore City, Singapore

BINGSHENG HE, National University of Singapore, Singapore City, Singapore

View all

Published: 27 April 2024

Citation in BibTeX format

ASPLOS '24: 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2
April 27 - May 1, 2024
CA, La Jolla, USA

Conference Sponsors:
SIGARCH
SIGOPS
SIGPLAN

FaaSGraph: Enabling Scalable, Efficient, and Cost-Effective Graph Processing with Serverless Computing

Yushi Liu
Shanghai Jiao Tong University
Shanghai, China
ziliuziliulys@sjtu.edu.cn

Shixuan Sun
Shanghai Jiao Tong University
Shanghai, China
sunshixuan@sjtu.edu.cn

Zijun Li
Shanghai Jiao Tong University
Shanghai, China
lzjzx1122@sjtu.edu.cn

Quan Chen
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Sen Gao
National University of Singapore
Singapore
sen@u.nus.edu

Bingsheng He
National University of Singapore
Singapore
hebs@comp.nus.edu.sg

Chao Li
Shanghai Jiao Tong University
Shanghai, China
lichao@cs.sjtu.edu.cn

Minyi Guo
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

Abstract

Graph processing is widely used in cloud services; however, current frameworks face challenges in efficiency and cost-effectiveness when deployed under the Infrastructure-as-a-Service model due to its limited elasticity. In this paper, we present FaaSGraph, a serverless-native graph computing scheme that enables efficient and economical graph processing through the co-design of graph processing frameworks and serverless computing systems. Specifically, we design a data-centric serverless execution model to efficiently power heavy computing tasks. Furthermore, we carefully design a graph processing paradigm to seamlessly cooperate with the data-centric model. Our experiments show that FaaSGraph improves end-to-end performance by up to 8.3X and reduces memory usage by up to 52.4% compared to state-of-the-art IaaS-based methods. Moreover, FaaSGraph delivers steady 99%-ile performance in highly fluctuated workloads and reduces monetary cost by 85.7%.

ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640361

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Mathematics of computing** → **Graph algorithms**.

Keywords: Serverless Computing, Graph Processing, Resource Sharing, Cold Start

ACM Reference Format:
Yushi Liu, Shixuan Sun, Zijun Li, Quan Chen, Sen Gao, Bingsheng He, Chao Li, and Minyi Guo. 2024. FaaSGraph: Enabling Scalable, Efficient, and Cost-Effective Graph Processing with Serverless Computing. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620665.3640361

1 Introduction

Graphs effectively model complex relationships among entities, making them essential for various real-world applications such as social networks, transportation road networks, and e-commerce transaction networks. As graph data grows exponentially, many graph processing frameworks [18, 19, 34, 40, 57, 61] have been proposed to parallelize graph processing and reduce programming effort. Users often deploy graph frameworks on cloud servers using the Infrastructure-as-a-Service (IaaS) model to minimize hardware maintenance expenses for large-scale graph-based applications. To ensure good service quality (short query latency) [46, 47], users typically reserve servers based on peak load [45], and keep them running constantly (i.e., always-on policy) due to IaaS's limited elasticity.

However, this approach can lead to significant resource waste [24, 41] and cost-effectiveness issues when processing diverse workloads. Furthermore, the slow scaling of IaaS

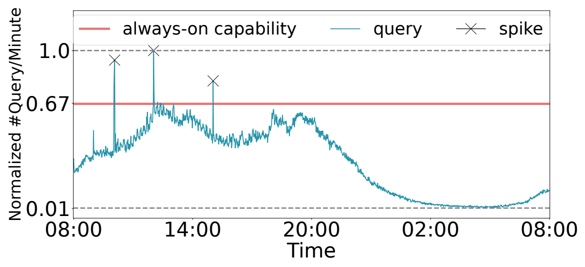


Figure 1. A typical pattern of normalized queries per minute on a city road network from our industry partner, displaying temporal distribution and occasional query spikes.

makes it incapable of handling query spikes. In Figure 1 showing a typical pattern of normalized queries per minute on a road network, the tail latency on the query spike around 12:00 increases from 1.5 seconds to over 30 seconds, and 85.7% of resources are wasted for one day (see Section 6.4 for details). An alternative approach is the on-demand policy that initiates servers upon request arrival. However, this method is hindered by the lengthy cold start latency. A potential enhancement to this policy is to dynamically scale VMs according to the prediction or profiling results derived from query workload. While this strategy can diminish query latency, it necessitates a prior understanding of workloads and substantial engineering effort to dynamically integrate servers into services, posing a practical challenge. Moreover, accurately predicting the occurrence of query spikes remains challenging. In summary, current IaaS graph processing engines are not elastic enough to adapt to the workload fluctuation.

The emerging serverless computing (Function-as-a-Service, FaaS) offers high elasticity and a pay-as-you-go billing model to users [26, 32, 55]. It has been successfully employed in lightweight services like web services [23] and workflows [33, 35, 36]. It inspires us to investigate whether graph processing can also benefit from the resource efficiency and cost-effectiveness of serverless computing. To this end, we conduct profiling studies by executing Gemini [61], a state-of-the-art graph processing framework within simulated FaaS functions. Our investigation (see Section 2.2 for details) reveals that merely running graph processing frameworks on a serverless computing system significantly degrades performance. For instance, on a graph with 1.8 billion edges, FaaS-based Gemini takes up to 28.2X more in-memory compute time than its server-based counterpart.

Our main insight is that the primary reason for poor performance is the mismatch between the data and communication-intensive nature of graph processing and the lightweight tasks targeted by the current FaaS architecture. Specifically, this mismatch leads to the following three problems.

First, graph processing requires substantial resources and involves iterative computations, with each iteration demanding extensive communication between workers. Owing to limited function resources, FaaS-based methods divide graphs into more partitions than server-based counterparts, increasing communication volume and worsening performance issues due to the poor communication capabilities of functions.

Second, I/O and preprocessing of large graphs result in considerable time costs, leading to an extended cold start for functions. When queries arrive in rapid succession (i.e., query spikes), serverless computing systems aggressively allocate new containers as existing containers are occupied by cold-start overhead. Consequently, serverless systems suffer from large memory footprint.

Third, existing graph processing frameworks optimize in-memory compute time by building auxiliary data structures [38], assuming servers have ample memory and run continuously. However, this approach leads to high preprocessing costs, dominating end-to-end query time in serverless computing with short-lived containers. Furthermore, memory costs cause scalability issues due to limited function resources.

In light of our findings, we propose **FaaSGraph**, a scheme that co-designs serverless architecture with graph processing frameworks. FaaSGraph serves as a graph-as-a-service solution targeted at cloud providers, enhancing the versatility of serverless. Users can easily create graph applications by uploading functions along with the corresponding graphs.

FaaSGraph employs a data-centric serverless execution model, introducing a fundamental unit (named *ContainerSet*) for a single heavyweight computing task. *Locality-aware resource fusion* breaks down the barriers between containers, enabling intra-host container resource sharing and shared memory communication. Meanwhile, inter-host containers communicate via the network with computation-communication interleaving and message consolidation. The bounded resource scaling policy employs a queuing method to limit container cold start rate, minimizing memory footprint while processing diverse workloads. Furthermore, we re-evaluate design choices in message passing and graph storage, minimizing auxiliary data structures to reduce memory consumption and preprocessing overhead.

Our extensive results demonstrate that FaaSGraph improves end-to-end performance by up to 8.3X, and reduces up to 52.4% of memory usage. Given the query pattern in Figure 1, FaaSGraph effectively handles fluctuating workloads while maintaining consistent 99%-ile performance, saving up to 85.7% of monetary cost. In summary, this paper makes three main contributions:

- Identifying the root causes of inefficient graph processing on serverless computing platforms.
- Designing a data-centric serverless model that bridges the gap between graph processing and serverless.
- Re-visiting the design choices of graph processing paradigms and developing appropriate ones that are suitable to integrate with serverless execution model.

2 Motivation of FaaSGraph

In this section, we investigate whether we can directly integrate graph processing frameworks with serverless platforms efficiently. The investigation motivates FaaSGraph.

2.1 Preliminaries

Graph Processing Frameworks. A graph $G = (V, E)$ is a data structure where V is a set of vertices and $E \subseteq V \times V$ represents the edges between vertices. *Graph processing frameworks* are designed to efficiently and easily process and analyze large-scale graphs. Broadly speaking, their computation can be abstracted as a process of updating states associated with vertices and propagating updates along edges. For example, in Gemini [61], a distributed framework, each node (server) holds a graph partition (subgraph). At each iteration, given a vertex, it first gathers states from its neighbors, updates its own state, and then passes the updated state to its neighbors. This process repeats until the stop condition is met. Users can develop various graph algorithms such as PageRank using APIs provided by Gemini, while Gemini automatically applies these functions to the graph in parallel.
Serverless Computing. In serverless computing, users create and deploy functions (i.e., code snippets) for specific tasks, while cloud providers manage underlying infrastructures (e.g., allocating and provisioning computing resources) and handle computation (e.g., scaling and load balancing). FaaS presents several limitations on functions. First, each function instance has restricted computing resources. Second, they have limited network bandwidth and cannot directly exchange data, relying on remote storage like file systems and databases for communication. Third, in multi-function coordinating scenarios (i.e., serverless workflows), the control flow for task scheduling must be predefined and submitted to serverless platforms.

