FaaSBoard: Efficient Graph Processing with a Disaggregated Architecture on Serverless Services

Abstract

Graph processing workloads are increasingly being migrated to the cloud. With the growing adoption of serverless computing, graph processing gains advantages such as cost-effectiveness and resource elasticity. However, existing graph processing systems with monolithic function architecture struggle to detect intra-job resource elasticity and suffer from significant communication overhead.

In this paper, we present FaaSBoard, a graph processing system with a disaggregated serverless architecture powered entirely by serverless cloud services. FaaSBoard features a multi-tier data communication mechanism and an autonomous-elastic computing mechanism. Specifically, these two mechanisms are realized through image-based graph loading for faster starts, proxy-based collective communication leveraging high-bandwidth shared memory, 2D balanced graph partitioning for improved load balance, and a proactive terminate-and-respawn mechanism enabling fine-grained elasticity. Together, these four techniques collectively enhance both resource and overall execution efficiency. Experimental results demonstrate that FaaSBoard delivers up to 3.8× higher compute performance and reduces monetary cost by up to 63.7% compared to FaaSGraph, the current state-of-the-art serverless-based graph processing system.

1 Introduction

Graphs are powerful tools for modeling real-world entities and their relationships. Through everyday use of applications such as social networks, e-commerce platforms, and online banking, users generate numerous footprints that collectively form large-scale graphs, which companies can analyze to extract valuable business insights. As a result, graph processing has attracted significant attention, leading to the development of numerous frameworks [2, 3, 11, 16, 18, 35, 40, 52, 53, 58, 59] designed to accelerate computation by efficiently utilizing resources in and beyond a single server.

As graph datasets continue to grow in size, graph processing applications have increasingly shifted to the cloud [12, 38, 39] to conveniently leverage large-scale computing resources. Since most graph processing frameworks are originally designed for local clusters, a common approach is to replicate this setup in the cloud by provisioning cluster with monolithic servers. As illustrated in Figure 1a, the Graph Service Developer (i.e. the User of the Cloud) allocates a fixed number of VM instances, each with ample CPU and memory resources to host the graph processing framework.

However, this serverful model suffers from poor resource elasticity, making it inefficient for workloads that are sporadic or infrequent—such as computations that only need to run occasionally over long intervals [26]. To address these limitations, the rise of serverless offers a compelling alternative. It allows users to exploit the burstability of serverless functions for large-scale data processing [9, 17, 20, 29, 46, 48] without the burden of resource management. Additionally, the pay-as-you-go billing model reduces costs during idle periods. Motivated by these advantages, graph processing systems are embracing serverless, adopting the monolithic function architecture like Figure 1b. For instance, FaaSGraph [35] integrates a built-in FaaS runtime that orchestrates multiple serverless functions to perform graph processing.

Nevertheless, our investigation shows that the current monolithic function architecture comes with trade-offs. First, long-running graph processing often exposes fine-grained resource elasticity due to load imbalances within iterations, where one compute instance must wait for others. However, it is challenging for the architecture to exploit this intra-job elasticity due to the complex iteration-based coordination between functions, leading to unnecessary monetary costs due to resource idle (36.4% in our profiling of FaaSGraph). Second, the communication of large data volumes becomes a critical bottleneck. The existing architecture is limited to either cloud storage or direct communication, suffering from high communication overhead due to low bandwidth. These methods incur 1.43×-3.02× longer graph processing time compared to the optimal performance.

We identify the root cause as the architecture preserving the monolithic nature of resources, where functions are used like monolithic servers, but with limited resources and faster initialization. *We propose a disaggregated serverless architecture to address the deficiencies.* Figure 1c shows the architecture along with two critical mechanisms. First, a multi-tier data communication mechanism is introduced on top of a shared memory model enabled by the architecture. The shared memory provides a high-bandwidth communication channel to facilitate data transmission. Additionally, efficient data loading is supported by a tiered cache layer, where data is integrated closely with compute resource. Second, an autonomous-elastic computing mechanism is introduced, allowing resources to be terminated and spawned more frequently and proactively to exploit intra-job resource elasticity, while leveraging shared-memory to track query execution without losing progress.

Building on this design, we introduce FaaSBoard, a graph processing system that fully adopts a disaggregated serverless architecture. The user begins by partitioning the graph dataset, bundling with the execution binary into container images, and uploading to an image repository. Then, user can invoke queries seamlessly in the cloud without the need to provision or manage any servers. To this end, *FaaSBoard runs entirely in a serverless cloud ecosystem, eliminating the need for dedicated servers or long-running instances.* This design fully leverages the elasticity of serverless services, enhancing the system's generality and cost-efficiency.

In FaaSBoard, we propose four techniques across the data and compute layers to collaboratively enhance end-to-end system efficiency. Specifically, to implement the multi-tier data communication, it employs *Image-based Graph Loading*, which utilizes container images as a high-tier cache layer to enable efficient graph loading. Furthermore, *Proxy-based Collective Communication* leverages serverless containers as a shared memory layer to support high-bandwidth broadcast and reduce operations. On the compute side, FaaSBoard introduces the autonomous-elastic computing through a *Proactive Terminate-and-Respawn* mechanism, enabling

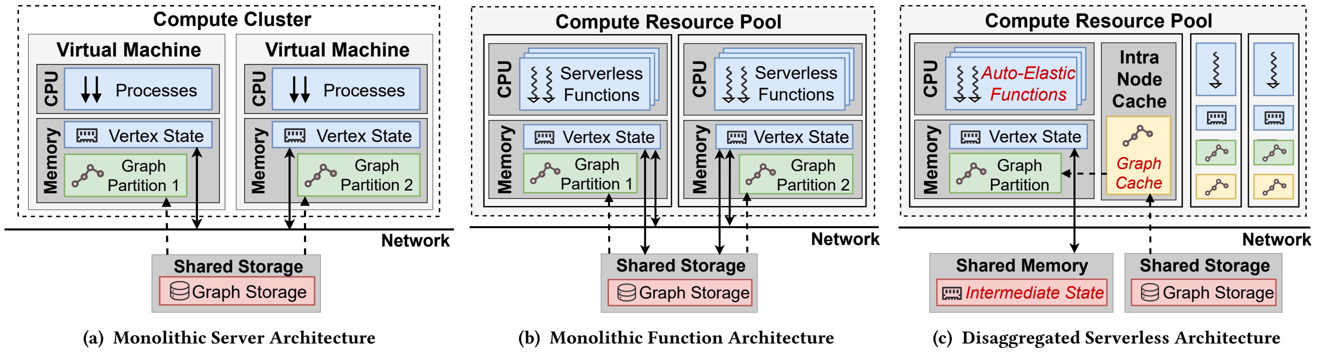


Figure 1: Comparison of graph processing framework architectures in cloud environments. The dashed arrow represents graph loading, while the solid arrow indicates vertex state exchange. (a) Before graph processing begins, subgraphs are loaded into memory from shared storage. Vertex states are computed and exchanged over the network using broadcast and reduce. (b) A fleet of serverless functions replaces traditional servers for computation, with state exchanged via storage services (e.g., S3) or direct communication channels [13, 51]. (c) The graph processing system autonomously manages resource termination and respawning. Subgraphs are loaded via a cache layer, while vertex states are exchanged through a shared memory layer.