2.2 Integrating Graph Processing and Serverless

A straightforward idea is to migrate graph processing frameworks onto serverless computing platforms by encapsulating user-defined APIs into functions. Following this idea, we select Gemini [61], a state-of-the-art graph processing framework, as a representative and adapt it to the FaaS model to investigate the performance.

2.2.1 Experimental Setup.

Configuration. We treat nodes in IaaS as functions in FaaS, with one function responsible for processing a graph partition. We modify Gemini to enable communication between functions through remote storage instead of networks, as functions are non-addressable. In summary, we evaluate Gemini under three different configurations.

- *VM*: Gemini running in VM environment (i.e., IaaS), with the graph dataset stored on the local disk.
- *<u>S</u>erverless-<u>R</u>emoteComm (S-R)*: The adapted Gemini runs in simulated FaaS functions by launching it in containers[1]. Containers communicate through remote storage and fetch their graph partitions from remote storage.
- *<u>S</u>erverless-<u>D</u>irectComm (S-D)*: Gemini running on containers with the same configuration as *S-R*, except they communicate through the network instead of remote storage.

We conduct experiments on a four-node cluster, each equipped with 24 cores and 96 GB of memory. Section 6.1 provides detailed hardware and software configurations. *VM* uses all four nodes, with each VM representing a node. In contrast, each container[2] has 2 cores and 3 GB of memory, following a common acknowledgment that serverless platforms hardly provide large-sized instance specifications [24, 41]. *S-R* uses 8 containers, as this configuration achieves optimal performance among configurations scaling containers from 8 (the minimum number of containers required to hold the graph in memory) to 48 (the maximum number of containers our experimental environment can hold). *S-D* uses the same number of containers as *S-R*.
Workload. In accordance with previous graph processing research [18, 40, 61], we evaluate four common workloads on the *friendster* (*fr*) graph [58]: PageRank (PR), Breadth-First-Search (BFS), Single-Source-Shortest-Path (SSSP), and Connected-Components (CC). *fr* consists of approximately 65.6 million vertices and 1.8 billion edges.
Pricing. In line with the billing practices of cloud providers, we estimate the monetary cost, denoted as m, for processing queries using the formula: $m = t \times w \times c$. t represents the end-to-end execution time, w stands for the total number of workers involved, and c is the per-worker cost. We can see that reducing latency and resource consumption can enhance cost-effectiveness.

In our experiment, we derive the value of c based on AWS pricing: the charge for utilizing a function equipped with 2 cores and 3 GB memory is \$0.00005 per second, equivalent to 3 × \$0.0000166667 per second per GB. AWS offers two 24-core VM options: one is *compute-optimized* with 48 GB memory (maintaining a CPU-to-memory ratio of 1:2), and the other is a *general-purpose* option boasting 96 GB of memory (with a CPU-to-memory ratio of 1:4). Given that all test cases manage within a 48 GB memory span, we opted for the compute-optimized price for a fair comparison, despite our test bed having a 96 GB memory capacity. The expenditure for a single VM node usage is \$0.000328 per second (i.e., \$1.18 per hour /3600). Unless stated otherwise, we adhere to the aforementioned pricing strategy throughout this paper.

[1]As Gemini relies on MPI, containers must be addressable to start up, which is not officially supported by current commodity serverless platforms. Therefore, we create an environment simulating serverless scenarios.
[2]In this paper, we execute a function within a container. As a result, for the sake of brevity, we will use the terms "container" and "function" interchangeably throughout the remainder of this paper.

Table 1. Breakdown of execution time (seconds) for *fr*.

Config-App	IO	Preprocess	Compute	Store	Query	Total
VM-BFS	24.6	88.6	1.3	0.3	114.7	135.4
S-R-BFS	35.6	376.3	21.0	0.8	433.8	434.6
S-D-BFS	33.6	105.5	4.1	0.8	143.9	144.7
VM-CC	24.6	96.1	3.9	0.3	124.8	145.5
S-R-CC	41.3	350.8	110.2	0.8	503.1	503.9
S-D-CC	37.2	106.5	28.3	1.2	173.2	174.0
VM-PR	26.3	86.9	14.1	0.5	127.8	148.5
S-R-PR	34.1	367.2	257.5	1.6	660.5	661.3
S-D-PR	32.7	110.0	115.5	2.0	260.2	261.0
VM-SSSP	37.7	71.4	7.1	0.3	116.4	137.0
S-R-SSSP	50.2	533.4	118.8	0.8	703.3	704.0
S-D-SSSP	54.7	117.3	58.5	0.8	231.2	232.0

2.2.2 Poor Performance of Direct Integration.

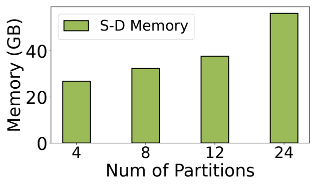
Latency Comparison. Table 1 presents the time for each stage: IO time (graph loading), Preprocess time (graph preprocessing), Compute time (in-memory computing), and Store time (result storage). The Query time is the sum of these four components, and the Total time additionally includes the Start-up time (preparation time for a cold start execution environment). In our experiments, the Start-up time for virtual machines is approximately 20.7 seconds, while for containers, it is approximately 0.8 seconds.

As observed, although VM exhibits shorter IO times than *S-R* and *S-D* due to faster local storage compared to remote storage, loading graphs incurs significant overhead for all three configurations. *S-R*, which communicates through remote storage, consumes up to 7.5X more Preprocessing time than VM. By enabling network communication, *S-D* significantly reduces Preprocessing time but remains slower than VM due to increased communication overhead from a higher number of workers. Nevertheless, in most cases for the three configurations, Preprocessing time dominates Query time.

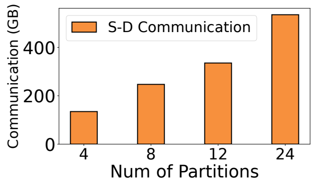
For Compute time, which existing graph computing techniques primarily optimize, *VM* achieves up to 28.2X and 8.3X speedups over *S-R* and *S-D*, respectively. This indicates that poor communication in FaaS significantly degrades performance. Since state data is considerably smaller than the graph, storing results takes very short time.

Overall, *S-R* consumes 4.8X more Query time than *VM* on average. Furthermore, Start-up time accounts for up to 15.3% of the total time, representing a significant resource initialization overhead. In summary, simply migrating graph processing frameworks to serverless computing systems can significantly degrade performance.

Impact of Fine-grained Partitioning. Due to limited function resources, the FaaS-based method divides a large-scale graph into more partitions than its IaaS-based counterpart. We examine the impact of fine-grained partitioning in Figure 2. As the number of partitions increases, communication volume grows dramatically. This result implies that the added communication overhead outweighs the benefits of



(a) Memory cost.



(b) Communication volume.

Figure 2. Impact of fine-grained partitioning on PR for *fr*.

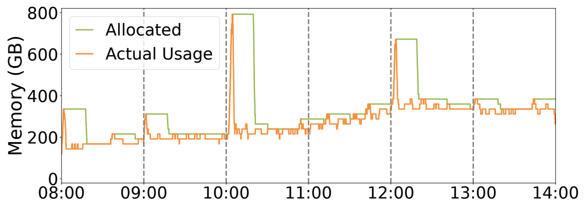


Figure 3. Impact of unbounded resource scaling on memory.

more computing resources. The memory cost from auxiliary data structures can also limit scalability and impact cost-effectiveness.

Monetary Cost Comparison. According to the pricing method in Section 2.2.1, *S-R* costs 1.1X-1.8X more money than *VM* on test cases in the experiments due to the long query latency although *S-R* manages computing resources in a fine-grained manner (8 containers with 2 cores and 3 GB memory). Thus, the direct integration of existing graph processing frameworks and serverless cannot enhance cost-effectiveness.

Impact of Unbounded Resource Scaling. Figure 3 illustrates the memory footprint of the FaaS-based method for PR on the *fr* graph using the pattern in Figure 1 (refer to Section 6.4 for detailed configuration). Unbounded scaling allocates new containers for queries when no warm container is available, reserving substantial resources during query spikes due to the lengthy cold-start time caused by data IO and preprocessing. Because of the keep-alive policy (i.e., keeping idle containers for 15 minutes), a substantial gap forms between occupied memory and actual memory usage. In total, 23.6% of memory resources are wasted.

2.3 Key Insights and System Implications

Based on above observations, we present the following key insights and system implications:

- Merely enabling network communication among functions cannot resolve performance issues due to increased communication volume caused by fine-grained partitioning. As such, an effective serverless computation paradigm should be designed to support this data-intensive application.
- While serverless computing systems can rapidly scale up to handle query spikes, unbounded resource scaling

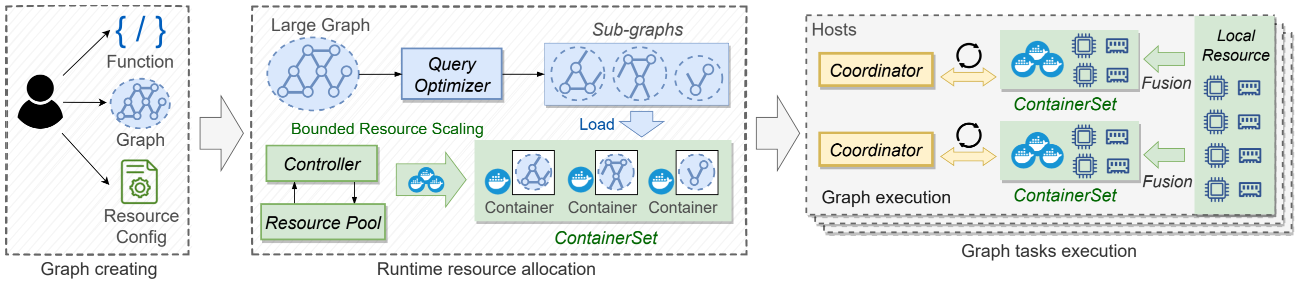


Figure 4. The execution flow of FaaSGraph.

results in a large memory footprint due to the excessive scaling incurred by long cold start time. Consequently, a new resource scaling policy is required to reduce the memory footprint.

- The preprocessing overhead of auxiliary data structures can dominate end-to-end query time, and its memory cost can limit scalability in serverless environments. Therefore, a balanced optimization approach for in-memory computing and its overhead is necessary for graph processing to seamlessly cooperate with serverless computing systems.

3 Serverless-native Graph Processing

Inspired by the implications in our investigation, we propose **FaaSGraph**, a co-designed scheme that combines graph processing and serverless architecture.

3.1 Subgraph-centric Graph Processing

FaaSGraph utilizes the subgraph-centric model [18], treating the processing of subgraphs as a parallel task. Specifically, the graph is divided into a set of non-overlapping *partitions*, each allocated to a container for processing. FaaSGraph employs the bulk synchronous parallel model to process graphs iteratively. In each iteration, FaaSGraph applies the user-defined functions to subgraphs in parallel to update vertex states, with messages exchanged between different partitions to synchronize these states. FaaSGraph repeats the iterative computation until convergence, after which the results are returned. We opt for the subgraph-centric processing because previous studies [18, 57] have demonstrated that coarse-grained processing can reduce communication overhead compared to the vertex-centric model. Programming with FaaSGraph is similar to that of previous frameworks [57], following the "think-like-a-subgraph" method. Due to space limitations, we omit the details of APIs.

3.2 Data-centric Serverless Execution

Moreover, during execution, the default *task-centric* model of serverless computing is not suitable for graph applications. Using this model, an application is decomposed into several tasks, each responsible for a different operation and managed by an individual function. The execution of these tasks follows a predefined workflow graph delineating their interdependencies. However, a graph application has a single task forked to manage various data partitions when the workload exceeds the computing capacity of a single function. The same operations are done on different data partitions, with communication being a critical requirement. To bridge the gap between serverless computing and graph processing, we propose *data-centric serverless execution.*

Resource Abstraction. Throughout the graph processing, the large graph remains immutable, while the states associated with each vertex are updated. Drawing from this observation, we propose data-based resource abstraction to tackle the challenges that a graph task may exceed the computing capability of a container. Specifically, we encapsulate a set of containers into a unit called *ContainerSet* where these containers can spread across multiple nodes. Each container is responsible for processing a subgraph. The *ContainerSet* serves as the basic unit for allocating and releasing resources.

Execution Flow. Based on the above designs in principle, Figure 4 depicts the execution flow in FaaSGraph. Users develop graph algorithms using FaaSGraph's interfaces, specifying their functions, graph datasets and resource specifications. Upon query reception, FaaSGraph engages the *Query Optimizer* to ascertain the computing resources required for graph processing. According to this assessment, the *Controller* allocates a *ContainerSet* from the resource pool based on the *bounded resource scaling* policy (Section 4.2), which address the resource waste stemming from the lengthy cold start time. Each container is responsible for a graph partition, which it retrieves from the remote storage where the entire graph is stored in a single file.

Upon the allocation of a *ContainerSet*, FaaSGraph triggers user-defined functions to process subgraphs in parallel. The *Controller* assigns a *Coordinator* for each *ContainerSet* to synchronize the computation of containers. Moreover, FaaSGraph introduces *locality-aware resource fusion* to facilitate frequent vertex state exchanges between containers within a *ContainerSet* at each iteration. After the task is completed, FaaSGraph stores the outcomes in a remote storage and communicates the results to the user through a response.

4 Serverless Architecture Optimizations

In this section, we propose *locality-aware resource fusion* and *bounded resource scaling*, enhancing the data-centric serverless execution by improving the communication, computation and resource scaling capabilities.

4.1 Locality-Aware Resource Fusion

In the task-centric model, containers are isolated to ensure security, resource allocation, fairness, and computational separation, as they serve different tasks and functions. Communication between containers is typically limited to remote storage access. Due to container resource constraints, more containers are needed compared to traditional VMs to achieve equivalent computing resources, such as CPU cores and memory. The increased communication and load imbalance resulting from resource isolation lead to significant performance degradation.

In contrast, our *ContainerSet* abstraction encapsulates these containers into a single unit to serve a specific task. We develop a *locality-aware resource fusion* mechanism that breaks the resource wall of containers in a *ContainerSet*. Figure 5 illustrates resource fusion. With *local resource fusion*, containers on the same host efficiently share resources, utilizing the same set of CPUs and memory segment. *Remote resource fusion* reduces communication overhead by consolidating messages to remote containers on the same host and transmitting them asynchronously.

Local Resource Fusion. The potential localities of containers on the same server offer opportunities for resource sharing, thereby accelerating communication and fully utilizing computing resources. We create a shared memory buffer for containers within a *ContainerSet* on the same host, which serves as an efficient communication channel, bypassing remote storage. The buffer size is determined by a configurable hyperparameter and currently limited to 300 MB to minimize host overhead. To share computing resources, FaaSGraph allows idle CPUs in one container to be utilized by others within the same *ContainerSet* to balance computation workload. We set up a shared Linux cgroup for containers within the same *ContainerSet* located on the same host to enforce CPU usage restrictions.

Remote Resource Fusion. To break the resource wall among containers located on different hosts, we develop a *Proxy* that works in tandem with local resource fusion. When a *ContainerSet* is allocated, *Proxy* is established for containers on the same host, with its address distributed to every container in the set. *Proxy* receives messages from remote containers and writes them to shared memory, ensuring all containers on the same host can access updates. It merges requests for containers from the same host, which reduces communication cost compared to that containers directly communicate with each other. The overhead associated with *Proxy* is negligible,

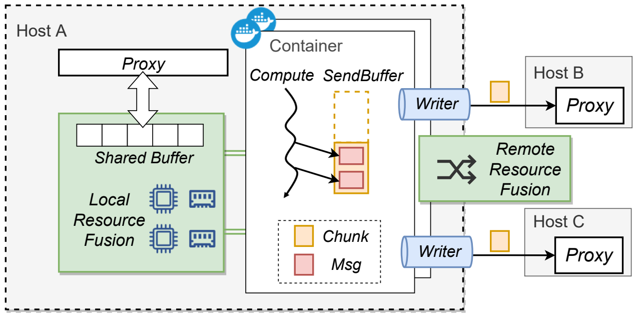


Figure 5. Locality-aware resource fusion. Containers on the same host communicate through a shared memory buffer and balance CPU cores under cgroup restrictions. Containers communicate with remote hosts by consolidating messages as a chunk via *Proxy*. Writers run asynchronously alongside computing threads using coroutines.

as its primary function is to serve as a proxy for transferring data from the network to shared memory. Conversely, each container maintains a SendBuffer, which, when full, triggers writer coroutines to asynchronously send message blocks to the corresponding *Proxys*. This minimizes context switch costs and allows for overlap of communication and computation.

Bin-packing Resource Allocation. When creating a *ContainerSet*, the locality of containers influences the communication efficiency. We adopt a *bin-pack resource allocation* strategy, treating the problem of allocating containers across hosts as a bin-pack problem, with containers as items and hosts as bins. Our goal is to minimize the number of hosts for a *ContainerSet*. First, the strategy tries to find the least loaded host that can accommodate the entire *ContainerSet*; if it fails, the scheduler tries to distribute the *ContainerSet* across two hosts, and so on. When allocating containers across multiple hosts, our controller aims to distribute them evenly to minimize load imbalance. Other schedulers that favor potential locality are also applicable and orthogonal to FaaSGraph. [25, 30, 42, 46, 51].

Potential Security Issues of Resource Fusion. Though containers in a *ContainerSet* are not directly linked, they are connected indirectly through the *Coordinator* and *Proxy* in the control-plane and data-plane. In this case, a low overhead authentication mechanism (e.g., AWS Identity and Access Management (IAM) [3]) could be used in both components to prevent malicious vertex triggering and messaging. To safeguard the shared memory segment, Linux permission system is used to prevent unauthenticated memory attaching. Moreover, secure containers such as Firecracker [6], RunD [31], and Kata Containers [8] can prevent untrusted guestOS operations from compromising the hostOS.