autonomous resource management on serverless service. Furthermore, a *2D Balanced Graph Partitioning* method is introduced to improve graph processing performance under the resource constraints of serverless environments.

Our extensive experiments demonstrate that FaaSBoard achieves up to 3.8× higher graph processing performance and reduces monetary cost by up to 63.7%, compared to FaaSGraph [35], a state-of-the-art graph processing system adopting the monolithic function architecture. In summary, this paper makes three key contributions:

- We analyze the limitations of monolithic function architectures in graph processing, and motivate the design of a disaggregated serverless architecture.
- We introduce FaaSBoard, a graph processing system built on serverless cloud services to validate the effectiveness of the disaggregated architectural design.
- We conduct a comprehensive evaluation of FaaSBoard to demonstrate its superior performance and cost.

The remainder of the paper is organized as follows. Section 2 reviews background and related work. Section 3 motivates a disaggregated architectural design. Section 4 introduces the architecture and execution flow of the FaaSBoard system. Sections 5, 6 and 7 detail our implementation of such disaggregated architecture on serverless services. Section 8 presents a comprehensive evaluation to validate our design.

2 Background and Related Work

Serverless. Serverless computing has emerged as a compelling cloud service model, offering auto-scaling capabilities, a pay-per-use billing system, and a management-free experience for users. In this model, code packages are encapsulated as functions, and deployed by the cloud provider which handles request scheduling and resource scaling.

Many studies have adopted serverless computing as an approach to providing burstable computing resources, thereby facilitating large-scale data processing [9, 17, 29, 46, 48]. Due to the stateless nature of serverless functions, a common strategy for communication and data loading is to utilize shared storage services (e.g., S3, ElastiCache) [6, 33, 41–43]. However, this approach often results in performance degradation due to the high overhead and limited bandwidth of storage services. In FaaSBoard, we establish channels that are more tightly integrated with the serverless function service to optimize data transmission.

Some studies have proposed direct connection (e.g. Boxer [51], FMI [13]) to enable more efficient communication between functions. However, when only serverless functions are involved, communication performance remains limited due to the low bandwidth of individual functions. Other studies including Pocket [24] and Jiffy [23] utilize the EC2 cluster as a far memory pool to build a general-purpose storage system. In contrast, FaaSBoard is specifically designed to optimize collective operations common in graph processing by offloading reduce computations to a shared memory pool and leveraging high bandwidth to improve performance.

There are multiple ways to achieve fine-grained resource elasticity in serverless computing. For instance, small applications can be decomposed into microservices [19, 22, 30, 31] or workflows [32, 36, 37]. Data-intensive applications such as linear algebra [46] and SQL queries [21, 41, 43, 57] can be decoupled into small operators. However, our work focuses on supporting general graph processing algorithms, which are inherently difficult to decompose. Instead, we propose the autonomous elastic computing to leverage fine-grained elasticity through a resource-aware mechanism.

Graph Processing. A graph $G = (V, E)$ represents a set of vertices V and the edges $E \subseteq V \times V$ between them. Graph processing frameworks are built to efficiently analyze large-scale graph datasets through concurrent computing. Following previous work [35, 59], graph processing involves iteratively applying user-defined algorithms to subgraphs, generating a state for each vertex as the output. For example, in the single-source shortest path algorithm, the graph processing framework produces the minimum distance for each vertex.

Graph partitioning is crucial for minimizing communication and ensuring load balance. A common approach is chunk-based partitioning [59], favored for its low overhead and effective exploitation of vertex locality. However, the 1D partitioning used in Gemini [59] and FaaSGraph [35] significantly increases communication with more partitions while some 2D methods [2, 3, 40] often ignore graph sparsity. FaaSBoard uses a novel 2D balanced partitioning, well-suited for serverless environments where limited function resources require a high number of functions to accommodate the graph, and make load balancing essential to avoid OOM errors.

Graph processing has increasingly embraced serverless computing. FaaSGraph [35] leverages serverless elasticity to handle real-world fluctuating workload. Unlike FaaSGraph, which modifies the serverless infrastructure, FaaSBoard focuses on efficiently utilizing existing services, enhancing its generality. Graphless [49] adopts a similar approach to ours, but its performance lags behind many legacy systems.

3 Motivation

In this section, we motivate the need for developing a disaggregated architecture in serverless computing to streamline graph processing with enhanced efficiency and elasticity.

3.1 Linear Algebra based Execution Model

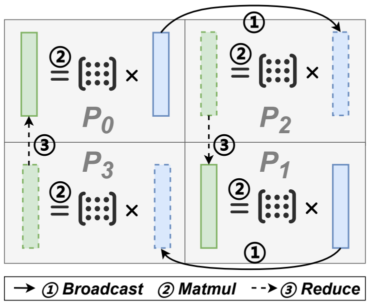
Following GraphBLAS [14], graph computations can be equivalently transformed into linear algebraic operations. More formally, iteration t of graph processing is represented as a sparse matrix-vector multiplication (SpMV) operation: $x^{(t)} = A^T x^{(t-1)}$, where A denotes the adjacency matrix of G, and $x^{(t)}$ represents the vector of vertex states at iteration t. The graph algorithm is determined by the multiplication $*$ and the addition $+$ in matrix multiplication, which describe how to derive and aggregate vertex states. The programming model bears similarity to that of Graphite [40], allowing users to define functions that specify how to perform computations for neighboring vertex states.

Figure 2a illustrates a typical distributed execution pattern, where the adjacency matrix is divided into four submatrices, each assigned to a separate process. The blue vector from P_0, P_1, representing the partitioned vertex states from the previous iteration, is broadcast within the row group to processes P_2 and P_3. Subsequently, each submatrix performs local computations to generate the green vector, which is then reduced within the column group to produce the result for the current iteration.

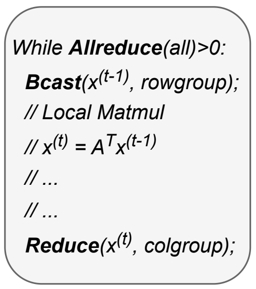
3.2 Deficiency of Monolithic Function Architecture

Figure 2b shows a code snippet that illustrates the execution model described in Section 3.1. We first implement this model on top of the existing monolithic function architecture and investigate its behavior. All experiments are conducted using BFS on the Twitter [27] graph with around 1.8 billion edges. Constrained by limited memory of functions, the execution requires six functions.

Deficiency in Resource Elasticity. Due to the complex structure of graph data and irregular computation patterns, achieving perfect load balancing across workers is difficult. This leads to significant variation in execution times among functions in each iteration. Figure 3 shows the normalized execution time of six functions over three iterations, revealing substantial performance disparity. As BFS progresses and explores new frontiers, workloads shift dynamically—for instance, Function F2 is idle in iteration 1 but becomes heavily loaded in iteration 2. Such fluctuations complicate function behavior over time. In a complete execution, 36.4% of resources are wasted due to intra-iteration idle time, directly translating into equivalent monetary cost overhead.



(a) 2D Execution Pattern



```
While Allreduce(all)>0:
  Bcast(x^(t-1), rowgroup);
  // Local Matmul
  // x^(t) = A^T x^(t-1)
  // ...
  // ...
  Reduce(x^(t), colgroup);
```

(b) Code Snippet

Figure 2: Distributed Execution Model of FaaSBoard

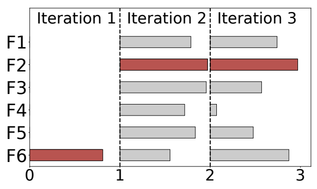


Figure 3: Normalized execution time in three consecutive iterations of tw-BFS.

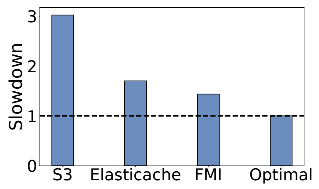


Figure 4: Performance when applying various communication methods.

However, serverless platforms cannot directly improve resource efficiency by utilizing the idle resources in each iteration because they lack visibility into application-level execution. A straightforward approach, terminating a worker after execution, falls short for graph processing, which requires iteration-based coordination among multiple workers. Additionally, workloads fluctuate across iterations, and functions may be invoked dynamically and unpredictably. Moreover, implementing such an approach often requires injecting custom logic into graph applications, leading to significant engineering overhead. Consequently, enabling fine-grained intra-job resource elasticity remains a major challenge for improving resource efficiency.

Observation 1. *In graph processing, worker execution times vary significantly across iterations, causing substantial resource underutilization. However, optimizing resource efficiency is challenging due to the need for coordinated, iterative execution across workers.*

Deficiency in Data Communication. Graph processing depends on collective operations to exchange vertex states, often involving large volumes of data. However, serverless functions have limited communication capabilities. We evaluate the impact of representative serverless communication mechanisms on BFS performance, including shared storage (e.g., S3, ElastiCache) and direct communication (e.g., FMI [13]). Figure 4 shows the slowdown compared to a baseline cluster with high-speed networking. The results reveal significant performance gaps, highlighting the inefficiency of bulk data transfers between serverless functions.[1]

Most existing systems use shared storage for communication [20, 33, 41–43, 46, 47], but performance suffers from low bandwidth and high latency [6, 13, 51]—a consequence of the decoupling between storage and compute. Additionally, since storage lacks compute capabilities, extra functions must be invoked to perform reduce operations [20], introducing further overhead. To address this, direct communication methods involving only serverless functions have been proposed [13, 51]. However, we find them suboptimal, as the limited bandwidth of serverless functions (75–90 MiB/s [6, 41]) necessitates binomial tree-style broadcast and reduce patterns [13], which increase latency for leaf functions and add coordination overhead. Reduce operations also consume precious compute resources that could otherwise be used for graph processing. Moreover, we were unable to reproduce these methods on real-world platforms like AWS Lambda[2], suggesting they may be constrained by security, billing concerns, or susceptibility to infrastructure updates.

Observation 2. *Graph processing requires frequent bulk data exchange through collective operations, which serverless platforms handle poorly, resulting in performance degradation.*

3.3 Implication for System Design

Based on these observations, directly applying traditional cluster-based graph processing architectures to serverless platforms leads to limited resource elasticity and inefficient data communication. To address these challenges, we adopt a disaggregated architecture tailored for serverless environments. Specifically, we focus on two key mechanisms.

Autonomous-Elastic Computing. Current serverless graph processing systems scale only at the query level, limiting resource efficiency. Specifically, the complex coordination and the dynamic workload of functions hinder the system from exploiting per-iteration elasticity. To enable such fine-grained elasticity, we leverage a disaggregated architecture that provides shared memory for maintaining execution progress of the whole query independently of function lifecycle, allowing functions to terminate and respawn without losing progress. Building on this, we design an autonomously elastic computing mechanism that improves resource efficiency while minimizing decision-making overhead.

Multi-tier Data Communication. Communication in the monolithic function architecture is restricted to direct communication or shared storage, leading to overhead due to the absence of an intermediate approach. In contrast, our disaggregated system architecture utilizes a multi-tiered data hierarchy. Specifically, tiered caches can accelerate graph loading, while the shared memory pool enables efficient collective communication with high-bandwidth data transfer and built-in compute capabilities for reduce operations.

4 FaaSBoard Overview

We present FaaSBoard, a graph processing system that fully adopts the disaggregated serverless architecture in Section 3.3. Our core principle is the *exclusive reliance on serverless cloud services*, which guarantees scalability for every component, and makes our design orthogonal to the underlying cloud implementation, thereby enhancing the applicability.

Figure 5 illustrates the architecture and execution flow of FaaSBoard.

Initialization. During the initialization phase, the user uploads a graph dataset along with the corresponding algorithm code. The graph is then partitioned into subgraphs using the *2D Balanced Graph Partitioner* (Section 6.1). Simultaneously, the code is compiled together with the helper graph runtime into an executable binary. The resulting subgraphs and the binary are packaged into images and submitted to a cloud image repository.