4.2 Bounded Resource Scaling

The task-centric model employs an *unbounded scaling* strategy, which allocates a new container when no warm container is immediately available upon request arrival [4]. While this on-demand resource allocation works well for lightweight tasks, it can cause significant issues for graph processing due to the extended cold start times, especially when workloads exhibit high variability. Excessive containers are initialized because previous containers are trapped in the cold start, causing a significant upsurge in memory footprint, which is costly in data centers. Moreover, cold start containers must load large amounts of data, resulting in bandwidth contention for both storage and network resources.

To tackle this issue, we propose the *bounded resource scaling* method. In FaaSGraph, the *ContainerSet* serves as the fundamental unit for allocating and releasing resources. Upon receiving a query, if there is no *idle ContainerSet*, bounded scaling checks whether it can wait for an *occupied ContainerSet* that is in-use instead of allocating a new one. The *Controller* maintains a first-in-first-out (FIFO) queue for each *occupied ContainerSet*, grouping them based on the graph they are processing. It first identifies *ContainerSet* processing the same graph as the query, then employs the Join-Shortest-Queue policy to select the shortest queue to join. If all queues reach a capacity threshold c, a new *ContainerSet* is allocated.

To manage latency caused by waiting, the capacity threshold c controls the maximum number of queries that can wait for a *ContainerSet.* Intuitively, increasing c reduces memory footprint at the expense of increased queuing time. Further optimization of the queuing policy is enabled when retrieving recent query performance to dynamically adjust c. Specifically, a graph query selects the queue with the least workload and ensures that the waiting time T_W in the queue does not exceed the cold start time T_C (i.e., the time for startup and loading graphs) using Equation 1, where Q represents the queue and T_q denotes the execution time of q. In other words, this approach theoretically indicates that the execution time of q will not be worse than initiating it with a cold start. If no queue satisfies the condition, a new *ContainerSet* will be allocated to serve the query.

$$T_W = \sum_{q \in Q} T_q \leq T_C. \tag{1}$$

5 Graph Processing Optimizations

In this section, we introduce the graph-side optimizations in FaaSGraph, which collaborate with the data-centric model to accelerate queries and reduce resource consumption.

5.1 Query Optimizer

Traditional frameworks typically utilize all available resources in a cluster. However, this can lead to severe cost-effectiveness issues [39]. Therefore, we design a *Query Optimizer* to determine the number w of containers in a *ContainerSet* for a given query q on graph G.

We prioritize memory metrics in optimizing w. Using the memory demands of a query and a container's memory capacity, we calculate the least number of containers necessary for the task. This design rationale is rooted in maintaining cost-effectiveness; a mere expansion of containers not only escalates resource utilization but also generates increased communication overhead, stemming from a higher number of partitions. Equation 2 shows the detail. $M(G)$ is the memory consumption of G, $M(C)$ is the memory size of a container C, and $\alpha(q)$ is the memory consumption amplification factor incurred by auxiliary data structures. In our experiments, we set $M(C)$ to 3 GB. $\alpha(q)$ depends on the actual algorithm and can be easily determined by graph algorithm developers. In our experiments, we set $\alpha(q)$ to 1.45, which covers algorithms studied in this paper.

$$w = \lceil \frac{M(G) \times \alpha(q)}{M(C)} \rceil \tag{2}$$

5.2 Message Passing Mechanism

Given the target number w of partitions, we divide graph G with the edge-cut strategy [38]: Split vertex set V into w disjoint sets $V_0, V_1, ..., V_{w-1}$, and a partition contains all edges adjacent to vertices in V_i. For a partition G_i, we refer to vertices in V_i as *main vertices* and vertices not in it as *mirror vertices.* Each vertex $v \in V$ has exactly one main vertex in one partition, which maintains the state associated with it during computation. As each container holds a partition, we need to pass states among different partitions.

Prior frameworks, e.g., Ligra [49] and Gemini [61], use a *dual-mode* message passing technique. They alternate between push and pull-based patterns across various iterations during query processing, leveraging the active set density to minimize execution latency. Particularly, Gemini utilizes mirror vertices to facilitate updates through adjacent edges. While it works efficiently on clusters, it necessitates an array of size $O(|V|)$ to index the neighbors of each vertex within a partition. Moreover, it stores both incoming and outgoing edges for every vertex in that partition due to mixed push and pull operations. This causes severe issues in serverless environments: 1) the memory overhead can limit scalability and incur extra monetary cost, and 2) constructing the array leads to significant preprocessing overhead.

FaaSGraph simplifies the dual-mode and adopts the *Main-Initiated message passing*: main vertices are responsible for transiting updates, and users exclusively adopt push or pull-based pattern. Therefore, we only need to index neighbors for main vertices in one direction, reducing the memory cost of index for each partition from $O(|V|)$ to $O(|V_i|)$, storing incoming or outgoing edges and eliminating preprocessing overhead. Using partitions G_0 and G_1 in Figure 6 as an example, the Main-Initiated message passing requires $v2$ in G_1 to

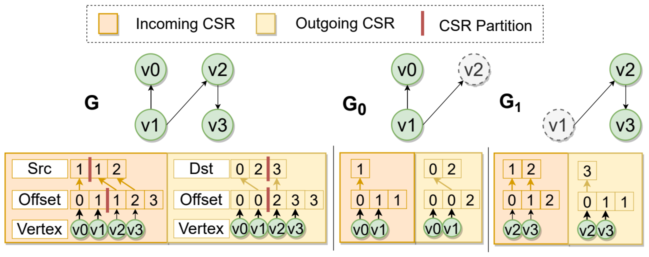


Figure 6. An example of graph storage and partitioning in FaaSGraph, where the vertex array is divided into two contiguous chunks: [0, 1] and [2, 3]. G_0 and G_1 are stored in two CSRs: one for in-neighbors (Src) and another for out-neighbors (Dst). The Src and Dst arrays store neighbors of vertices, while the Offset array records the neighbor set offset for each vertex. Green vertices represent main vertices, and grey vertices indicate mirror vertices.

pass its state to its outgoing neighbors (when using push). Thus, G_1 only needs to maintain the index array (i.e., offset) for main vertices $v2$ and $v3$. In contrast, with the message passing model in Gemini, G_1 must store the index array for all vertices, as $v1$ needs to know its outgoing neighbors to push the state. It also needs to maintain both incoming and outgoing edges to support edge traversal of both directions.

Prior work [49, 61] and our experimental results in Section 6.3.1, demonstrate that the effectiveness of push- and pull-based patterns is contingent upon the particular graph applications. Luckily, users can easily choose the most suitable pattern with just a few trial runs. By default, FaaSGraph operates exclusive by push or pull to reduce resource consumption. For developers prioritizing high performance, FaaSGraph retain the option to use dual-mode message passing under Main-Initiated message passing context.

5.3 Graph Storage and Preprocessing

We store graph $G = (V, E)$ using the popular Compressed Sparse Row (CSR) structure due to its compact format, which both reduces memory consumption and utilizes memory locality for computation [16]. We store incoming CSR and outgoing CSR of a graph in two separated files. We employ a chunk-based method [61] that divides V into w continuous chunks because it can easily partition G given w at runtime and requires negligible partition overhead.

The storage and partition methods working together with the Main-Initiated message passing mechanism can eliminate the graph preprocessing cost to accelerate the query. Specifically, given the number w of partitions, FaaSGraph can directly load edges from the CSR file by adjusting the index offset of vertices because the Main-Initiated message passing mechanism only requires to index the main vertices. For instance, while processing the graph depicted in Figure 6 utilizing two partitions and the push-based pattern, FaaSGraph is able to directly access the neighbor sets (i.e., Dst arrays) from the outgoing CSR file of G by remapping the main vertices' offset.

Table 2. Hardware, software setup and benchmarks.

	Configuration			
Hardware	CPU: Intel Xeon(Ice Lake) Platinum 8369B @3.5GHz Cores: 24, DRAM: 96GB, NAS: bandwidth 300MB/s			
Software	OS: Linux with kernel 5.15.0 Golang: 1.18.4, Docker: 20.10.17			
Container	Container Runtime: Golang 1.18-alpine Resource Limit and Lifetime: 2core, 3G, 15min			
Benchmark	Graph	Vertices	Edges	CSR Size
	amazon(am)	334,863	925,872	18Mb
	livejournal(lj)	4,847,571	68,993,773	1.1Gb
	twitter(tw)	41,652,230	1,468,364,884	23Gb
	friendster(fr)	65,608,366	1,806,067,135	28Gb
	road(rd)	3,996,221	4,246,845	176Mb

6 Evaluation

In this section, we evaluate FaaSGraph's performance.

6.1 Implementation and Experimental Setup

We implement FaaSGraph in roughly 2,000 lines of Golang [7] code[3]. FaaSGraph runs on Linux platforms. We employ Docker [5] to create containers and use NAS (Network Attached Storage) [1] for remote storage. We rely on HTTP to facilitate communication among containers. Table 2 describes the hardware and software setups.

Baselines. In addition to FaaSGraph, we also examine several state-of-the-art distributed graph processing frameworks, Gemini [61], Graphite [40], and Graphscope [17]. We also include the serverless-restricted Gemini (*S-R* and *S-D* configuration from Section 2.2) as a baseline. Additionally, we choose Graphit [60], a single-machine framework, to represent shared-memory solutions in our comparison.