Execution. In the execution phase, after the user submits a query to the FaaS platform, a fleet of serverless functions is triggered, with each function processing a distinct subgraph. In cases of a hot start, graph data within each function is reused; otherwise, the system employs *Image-based Graph Loading* (Section 5.1) to load the graph partitions, using container image as a high-tier cache layer. During execution, the FaaS runtime implement the *Proactive Terminate-and-Respawn* mechanism (Section 6.2) to conserve resources during idle periods. Additionally, they utilize *Proxy-based Collective Communication* (Section 5.2) to perform collective operations such as broadcast and reduce, with proxies built atop the serverless container service acting as a shared memory layer.

Generality. In FaaSBoard we focus on leveraging universal characteristics of serverless computing rather than relying on unique features offered by specific cloud providers. For instance, in Section 5.1, our approach is built on the insight that loading data from image is faster than remote storage service. Similarly, in Section 6.2, our mechanism is grounded in the pay-per-use billing model and the keep-alive policies commonly adopted by cloud providers. We believe these characteristics represent fundamental aspects of serverless computing and will remain relevant in the future.

We provide a detailed explanation of the mechanisms in the following sections. Although we build our system on AWS, we note that similar services are widely available across other major cloud providers.

5 Graph Loading and Communication

In this section, we present the multi-tiered data communication supported by cloud services, incorporating efficient mechanisms for graph loading and vertex state exchange.

5.1 Image-based Graph Loading

Graph loading typically occurs during a cold start, when no computing resources are available and containers must be initialized from scratch. Users often rely on external storage services such as object storage (e.g., S3) or in-memory caching services (e.g., ElastiCache) for data storage. However, these approaches frequently suffer from

[1]Due to their poor performance, shared storage methods are excluded from later experiments in Section 8.3.2.
[2]Our experiments simulate FMI using a proxy; see Section 8.3.2 for details.

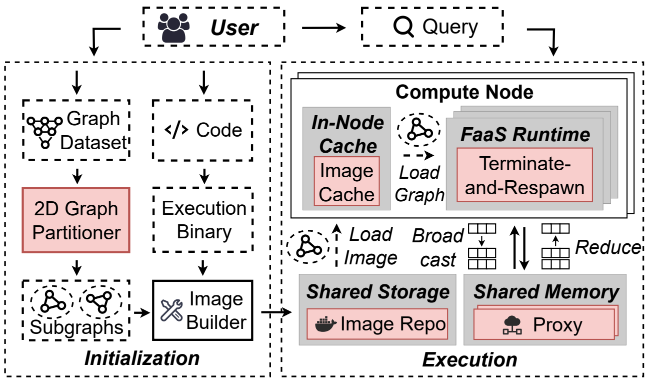


Figure 5: FaaSBoard Architecture and Execution Flow. The gray components indicate adherence to the disaggregated serverless architecture in Figure 1c, while the red components highlight our proposed mechanisms.

performance inefficiencies mainly due to high network overhead and bandwidth limitations. As a result, we consider them to be the low-tier storage due to their complete separation from computing resources (i.e., serverless function service).

We propose that *loading data from container images is faster than from remote storage*, and argue that container images can serve as an ideal high-tier cache layer. In serverless function service, functions are packaged and executed within containers launched from these images. The images encapsulate both the function code and all required software dependencies, which are loaded during a cold start. Cloud providers have been actively optimizing this loading process; for instance, AWS Lambda [7] employs multi-level caching to accelerate image loading, presenting a promising opportunity for high bandwidth data loading.

Thus, we propose our *Image-based Graph Loading* mechanism to benefit from such infrastructure. Specifically, after the graph is partitioned, each subgraph is bundled together with the execution binary into a container image. These images are then uploaded to the image repository service. During a cold start, rather than retrieving the subgraph from remote storage, the system directly loads the data from the local file stored within the container image.

Our mechanism substantially reduces network overhead, while mitigating bandwidth bottlenecks by leveraging the local cache bandwidth across multiple compute nodes when functions are distributed. However, a challenge in fully utilizing this mechanism is the opacity of cache behavior to users, which makes graph loading performance unpredictable. To maintain the "hotness" of graph data, we introduce a *Data Ping* mechanism. This involves invoking functions at regular intervals (1 hour in FaaSBoard) to load graph data, ensuring persistent availability in the local cache, while incurring additional monetary costs due to repeated function invocations. Section 8.3.1 evaluates the loading performance and cost overhead in comparison to the shared storage approach.

5.2 Proxy-based Collective Communication

Figure 2b shows the collective operations within FaaSBoard, where broadcast and reduce operations account for the majority of communication volume. In graph processing terms, the broadcast operation sends activated vertices and their associated states to other functions. Conversely, the reduce operation aggregates updated vertex states. Additionally, a vote operation (i.e. Allreduce) is employed to determine whether to compute in the next round. Next, we describe our *Proxy-based Collective Communication* mechanism, which focuses on efficient execution of collectives.

5.2.1 *Proxy.*

The Proxy operates on top of the serverless container service (e.g., AWS Fargate). It acts as a shared memory layer, enabling fast data transfer between functions and overcoming the limitations of direct connection. Each function establishes a connection to a Proxy instance through a network socket at the beginning of a query, and disconnects once it is completed. Proxies can be multiplexed, allowing functions for different queries to connect to a single Proxy instance, thereby amortizing the monetary cost.

5.2.2 *Collective Operations with Proxy.*

Next, we describe the implementation of collective operations using the Proxy. We employ *Vector* as the message abstraction.

Broadcast. During broadcast, the sender function first transmits the vector to the Proxy, which then distributes it to the receivers. Serverless containers offer substantially higher network bandwidth than functions. Specifically, using iPerf3 [1], we measured that each AWS Fargate container (16vCPU, 64GB memory) has a bandwidth of 636 MiB/s, which is 7.0X-8.5X greater than that of AWS Lambda. Thus, compared with direct connection approaches, vectors can be broadcasted in parallel to other functions, eliminating the need for a binomial tree routing and reducing network hops.

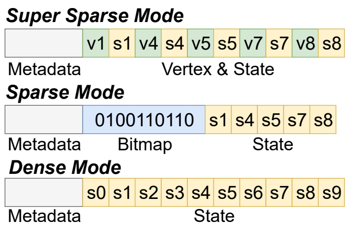
Reduce. Result vectors from the senders are aggregated at the Proxy. Due to load imbalance, functions may generate their results at different times. Inspired by In-network Aggregation [28, 34, 44], an *In-memory Buffer* is introduced to support on-the-fly reduction, allowing vectors to be automatically aggregated upon arrival. Once the Proxy has collected all the results, it forwards the aggregated data to the receiver. Compared to direct connection, the Proxy offloads reduce computation from serverless functions, enabling concurrent execution of reduce and graph processing.