Graph Benchmarks. Following previous research [18, 40, 61], we use four representative graph datasets [28], including Amazon, LiveJournal, Twitter, and Friendster, as well as four common graph algorithms: PageRank (PR), Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), and Connected Components (CC). To maintain consistency with prior work [61], we run 20 iterations of PR and execute other applications until convergence. Furthermore, we use the city road network in our case study. Table 2 provides details on these graphs range from 0.9 million to 1.8 billion edges.

Pricing Strategy. We adhere to the pricing strategy delineated in Section 2.2.1. Unlike FaaS and on-demand execution in IaaS, the always-on execution cost is fixed since the hardware is constantly running. For FaaSGraph, we use the *ContainerSet* as the basic unit for billing since all containers are kept alive during processing.

[3]https://github.com/ziliuziliu/FaaSGraph

Table 3. Comparison of *S-R*, Graphite(GH), Gemini(GE), Graphit(GI), Graphscope(GS) and FaaSGraph(FG) in terms of Total time (in seconds).

Graph-App	*S-R*	GH	GE	GI	GS	FG
am-BFS	3.3	22.4	21.2	20.7	21.7	0.9
am-CC	4.3	22.2	21.4	20.7	21.8	0.8
am-PR	5.7	22.4	21.6	20.7	22.2	1.0
am-SSSP	3.5	22.1	21.2	20.7	22.0	0.8
lj-BFS	87.7	26.0	23.7	23.4	28.5	2.8
lj-CC	41.7	24.7	24.1	23.5	28.5	3.2
lj-PR	59.7	25.8	24.3	23.8	29.8	7.1
lj-SSSP	57.5	26.2	25.5	25.4	29.2	5.4
tw-BFS	334.3	127.5	89.3	119.2	192.7	45.9
tw-CC	370.3	104.1	87.4	122.2	195.5	50.4
tw-PR	524.4	114.3	90.0	143.3	219.9	68.2
tw-SSSP	512.2	138.3	115.0	163.3	202.4	68.6
fr-BFS	434.6	153.0	135.4	119.6	216.0	58.6
fr-CC	503.9	124.1	145.5	120.7	215.2	67.0
fr-PR	661.3	141.3	148.5	130.4	252.2	107.3
fr-SSSP	704.0	170.2	137.0	170.7	213.9	100.5

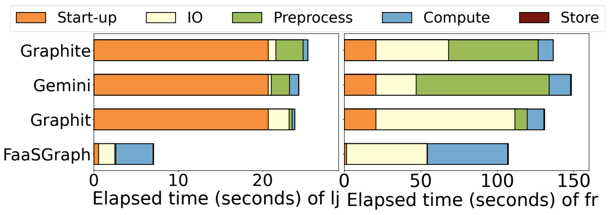


Figure 7. Time breakdown of execution of PR on *lj* and *fr*, comparing FaaSGraph with Gemini and Graphite.

6.2 Overall Comparison

We begin with a general comparison of one-shot queries. Unless specified, we compare FaaSGraph with Graphite, Gemini, and Graphscope on a 4-node cluster, detailed in Table 2. Graphit, being a shared-memory framework, operates on a single node with 96 cores and 384 GB of memory, maintaining equal computing resources. VM-based frameworks utilize all resources in the cluster, while resource allocation for FaaSGraph is handled by the *Query Optimizer*.

Time. Table 3 presents the experiment results in the on-demand manner, where resources are initiated from scratch. As observed, FaaSGraph outperforms its counterparts in all cases, demonstrating its performance efficiency. For small graphs like *am*, where serverless-based solutions are efficient due to low start-up time, FaaSGraph runs 3.6X-5.7X faster than *S-R*; when handling medium and large-sized graphs, FaaSGraph runs 1.2X-8.3X faster than IaaS-based solutions.

In more detail, Figure 7 provides the breakdown of time components, using PR on *lj* and *fr* for brevity. In *lj*, the total processing time with FaaSGraph is shorter than the Start-up time of Gemini and Graphite, demonstrating the advantages of FaaS over IaaS on lightweight workloads. In *fr*, Gemini and Graphite suffers from long Preprocessing time, which dominates the Total time. Graphit has slow IO operation. FaaSGraph outperforms Gemini, Graphite and Graphit even with a longer in-memory compute time.

Table 4. Speedup achieved by FaaSGraph over *S-R* in Compute time.

Graph	*lj*				*fr*			
App	BFS	CC	PR	SSSP	BFS	CC	PR	SSSP
Speedup	4.5	4.1	5.2	5.1	2.8	7.8	4.9	4.3

Table 5. Comparison of memory cost (in GB).

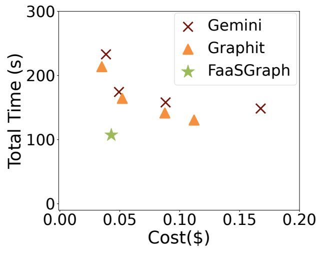
Graph-App	S-R	GR	GE	GI	GS	FG
lj-BFS	2.2	3.0	1.1	1.5	4.6	0.7
lj-CC	2.2	2.1	1.3	1.5	4.6	0.8
lj-PR	2.4	9.8	1.6	1.5	4.5	0.8
lj-SSSP	3.1	4.1	2.2	2.2	6.5	1.2
fr-BFS	23.9	82.7	20.8	29.6	111.6	16.5
fr-CC	25.6	46.4	21.1	29.6	112.4	16.1
fr-PR	27.2	152.3	23.1	29.6	113.4	18.8
fr-SSSP	38.7	63.7	34.8	57.7	157.7	27.7

Table 4 presents the speedup of Compute time, comparing FaaSGraph with *S-R*. FaaSGraph runs 2.8X-7.8X faster than *S-R*, highlighting the effectiveness of our data-centric serverless execution model.

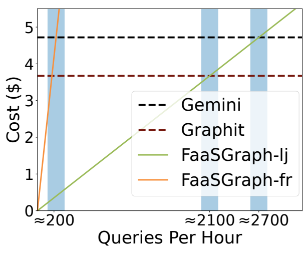
Lightweight container isolation mechanisms can offer lower start-up overhead, with RunD [31], for instance, boasting a start-up latency of 0.2-2 seconds. Leveraging these MicroVMs can substantially enhance performance for small graphs like *am* and *lj*, where the start-up time is a dominant factor of total time. Despite that, FaaSGraph generally outperforms counterparts on small graphs. However, the benefits are marginal for larger graphs, as the start-up latency constitutes a small portion of the total time. FaaSGraph runs faster than other frameworks in these cases.

Memory Consumption. Table 5 reports the memory costs associated with each method. In all cases, FaaSGraph consistently consumes the least amount of memory, significantly reducing the costs of processing graph queries. Notably, FaaSGraph uses up to 52.4% less memory than Gemini. In contrast, Graphit, Graphite and Graphscope requires substantially more memory compared to its counterparts. It demonstrates the effectiveness of our graph processing optimizations.

Monetary Costs. To assess the cost-effectiveness across various resources, we examine the expense of queries for PR on *fr*. Graphite and Graphscope are omitted from this comparison due to their inferior performance. For Graphit, We include single machine setups in terms of core-GB: 16-64, 32-64, 64-128 and 96-192. For Gemini, we explore distributed setups in terms of core-GB: 4-16, 8-16, 16-32, 24-48, each across 4 nodes. Note that when utilizing same amount of resource, the distributed performance in Gemini is better than its single-machine counterpart due to more IO bandwidth.



(a) On-demand (One-shot query)



(b) Always-on (Batch of queries)

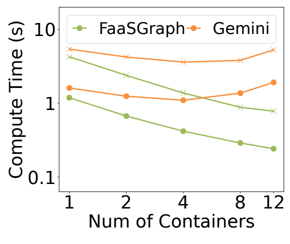
Figure 8. Monetary cost analysis.

Figure 8a showcases the monetary cost in the on-demand manner. FaaSGraph stands out by achieving an optimal balance of speed and cost. When compared to the most affordable VM configuration, it not only matches the monetary expense but also operates 1.99X faster. Moreover, it competes with the speed of the fastest VM setup while cutting the cost by 62.5%. Figure 8b compares the monetary cost of three competing frameworks in the always-on manner for application PR. For Gemini, we follow the default pricing setting in Section 2.2.1. For Graphit, the cost is fixed at \$0.00102 (i.e. \$3.672 / 3600, with core-GB 96-192) per hour. The price of FaaSGraph depends on the number of queries processed. The time of processing a query is equivalent to the Compute time in Table 3. Although VM-based frameworks have shorter compute time, they incur significantly higher expenses in low workload. When the workload becomes heavy (i.e., at the intersection point), the cost of FaaSGraph surpasses other frameworks.

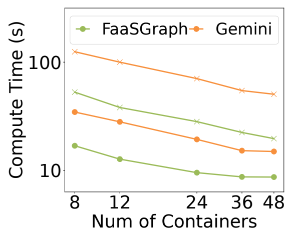
In summary, FaaSGraph is an ideal choice for both highly variable workloads and lighter workloads, a common scenario in data processing [41], such as when data scientists analyze data interactively. For heavier workloads with high computing demands and a focus on throughput, traditional methods in the always-on paradigm still retain their advantages due to their in-memory computing performance.