Vote. The Vote operation combines reduce and broadcast. The number of activated vertices for the next round is first reduced at the Proxy, then a decision is broadcast to all functions, indicating whether to continue computation.

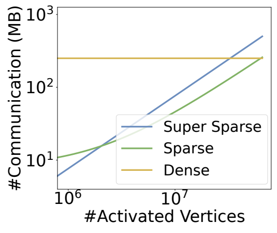
5.2.3 *Triple-mode Vector Format.*

Figure 6a illustrates the different formats for Vector. For each format, metadata such as the message source and the number of companion vectors to wait for is prefixed. When the activated vertices are highly sparse, the vertices and their corresponding states are directly appended. If the activated vertices are moderately sparse, a bitmap is appended to indicate the activated vertices, followed by their states at the end. In scenarios where the activated vertices are dense, the states of all vertices are appended sequentially, regardless of their activation status.

Figure 6b illustrates the communication volume associated with various vector formats under different amount of activated vertices, using the Friendster graph [54] as an example. It demonstrates that



(a) Different Formats



(b) #Communication

Figure 6: Triple-mode Vector Format

adaptively choosing between formats can reduce communication overhead. For instance, selecting the Super Sparse Mode when the number of activated vertices is low proves beneficial. On the other hand, the Dense Mode is particularly advantageous as it is the only format that stores states in a contiguous order, enabling AVX Instruction Set-based optimizations at the Proxy server during reduce computations to improve performance. Additionally, Dense Mode is the only approach that enables zero-copy transmission, as FaaSBoard directly sends contiguous vertex states under this mode. In contrast, the other two formats require message reconstruction, introducing additional overhead due to the copying of vertex states.

We set the default thresholds for Super Sparse-Sparse and Sparse-Dense to 10^5 and 10^6, respectively. However, these thresholds also impact communication traffic under concurrent workloads, which we leave for future exploration. Section 8.3.2 highlights our performance advantages.

6 Graph Processing and Partitioning

In this section, we present our compute-oriented mechanisms, including a novel partitioning method that enhances graph processing performance under the resource constraints of serverless functions, and a Terminate-and-Respawn mechanism that enables autonomous-elastic computing.

6.1 2D Balanced Graph Partitioning

Impact of Graph Partitioning. Many state-of-the-art graph processing systems [2, 3, 35, 40, 59] use chunk-based partitioning due to the locality in real-world graphs [59]. Figures 7a, 7b and 7c illustrate representative partition methods adopted by prior works, using the adjacency matrix with four partitions as an example.

Graph partitioning directly impacts the total communication volume during graph processing and the load balancing across subgraphs. For broadcast and reduce operations, 1D partitioning exhibits higher communication complexity compared to 2D methods [2]. However, previous 2D approaches will encounter load imbalance caused by uniformly spaced cutting, which fails to account for the sparsity of the graph.

Our Principle. Communication of large data volumes is a major overhead in serverless-based analytic systems [13, 32, 35, 56]. Thus, our system prioritizes communication optimization, adopting 2D methods for their low communication complexity. Considering serverless functions have significantly smaller memory than servers, our system then ensures load balancing within the 2D framework to prevent from Out-of-Memory of one function and optimize performance.

Problem Formulation. We assume the graph G with n vertices and m edges will be partitioned into p subgraphs. We assume that p is a perfect square number, and relax this assumption in the subsequent discussion. We define the cut positions as $c_{0..\sqrt{p}}$ (where $c_0 = 0$, $c_{\sqrt{p}} = n$), making both vertical and horizontal cuts at the same positions to ensure that the diagonal submatrices remain square. Our objective is to maximize load balancing, which is equivalent to minimizing the workload of the most heavily loaded submatrix. Equation 1 defines the workload of a submatrix, where A is the adjacency matrix, and nnz counts non-zero elements of the submatrix. The first and second elements represent computation and communication load, respectively. A coefficient β is employed to modulate the significance between computation and communication, where we assign $\beta = 10$ in FaaSBoard. Equation 2 formulates the optimization target.

$$load(A_{c_i:c_{i+1},c_j:c_{j+1}}) = nnz + \beta(c_{i+1} - c_i + c_{j+1} - c_j) \tag{1}$$

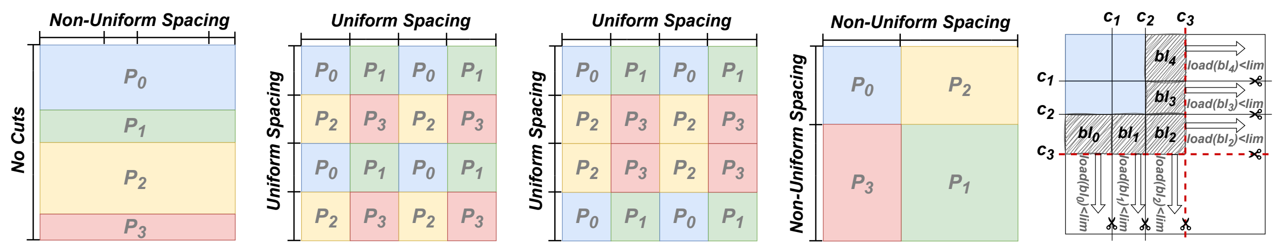
$$\min_{c} \max_{0<=i,j<\sqrt{p}} load(A_{c_i:c_{i+1},c_j:c_{j+1}}) \tag{2}$$

Step 1. Balanced Partitioning. Here, we propose a binary-search-based algorithm to generate the *optimal* partitioning under Equation 2. Specifically, we iteratively search for the minimum workload threshold, denoted as *limit*, such that there exists a partitioning of the graph where the workload in each subgraph does not exceed this threshold. If this condition is satisfied, we decrease the *limit* and repeat the check; otherwise, we increase the *limit*. This process continues until the optimal *limit* is determined, which at the same time generates the partition result (i.e. the cut positions c).