6.3 Detailed Evaluation of FaaSGraph

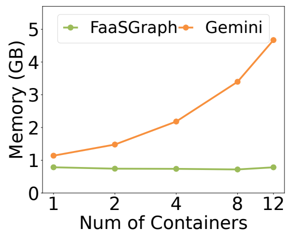
In this subsection, we delve into evaluations for each design detail. We choose Gemini as our primary baseline due to its efficient memory usage, high performance, and ability to operate in a distributed environment. We begin by comparing FaaSGraph with Gemini running on containers that enable network communication (i.e., *S-D* in Section 2.2). Since they use the same amount of resources and operate on containers with limited resources, these experiments demonstrate the effectiveness of our co-designed solution in a serverless environment. Subsequently, we investigate the effectiveness of our data-centric serverless execution model. Lastly, we conduct an ablation study to demonstrate the impact of graph processing optimization and data-centric serverless execution mechanisms on cost-effectiveness.



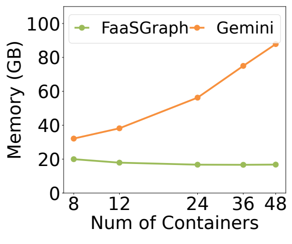
(a) Compute time on *lj*.



(b) Compute time on *fr*.



(c) Memory cost on *lj*.



(d) Memory cost on *fr*.

Figure 9. Scalability comparison of Gemini in Containers (*S-D*) and FaaSGraph on PR as the number of containers increases.

6.3.1 Graph Processing Optimizations.

Scalability. Figure 9 shows the compute time and memory consumption as the number of containers increases. For the medium graph *lj*, the number of containers scales from 1 to 12 within a single node, while it scales from 8 to 48 when spanning across 1 to 4 nodes processing *fr*. As seen in Figures 9a and 9b, FaaSGraph runs significantly faster than Gemini and achieves good scalability, demonstrating the effectiveness of our co-designed solution. FaaSGraph attains nearly linear scalability on *lj*, while Gemini experiences severe performance issues as the number of containers increases. This is due to our local resource fusion that eliminates network communication on a single node and maintains load balance through CPU sharing. The communication overhead in Gemini limit its scalability on lightweight queries.

Figures 9c and 9d display the memory cost when increase the number of containers. Gemini consumes a significant amount of memory because each container requires an array of size $O(|V|)$ for indexing. In contrast, the memory cost of FaaSGraph keeps steady because we reduce the memory cost of index from $O(|V|)$ to $O(|V_i|)$ as discussed in Section 5.2. This demonstrates the effectiveness of our optimization in reducing memory costs.

Query Optimizer. Table 6 presents the cost of processing queries using FaaSGraph in terms of memory footprints (GB-seconds). The row marked with bold font represents the configuration selected by our query optimizer. It is evident

Table 6. Memory footprint (GB-seconds) of FaaSGraph on PR as #containers increases. Warm start corresponds to Compute time, and cold start corresponds to Total time.

Num	*lj* Cold Start	*lj* Warm Start	Num	*fr* Cold Start	*fr* Warm Start
1	**7.4**	**4.2**	**8**	**2544**	**1266**
2	11.1	4.8	12	3331	1370
4	18.8	5.5	24	5907	2028
8	35.9	7.0	36	8178	2417
12	59.2	9.3	48	10671	2826

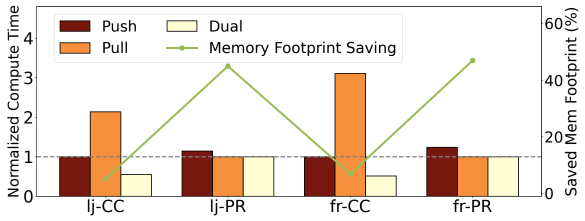


Figure 10. Effects of message passing method in Compute time and memory footprint. Each setting's Compute time is normalized by FaaSGraph's default performance (push for CC, pull for PR). Green line refers to the percentage of memory footprint saved by adopting single message passing method compared with dual mode.

that the *Query Optimizer* of FaaSGraph achieves the best cost-effectiveness for both cold start and warm start.

Message Passing Pattern. We evaluate the impact of different message passing mechanisms on both the computational time and memory footprint (which is Compute time multiplies memory consumption) of FaaSGraph, as shown in Figure 10 through the results for CC and PR. We implement push, pull and dual mode under *Main-Initiated Message Passing*. We use default resource settings in Section 6.2. As depicted, although dual-mode enhances performance, it essentially doubles memory consumption by storing edges in both directions. As a result, FaaSGraph saves memory footprint by 5.1%-47.0% by exclusively taking push or pull-based message passing method.

6.3.2 Data-Centric Serverless Execution.

Resource Fusion. We begin by evaluate the impact of container locality on performance within a *ContainerSet* to assess the effectiveness of resource fusion. Due to space constraints, we use PR on *fr* with 12 containers as a representative example. Figure 11 illustrates the experimental results for Compute time and the volume of network communication. As observed, FaaSGraph benefits from local resource fusion most when all containers are co-located on the same node, introducing no network traffic. The total communication relies on the number of nodes rather than the number of containers involved, as resource fusion eliminates communication on the same node and consolidates messages destined to same node.

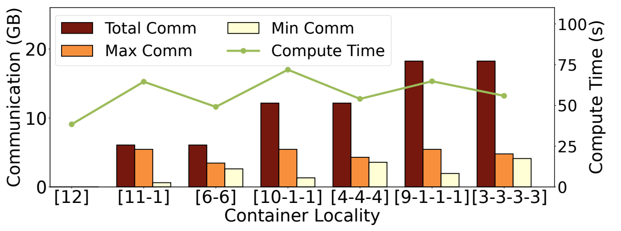


Figure 11. The impact of container locality for PR on *fr*. [a-b-c-d] represents the distribution of 12 containers across four nodes. Total, Max and Min Comm represents total, maximum and minimum communication volume across all nodes, respectively.

Table 7. The speedup of FaaSGraph for PR on *lj* and *fr* when enabling CPU sharing with varying numbers of containers.

Graph	*lj*				*fr*	
#Containers	2	4	8	12	8	12
Speedup	1.13	1.32	1.38	1.32	1.60	1.79

We can also find that 1) the configurations with fewer nodes perform faster due to reduced communication (e.g., [6-6] vs. [4-4-4]), and 2) when FaaSGraph utilizes the same number of nodes, evenly distributing containers across these nodes leads to better performance (e.g., [6-6] vs. [11-1]) because of balanced communication. These results demonstrate the effectiveness of the bin-packing resource allocation in FaaSGraph, as the method prioritizes minimizing the number of nodes used by a *ContainerSet* and prefers evenly distributing containers among nodes.

Table 7 illustrates the effectiveness of CPU sharing in local resource fusion. To eliminate the impact of communication, all containers are placed on the same node. We can see that enabling CPU sharing significantly improves performance, particularly when there are more containers involved. This improvement can be attributed to the fact that dividing graphs into many small partitions can lead to severe load imbalance issues, where CPU sharing effectively addresses this problem.

Bounded Resource Scaling. This experiment evaluates the effectiveness of bounded resource scaling with CC and PR on *fr*. We randomly synthesize the query trace according to the graph processing query pattern from 8:00 to 14:00 in Figure 1 as the workload. In the experiments, we perform the simulation because the required computing resources significantly exceed our experimental environment. In the simulation, the end-to-end execution time for a cold start and warm start on CC are 14.1 and 67.0 seconds, respectively, while on PR, they are 52.4 and 107.3 seconds. All latencies are consistent with our experiments in Section 6.2.

Table 8. The impact of bounded resource scaling as queue capacity c increases. Mem Reduced represents the memory footprint reduction compared to unbounded resource scaling ($c = 0$). Cold Start indicates the number of cold start queries. Avg, P99, and Max represent the average, 99%-ile, and maximum query latency (seconds), respectively.

App	c	Mem Reduced	Cold Start	Avg	P99	Max
CC	0	0	44	14.6	67.0	67.0
	1	17.4%	25	15.0	27.4	67.0
	2	18.1%	25	16.0	41.4	67.0
	4	21.1%	53	17.3	67.0	70.2
PR	0	0	73	53.3	107.3	107.3
	1	14.1%	47	59.4	104.6	107.3
	2	13.8%	460	70.0	156.0	157.2

Table 8 shows the experiment results. By increasing c, memory footprint can be reduced by up to 21.1%. When c is small, the number of cold start queries decreases as potential cold starts are replaced by queue waits. However, for larger c values, the waiting time may surpass the cold start overhead, resulting in better latency performance for unbounded scaling. Nevertheless, maximum latency can be theoretically guaranteed by adjusting c according to Equation 1.

6.3.3 Ablation Study. We conduct an ablation study to analyze the effectiveness of optimizations on both serverless and graph sides. Our baseline is *S-R*, the integration of Gemini with serverless architecture. We compare this with two FaaSGraph settings: one where resource fusion is disabled, referred to as *FaaSGraph-Unfusion*, and one where all optimizations are enabled, simply referred to as FaaSGraph. We also report the performance of Gemini for reference. The experiment environment is the same as that in Section 6.2.

Figure 12 presents the results of *fr*-CC and *fr*-PR. It reveals that graph-side optimizations are important in accelerating queries and reducing monetary costs, primarily by eliminating expensive preprocessing. Comparing FaaSGraph-U with *S-R*, the monetary cost is reduced by up to 83.7% because preprocessing dominates the Total time. When resource fusion is enabled, it fosters a reduction of up to 48.0% in monetary costs by further speeding up queries by 1.9X times, when comparing FaaSGraph-U with FaaSGraph.