A key challenge is determining whether the *limit* can be satisfied in polynomial time. To address this, we sequentially determine each cut position. Figure 7e illustrates an example. Suppose we have already determined positions c_1 and c_2. We initially set $c_3 = c_2$ and then incrementally move c_3 forward until one of the newly created submatrices $bl_{0..4}$ reaches the *limit* at $c_3 = c'$. At this point, we fix c_3 to be $c' - 1$ and proceed to search for the next cut positions. If more than $\sqrt{p} - 1$ cuts are required to ensure that the workload of each subgraph remains below the *limit*, this indicates that the *limit* cannot be satisfied. Otherwise, the *limit* is feasible.

Algorithm 1 provides the pseudocode for our partitioning algorithm. The inputs include the adjacency matrix A, the required number of cuts $cut = \sqrt{p}$, and the error rate ϵ. Using a binary search approach, we iteratively refine the decision boundary $[left, right]$ and evaluate the workload *limit* until either the optimal solution is found or the error rate falls below ϵ. For each *limit*, we generate cut positions *cuts* and check if the number of cuts exceeds *cut*. For each vertex i, we process its incoming and outgoing edges, assigning them to the corresponding block *cur_block*. If the total edges in *cur_block* surpass *limit*, a new cut is fixed, *cur_block* is cleared, and the process continues until the final *cuts* are determined.

The correctness is guaranteed by *Optimality*, ensuring it finds a valid partition result if one exists, and *Monotonicity*, which validates the effectiveness of the binary search.



(a) 1D [35, 59] (b) 2D-Cyclic [3] (c) 2D-Staggered [2, 40] (d) 2D-Balanced (Ours) (e) *limit* Check

Figure 7: Different chunk-based partition methods when $p = 4$, and the *limit* check example.

Algorithm 1: Balanced Partitioning Algorithm

```
Input: A adjacent matrix in CSR format, cut number of cuts, ε error rate
Output: cuts generated cuts
left,right,cuts := 1,A.edges,[];
while (right-left)/right > ε do
    limit,valid,cur_cuts := (left+right)/2,true,[0];
    block,cur_block := [],[];
    for i=0..N-1 do
        cur_block.clear();
        for src in i.inedges where src<=i do
            id := cur_cuts.size*2-2-between_id(cur_cuts,src);
            cur_block[id]++;
        end
        for dest in i.outedges where dest<=i do
            id := between_id(cur_cuts,dest);
            cur_block[id]++;
        end
        for j=0..cur_cuts.size*2-2 do
            if block[j]+cur_block[j] > limit then cur_cuts.push(i);
              block := cur_block;
            else block += cur_block;
        end
        if cur_cuts.size > cut then valid := false; break;
    end
    if valid then cuts,left := cur_cuts,limit+1;
    else right := limit-1;
end
```

Theorem 1. *Optimality: the algorithm produces minimal cuts for a given workload limit.*

Proof. Suppose there exists another partition plan $\hat{c}$ such that $|\hat{c}| < |c|$. Let k be the first position where $\hat{c_k} \neq c_k$. Firstly, $\hat{c_k} > c_k$ is impossible because the algorithm places c_k at the rightmost valid position without exceeding the *limit*. Secondly, if $\hat{c_k} < c_k$, moving the cut c_k leftward would leave a larger remaining subgraph, requiring at least $|c| - k$ cuts after c_k to satisfy the *limit*. This implies $|\hat{c}| >= |c|$, contradicting $|\hat{c}| < |c|$. Thus, we prove that c is optimal. ■

Theorem 2. *Monotonicity: a valid partition plan for limit implies one for limit′ > limit; no valid plan for limit implies none for limit′ < limit.*

Proof. We omit the proof as it is obvious. ■

The time complexity of this algorithm is $O(mlog(m+n))$, which accounts for the binary search and the *limit* check. Although this is higher than the complexities of previous methods ($O(n)$ for 1D-based and $O(1)$ for 2D-based), we consider it feasible since the partitioning algorithm only needs to run once. Furthermore, we introduce two optimizations to improve its performance. First, we implement an early-stop mechanism in the binary search instead of pursuing the optimal result. In our system, we tolerate a 1% error for the *limit*, as this has minimal impact on graph processing performance while significantly reducing the number of binary search iterations. Second, we parallelize the binary search process. Specifically, assuming we have t threads, we divide the current decision boundary for *limit* into $t + 1$ equally spaced intervals and evaluate the t possible *limit* values in parallel. This approach allows us to quickly narrow down the decision boundary and find the result efficiently.

Step 2. Binpacking. In FaaSBoard, Users should be able to adjust the number of subgraphs p with flexibility, which influences the number of functions and CPU resources. To enable tuning of the latency-cost trade-off, we propose a binpacking algorithm to support flexible adjustments of non-square p. First, the graph is partitioned into $\lceil\sqrt{p}\rceil^2$ balanced partitions. Then, partitions are iteratively merged in pairs to maximize communication savings, reducing the count to p. For instance, submatrices with interleaved rows or columns are merged. The process repeats, prioritizing pairs with the highest communication gain, until p subgraphs remain.

Merging subgraphs reduces communication overhead but increases the risk of load imbalance. In particular, the workload of a merged subgraph may exceed that of the largest subgraph, violating the optimization objective in Equation 2. To mitigate this, we introduce an additional input parameter provided by user, *ratio*, which constrains merging such that the combined workload remains within *ratio* times the average workload across all subgraphs. In FaaSBoard, the system is configured by default to generate the most possible balanced subgraphs. This is achieved by iteratively fine-tuning the *ratio* until it reaches its minimum possible value.

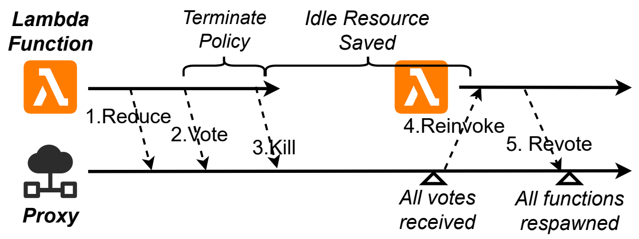


Figure 8: Terminate-and-Respawn Mechanism.

The binpacking algorithm operates efficiently with minimal overhead, as it processes no more than a few dozen graph partitions given our datasets.

The effectiveness and efficiency of our algorithm are analyzed in detail in Section 8.3.3.

6.2 Proactive Terminate-and-Respawn

Although the algorithm in Section 6.1 aims to optimize load balancing, achieving perfect load balance is often impractical. The challenge arises primarily due to the power-law degree distribution commonly found in real-world graph datasets [18], where a small subset of vertices possesses extremely high degrees. As a result, resources may remain idle, creating opportunities for optimization. Serverless, along with FaaSBoard's execution model, enables us to leverage this opportunity. As shown in Figure 2a, functions off the diagonal receive the vector at the start and send responses at the end without retaining vertex state across iterations. Serverless is ideal for such stateless computations, where functions can be spawned and terminated in each iteration without disrupting the overall query execution. This allows functions to utilize computational resources only for the necessary duration, thereby reducing unnecessary costs.

Here, we introduce our *Proactive Terminate-and-Respawn* mechanism. The execution flow of this mechanism is shown in Figure 8. Specifically, after sending the reduce vector, functions send a vote message to the Proxy and wait for others to complete. Rather than idling until the next iteration begins, they autonomously decide to exit the function execution based on a *Terminate Policy*. This termination is communicated via a kill message, which the Proxy uses to track the number of terminated functions. After some time, when the Proxy determines it has received sufficient vote messages, it reinvokes all terminated serverless functions. Each respawned function sends a revote message to the Proxy to signal its restoration. Once all terminated functions have been respawned, the next iteration proceeds as usual.

A key challenge in this mechanism is designing an effective Terminate Policy, in which the decision-making time is significantly smaller than the resource idle time for more cost savings. Due to the dynamic workload, the policy should also be able to detect appropriate load imbalance opportunities for termination, and avoid unnecessary terminations that could introduce latency overhead when the workload is balanced. Here, we propose a lightweight Terminate Policy. The idea is that if a function remains idle for a prolonged period, it is likely to stay idle for even longer. To implement this, we assign a grace period of T milliseconds to each function after its sending of the vote message. If all votes are received within this T ms window, the function continues to run. However, if the grace period elapses (i.e. the function doesn't receive the response of vote in T ms), the function is terminated. In FaaSBoard, we set the default T to 300 ms.

Users don't need to modify the execution process shown in Figure 2b, as the implementation is internally handled within the vote operation. Specifically, if the Allreduce returns a timeout signal, the function is terminated immediately. Upon reinvocation, the function retries the Allreduce to send a Revote message and resumes execution seamlessly, as if no termination had occurred.

While latency overhead may occur due to cold starts during function reinvocation, where resource initialization and subgraph reloading take place, this is often mitigated in practice. For most graph applications, the termination period between iterations is typically less than one minute. This short duration increases the likelihood that the released function resources remain alive in the cluster, thanks to the keep-alive policies commonly employed by major cloud providers [45]. As a result, the container and subgraph data can often be reused without requiring reinitialization from scratch.

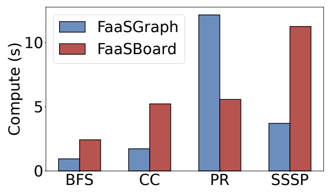
Section 8.3.4 presents the analysis of the cost savings and latency overhead linked to this mechanism.

7 Implementation

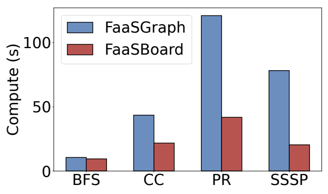
Tools. We developed the FaaSBoard system using approximately 9.6K LOC of C++ code. The Proxy is built using epoll, enabling it to continuously manage connections, send and receive messages, and aggregate intermediate vertex states. We implement the dual-mode message passing as well as the work stealing mechanism from Gemini [59] in FaaSBoard. Furthermore, we leverage the AVX2 instruction set for optimized processing of our uint32 data, operating on 256-bit batches.

Parallel Strategy. As shown in Figure 2a, a function must perform the broadcast-compute-reduce sequence for each subgraph. To enhance execution efficiency, we employ distinct parallel strategies for each operation, carefully tailored to their respective computational characteristics and resource demands. Specifically, we use graph parallelism for broadcast and reduce, enabling all communications to occur simultaneously to fully leverage network bandwidth. In contrast, we apply vertex parallelism for compute, allowing concurrent edge scanning for vertices but processing each subgraph sequentially to prevent compute resource contention.

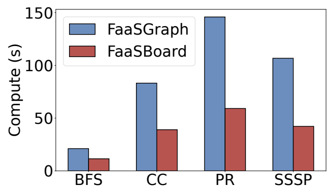
Proxy Scaling and Query Scheduling. Unlike Kubernetes, which supports scaling based on customized resource metrics, AWS Fargate only allows scaling based on CPU or memory utilization. This creates a mismatch with the network-intensive nature of the Proxy. Therefore, FaaSBoard adopts the Bounded Scaling Policy from FaaSGraph [35] for managing Proxies. Each Proxy is assigned a defined *capacity*, representing the maximum number of concurrent queries it can handle to avoid network contention. This capacity is computed by dividing the available bandwidth by the maximum potential message size (i.e., assuming the Dense format). For each incoming query, the system decides whether to queue it at an existing Proxy or to scale up a new one, aiming to minimize the



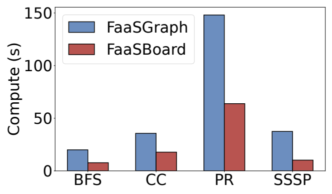
(a) Livejournal



(b) Twitter



(c) Friendster



(d) Rmat27

Figure 9: Compute Time Comparison.

Table 1: Hardware, software setup and benchmarks.

	Configuration			
Hardware	*FaaSGraph:* CPU: Intel Xeon E5-2676 V3 @2.40Ghz, Instance: m4.10xlarge, Cores: 40, DRAM: 160GB			
Software	*FaaSGraph:* Ubuntu 22.04, Golang 1.18.4, Docker 20.10.17 *FaaSBoard:* Image public.ecr.aws/lambda/provided:al2023			
Service	*FaaSBoard:* AWS Lambda, AWS Fargate			
Function	*FaaSGraph&FaaSBoard:* Cores: 2, DRAM: 3538MB			
Benchmark	Graph	Vertices	Edges	#Functions
	Livejournal	4.84M	68.9M	1(BFS,CC,PR,SSSP)
	Twitter	41.6M	1.46B	6(BFS,PR),12(CC,SSSP)
	Friendster	65.6M	1.80B	8(BFS,PR),14(CC,SSSP)
	Rmat27	134M	2.14B	9(BFS,PR),17(CC,SSSP)

estimated waiting time. Unused Proxy is automatically reclaimed after remaining idle for 15 minutes.

8 Evaluation

This section validates the disaggregated serverless architecture by evaluating the performance of FaaSBoard.