6.3.4 Overhead of *Coordinator* and *Proxy*. We assess the overhead of the *Coordinator* and *Proxy* by measuring their memory consumption and the volume of network communication during PR tests on *lj* and *fr*, utilizing the experimental setup described in Section 6.2. In the *lj*-PR test, each iteration involved 1.2 MB of communication, which constituted 1.9% of Compute time. The memory usage for the *Coordinator* and *Proxy* was 10.4 MB and 11.7 MB, respectively. Meanwhile, the *fr*-PR test has a communication volume of 125.1 MB per iteration for the *Coordinator*, accounting for 3.4% of the compute time, and memory usage of 135.9 MB for the *Coordinator* and 13.7 MB for the *Proxy*, representing a mere

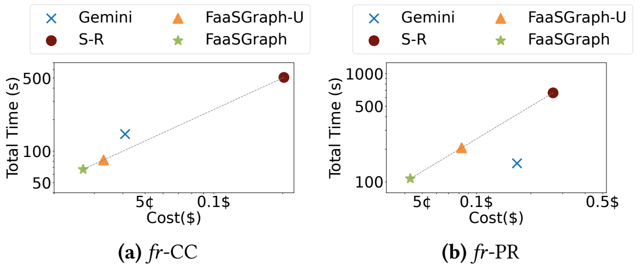


Figure 12. Ablation study on the graph-side and serverless-side optimizations.

0.7% of the total memory usage. Overall, the *Coordinator* and *Proxy* introduced minimal overhead in FaaSGraph.

6.4 Case Study

We compare the performance of Gemini and FaaSGraph using the query pattern in Figure 1. Due to privacy regulations, we cannot access detailed information about each query (e.g., query type and source vertex), so we randomly synthesize the query trace according to the workload pattern, and use BFS as a proxy due to its similar computational workload.

Setup. For FaaSGraph, we set the queue capacity to 1. Its Compute time and Total time are approximately 0.67 and 1.54 seconds, respectively. For Gemini, as Gemini achieves optimal performance ($\approx$ 1.2s Compute time) with 16 cores for a query, a Gemini instance is configured with 16 cores and 32 GB of memory. For the cluster size of Gemini instances, two configurations are included:

- *Gemini-Unscaling*: we set up 34 Gemini instances operating throughout the day, providing a processing capability of approximately 1700 queries per minute, which exceeds the throughput for 99% of the day.
- *Gemini-Autoscaling*: we dynamically adjust #VMs every 10 minutes, assuming that we have a priori knowledge of the pattern of queries per minute in Figure 1. However, we do not account for query spikes, as they are difficult to predict.

Figure 13a shows the 99%-ile latency in rush hour (11:00 to 13:00). *Gemini-U* remains steady before the query spike, but once the spike occurs, the tail latency increases from around 1.5 seconds to more than 30 seconds. *Gemini-A* scales to handle the overloaded request, but also has a latency spike at around 12:00 (14.13 seconds). In contrast, the tail latency of FaaSGraph remains steady throughout the whole process, and is only 2.14 seconds at peak.

Figure 13b showcases #ContainerSet results throughout the day. We observe that the #ContainerSet spikes sharply when a query surge occurs. This demonstrates that FaaSGraph can rapidly scale up and down to manage fluctuating workloads, even when enabling bounded resource scaling.

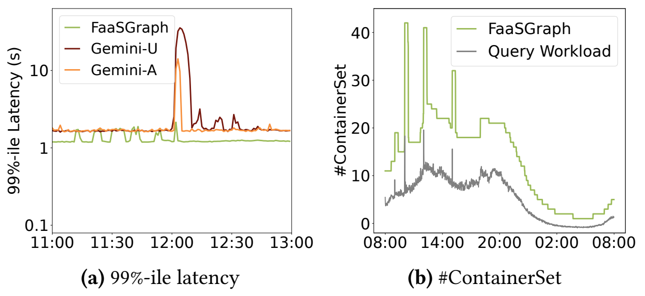


Figure 13. Tail latency and #ContainerSet during rush hours using query pattern in Figure 1.

Over the course of one day, FaaSGraph incurs a cost of only \$40.74, which represents savings of 85.7% and 77.3% compared to the costs of Gemini-U (\$283.96) and Gemini-A (\$179.24) respectively, given the unit price of a Gemini instance is \$0.408/h (for a1.4xlarge instance type).

7 Related Work

Graph Processing. In general, graph processing frameworks can be delineated into three categories [15] based on their parallization model: shared-memory, distributed and HPC systems. Firstly, there are frameworks designed for single-machine environments, leveraging shared memory to optimize performance, typically harnessed to manage small and medium scale datasets effectively [27, 49, 60]. Secondly, some are tailored for distributed setups, harnessing graph characteristics and formulated message-passing mechanisms to efficiently process large graph datasets across a server cluster [14, 17, 20, 37, 56, 61]. Lastly, frameworks [9, 40, 52] exist that express graph computing algorithms through a series of algebraic operations, employing the corresponding HPC computation models to effectively manage graph processes. While these frameworks accelerate the computation of a single query, they are primarily designed for high-performance local computing environments and do not take into account the unique characteristics of cloud infrastructures or the fluctuating workloads.

Serverless Data-processing. Leveraging serverless computing to process large-scale datasets is a burgeoning research area. For example, there has been several research focusing on providing ML training [13, 21, 24, 54] and ML inference [10–12, 22, 29, 59]. Lambada [41] and Starling [44] are two serverless data analytic frameworks, emphasizing redesign of relational operators (e.g., scan, shuffle, exchange) and their communication pattern in serverless. However, these works cannot directly facilitate FaaSGraph because graph processing consists of iteration-based computation, which makes it hard to be expressed by simple operators.

Graphless [53] is a serverless graph processing framework built on top of AWS. It pulls graph data from AWS S3, iteratively dispatches graph processing tasks to AWS Lambda, and enables the exchange of vertex states via Redis. Due to the lack of optimizations for graph processing on serverless (i.e., a simple direct integration), it is slower than Giraph [2], a legacy work. Nestorov et al. [43] propose a performance model to capture the end-to-end latency of data-centric workloads, and discuss key factors (e.g., concurrency and resource allocation) affecting performance without providing solutions. Different from these works, FaaSGraph co-designs serverless architectures and graph processing frameworks (e.g., communication and message passing) to accelerate graph computation.

Communication is critical in enhancing the performance of serverless applications. Studies such as Faasm [48], Cloudburst [50], and FaaSFlow [33] employ local caches or shared memory segments to sidestep network, utilizing fine-grained Put and Get interfaces grounded on KVS and leveraging global-tier storage for remote communication. However, to facilitate rapid state propagation, graph processing requires a more coarse-grained and efficient communication approach. Addressing this, FaaSGraph introduces a locality-aware communication strategy that harmonizes with graph processing frameworks through chunk-based interfaces. This promotes communication through shared memory within a host and facilitates network communication between different hosts.

8 Conclusion

We propose FaaSGraph, a serverless-native graph computing scheme that efficiently and cost-effectively processes large-scale graphs by bridging the gap between graph processing frameworks and FaaS. By co-designing serverless architectures and graph processing paradigms, FaaSGraph addresses challenges posed by the data and communication-intensive nature of graph processing and the lightweight tasks targeted by FaaS. Our evaluation demonstrates that FaaSGraph achieves significant improvements in latency, memory, and cost compared to traditional IaaS-based methods.

Acknowledgment

We sincerely thank our shepherd Dr. Rodrigo Fonseca and anonymous reviewers for their insightful comments and suggestions. This work is partially sponsored by the National Key Research and Development Program of China (2022YFB4501400) and National Natural Science Foundation of China (62232011, 62302302, 61832006). Yushi Liu and Shixuan Sun contributed equally to this work. Quan Chen is the corresponding author.

A Artifact Appendix

A.1 Abstract

FaaSGraph is a graph-as-a-service offering designed to be deployed by cloud providers. The artifact is open-sourced on Github (https://github.com/ziliuziliu/FaaSGraph). The repository encompasses the complete system implementation, experimental scripts, and a collection of anticipated results.

A.2 Artifact check-list (meta-information)

- **Algorithm:** A *Bounded Scaling* policy that limit the rate of container cold start to minimize memory footprint while processing diverse workloads.
- **Program:** FaaSGraph, Docker Runtime, Graph Applications
- **Data set:** Please download from https://snap.stanford.edu/.
- **Run-time environment:** Ubuntu 20.04 with Docker installed. Detailed software libraries are all scripted in the artifact.
- **Hardware:** The recommended hardware requirements is {Cores: 24, DRAM: 96GB, Disk: 100GB SSD}. More detail configurations are listed in the paper.
- **Execution:** We provide the corresponding scripts for part of the evaluation (latency breakdown of FaaSGraph). The scripts will send requests and collect the metrics.
- **Metrics:** We gather information on end-to-end latency, provide a detailed breakdown of latency, and track memory consumption.
- **Output:** csv files.
- **Experiments:** python scripts.
- **How much disk space required (approximately)?:** 100GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 1 hour
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** 10.5281/zenodo.10554182

A.3 Description

A.3.1 How to access. The source code of FaaSGraph is available on GitHub https://github.com/ziliuziliu/FaaSGraph.

A.3.2 Hardware dependencies. FaaSGraph is able to run in both shared memory (one-host) and distributed (multi-host) environments. For each host, the recommended hardware requirements is {Cores: 24, DRAM: 96GB, Disk: 100GB SSD}. For distributed environment, a NAS should be attached to each host to simulate remote storage access.

A.3.3 Software dependencies. The artifact requires VMs with Ubuntu 20.04 and Docker installed. We provide an installation script to prepare the software environment (including docker runtime, golang, python packages and other dependencies) for each host.

A.3.4 Data sets. We use the snap network datasets. Please download from https://snap.stanford.edu. Additional preprocessing scripts to transform the dataset is included in this artifact.

A.4 Installation

We recommend at least 1 node to install FaaSGraph and run subsequent experiments. The `README.md` in the artifact provides more details on installing software dependencies, installing graph datasets and conducting experiments.

Software Installation Run `util/prepare_machine.bash` on each node. This will install golang, docker, python package and build necessary images.

Dataset Installation Run `util/prepare_dataset.bash`. This will download graph datasets from snap, and build corresponding CSRs and offsets. This will take roughly an hour.

Configuration Update the `config/config.json` file on every node, setting the values of `CONTROLLER` and `WORKER` to their respective IP addresses. Please refer to the detailed instructions provided in the `README.md` file for a comprehensive illustration.

A.5 Experiment workflow

The experiment script is in `experiment/run.py`. This will take roughly an hour. All cases are runnable in a single node. It initiates the `faas_controller` server and the `local_manager` server, then invokes graph processing queries using a variety of graph datasets and graph applications.

A.6 Evaluation and expected results

The anticipated outcome is stored in the `experiment/expected_result` directory. Please note that IO performance are influenced by the choice of storage (local file system or NAS), leading to potential latency variations.

References

[1] Aliyun NAS. https://www.aliyun.com/product/nas.

[2] Apache Giraph. https://giraph.apache.org.

[3] AWS IAM. https://aws.amazon.com/iam.

[4] AWS Lambda Scaling Policy. https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html.

[5] Docker. https://docker.com.

[6] Firecracker. https://firecracker-microvm.github.io.

[7] Golang. https://go.dev.

[8] Kata containers. https://https://katacontainers.io.

[9] Muhammad Yousuf Ahmad, Omar Khattab, Arsal Malik, Ahmad Musleh, Mohammad Hammoud, Mücahid Kutlu, Mostafa Shehata, and Tamer Elsayed. LA3: A scalable link- and locality-aware linear algebra-based graph analytics system. *Proc. VLDB Endow.*, 11(8):920–933, 2018.

[10] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 69. IEEE/ACM, 2020.

[11] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proc. VLDB Endow.*, 15(10):2071–2084, 2022.

[12] Anirban Bhattacharjee, Yogesh D. Barve, Shweta Khare, Shunxing Bao, Aniruddha Gokhale, and Thomas Damiano. Stratum: A serverless framework for the lifecycle management of machine learning-based data analytics tasks. In *2019 USENIX Conference on Operational Machine Learning, OpML 2019, Santa Clara, CA, USA, May 20, 2019*, pages 59–61. USENIX Association, 2019.

[13] João Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy H. Katz. Cirrus: a serverless framework for end-to-end ML

workflows. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 13–24. ACM, 2019.

[14] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

[15] Miguel E. Coimbra, Alexandre P. Francisco, and Luís Veiga. An analysis of the graph processing landscape. *J. Big Data*, 8(1):55, 2021.

[16] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pages 918–934, 2019.

[17] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Kai Zeng, Kun Zhao, Jingren Zhou, Diwen Zhu, and Rong Zhu. Graphscope: A unified engine for big graph processing. *Proc. VLDB Endow.*, 14(12):2879–2892, 2021.

[18] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. Grape: Parallelizing sequential graph computations. *Proceedings of the VLDB Endowment*, 10(12):1889–1892, 2017.

[19] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 17–30, 2012.

[20] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 599–613. USENIX Association, 2014.

[21] Vipul Gupta, Swanand Kadhe, Thomas A. Courtade, Michael W. Mahoney, and Kannan Ramchandran. Oversketched newton: Fast convex optimization for serverless systems. In *2020 IEEE International Conference on Big Data (IEEE BigData 2020), Atlanta, GA, USA, December 10-13, 2020*, pages 288–297. IEEE, 2020.

[22] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018*, pages 257–262. IEEE Computer Society, 2018.

[23] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 152–166. ACM, 2021.

[24] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 857–871. ACM, 2021.

[25] Joe Wenjie Jiang, Tian Lan, Sangtae Ha, Minghua Chen, and Mung Chiang. Joint VM placement and routing for data center traffic engineering. In Albert G. Greenberg and Kazem Sohraby, editors, *Proceedings of the IEEE INFOCOM 2012, Orlando, FL, USA, March 25-30, 2012*, pages 2876–2880. IEEE, 2012.

[26] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[27] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 31–46. USENIX Association, 2012.

[28] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[29] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022.* USENIX Association, 2022.

[30] Yusen Li, Xueyan Tang, and Wentong Cai. On dynamic bin packing for resource allocation in the cloud. In Guy E. Blelloch and Peter Sanders, editors, *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*, pages 2–11. ACM, 2014.

[31] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. Rund: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 53–68. USENIX Association, 2022.

[32] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Comput. Surv.*, 54(10s):220:1–220:34, 2022.

[33] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: enable efficient workflow execution for function-as-a-service. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 782–796. ACM, 2022.

[34] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[35] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 285–301. USENIX Association, 2021.

[36] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless dags. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 303–320. USENIX Association, 2022.

[37] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010.

[38] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 631–643. USENIX Association, 2017.

[39] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what cost? In George Candea, editor, *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015.* USENIX Association, 2015.

[40] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. Graphite: A numa-aware hpc system for graph analytics based on a new mpi * x parallelism model. 13(6):783–797, mar 2020.

[41] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA],*

June 14-19, 2020, pages 115–130. ACM, 2020.

[42] Aniket Murhekar, David Arbour, Tung Mai, and Anup B. Rao. Brief announcement: Dynamic vector bin packing for online resource allocation in the cloud. In Kunal Agrawal and Julian Shun, editors, *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2023, Orlando, FL, USA, June 17-19, 2023*, pages 307–310. ACM, 2023.

[43] Anna Maria Nestorov, Jordà Polo, Claudia Misale, David Carrera, and Alaa Youssef. Performance evaluation of data-centric workloads in serverless environments. In Claudio Agostino Ardagna, Carl K. Chang, Ernesto Daminai, Rajiv Ranjan, Zhongjie Wang, Robert Ward, Jia Zhang, and Wensheng Zhang, editors, *14th IEEE International Conference on Cloud Computing, CLOUD 2021, Chicago, IL, USA, September 5-10, 2021*, pages 491–496. IEEE, 2021.

[44] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 131–141. ACM, 2020.

[45] Jiuchen Shi, Kaihua Fu, Quan Chen, Changpeng Yang, Pengfei Huang, Mosong Zhou, Jieru Zhao, Chen Chen, and Minyi Guo. Characterizing and orchestrating VM reservation in geo-distributed clouds to improve the resource efficiency. In Ada Gavrilovska, Deniz Altinbüken, and Carsten Binnig, editors, *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, pages 94–109. ACM, 2022.

[46] Jiuchen Shi, Jiawen Wang, Kaihua Fu, Quan Chen, Deze Zeng, and Minyi Guo. Qos-awareness of microservices with excessive loads via inter-datacenter scheduling. In *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022*, pages 324–334. IEEE, 2022.

[47] Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, and Minyi Guo. Nodens: Enabling resource efficient and fast qos recovery of dynamic microservice applications in datacenters. In Julia Lawall and Dan Williams, editors, *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, pages 403–417. USENIX Association, 2023.

[48] Simon Shillaker and Peter R. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 419–433. USENIX Association, 2020.

[49] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[50] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(11):2438–2452, 2020.

[51] Alexander L. Stolyar. An infinite server system with general packing constraints. *Oper. Res.*, 61(5):1200–1217, 2013.

[52] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, 2015.

[53] Lucian Toader, Alexandru Uta, Ahmed Musaafir, and Alexandru Iosup. Graphless: Toward serverless graph processing. In *18th International Symposium on Parallel and Distributed Computing, ISPDC 2019, Amsterdam, The Netherlands, June 3-7, 2019*, pages 66–73. IEEE, 2019.

[54] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*, pages 1288–1296. IEEE, 2019.

[55] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. Peeking behind the curtains of serverless platforms. In Haryadi S. Gunawi and Benjamin C. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 133–146. USENIX Association, 2018.

[56] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 194–204. ACM, 2015.

[57] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.*, 7(14):1981–1992, 2014.

[58] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*, pages 745–754. IEEE Computer Society, 2012.

[59] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 768–781. ACM, 2022.

[60] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. Graphit: a high-performance graph DSL. *Proc. ACM Program. Lang.*, 2(OOPSLA):121:1–121:30, 2018.

[61] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, 2016.