8.1 Setup

Baseline Systems. We use FaaSGraph [35], a state-of-the-art serverless graph processing system as our baseline for overall evaluation. To compare partitioning strategies, FaaSBoard implements many graph partitioning methods from prior researches [2, 3, 35, 40, 59]. Additionally, we integrate FMI [13] to evaluate communication approaches.

Hardware and Software. Table 1 summarizes the experimental configurations. FaaSBoard is deployed fully on AWS Lambda and AWS Fargate. Since FaaSGraph relies on a custom serverless runtime, we deploy it on an EC2 instance. To select the instance type, we evaluate CPU performance across AWS instance generations (e.g., m7, m6, m5, m4), and discover that the m4 series offers performance closest to Lambda, leading us to choose m4.10xlarge for our experiments. To simulate an environment identical to AWS Lambda, we disable FaaSGraph's shared-CPU and shared-memory. However, all containers run on the same instance to measure the upper-bound performance.

Resource Config. FaaSBoard adopts the same resource configuration as FaaSGraph, allocating 2 vCPUs and 3 GB of memory per function. For each graph benchmark, we use FaaSGraph's Resource Optimizer to determine the number of functions, with details listed in Table 1[3]. We deploy the Proxy on serverless container service, where each container is configured with 16 vCPUs and 64 GB of memory.

Pricing Strategy. For our evaluation on monetary cost, we naturally adopt AWS's pricing model. In FaaSboard, the total monetary cost comprises two components: (i) the cost of all AWS Lambda functions, which are employed as our workers; and (ii) the cost of AWS Fargate, which we introduce as proxy to handle communication.

Let M_l denote the cost of running all AWS Lambda functions and M_f the cost of running AWS Fargate. The cost M_l is calculated as $M_l = t \times m_l \times c_l$, where m_l is the total memory (in GB) allocated to all Lambda functions, and c_l is the per-second-per-GB cost for Lambda. The AWS Fargate cost M_f, following its dual-component pricing model, is given by $M_f = t \times (v_f \times c_v + m_f \times c_m)$, where v_f and m_f are the total number of vCPUs and the memory (in GB) allocated for Fargate, respectively; c_v and c_m are the per-second-per-vCPU and per-second-per-GB pricing coefficients for Fargate, respectively. Here, t stands for the end-to-end execution time of a graph computing task. The total monetary cost for FaaSBoard is thus given by:

$$M = M_l + M_f = t \times m_l \times c_l + t \times (v_f \times c_v + m_f \times c_m) \qquad (3)$$

For the x86 architecture, c_l is set to \$0.0000166667. The Fargate pricing coefficients c_v and c_m are set to \$0.05056 and \$0.00553, respectively, based on the price list in AWS.

We maintain AWS Lambda's pricing for FaaSGraph, assuming it is a serverless service and follows the pay-as-you-go billing. However, the billing for FaaSGraph corresponds solely to the M_l component within the FaaSBoard cost model, with no additional charges incurred for the proxy.

Graph Benchmarks. We use four widely adopted graph datasets from prior work [2, 35, 40, 59]: LiveJournal(lj) [5], Twitter(tw) [27], Friendster(fr) [54] and RMAT-27(rm) [10]. The graph sizes are summarized in Table 1. We select four standard graph workloads: Breadth-First-Search (BFS), PageRank (PR), Connected-Components (CC) and Single-Source-Shortest-Path (SSSP)[4].

[3]CC runs on an undirected graph, while SSSP runs on a weighted graph. Their CSR size is larger than the directed graph used by BFS and PR and thus needs extra functions to process.

[4]BFS and SSSP use vertex 0 as the source. For PR, we execute 20 iterations.

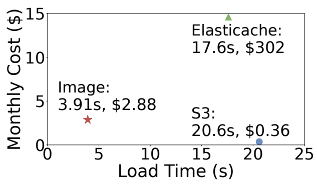


Figure 10: Load time and monthly monetary cost.

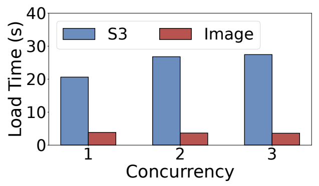


Figure 11: Load time under different concurrency.

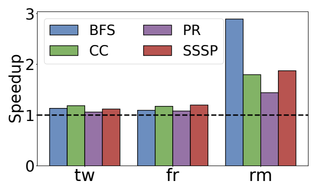


Figure 12: Speedup achieved by applying Proxy over FMI.

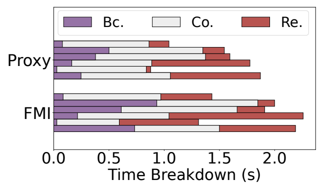


Figure 13: Execution breakdown. (Bc: Broadcast, Co: Compute, Re: Reduce)

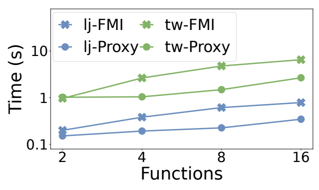


Figure 14: Performance of sending vector to different number of functions.

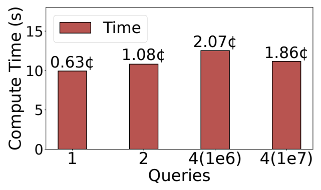


Figure 15: Concurrent tw-BFS with a single Proxy. The number indicates cost.

8.2 Overall Comparison

Time. Figure 9 presents the in-memory compute time of FaaSGraph and FaaSBoard. The results show that FaaSBoard performs differently on small and large graphs. For tw, fr and rm, FaaSBoard outperforms FaaSGraph by 1.1×-3.8×. This highlights that, despite the overhead in serverless cloud services, FaaSBoard can still perform better in performance. With FaaSBoard, we benefit both from the improved performance brought by the disaggregated serverless architecture and the operational simplicity provided by cloud services.

However, FaaSGraph outperforms FaaSBoard for small graphs like lj. This is because the subgraph-centric execution model reduces the number of iterations compared to the SpMV model we use. As a result, interactions with the Proxy or Coordinator are minimized, leading to lower synchronization overhead. This is further confirmed by the performance of PageRank, where FaaSBoard remains faster than FaaSGraph when both run the same number of iterations. Therefore, FaaSBoard is suitable for processing large graphs. In the following contents, we exclude lj from further discussion.

Cost. Next, we analyze the monetary cost of one-shot query. In FaaSGraph, the cost is solely determined by serverless functions, which depend on compute time and resource usage. In contrast, FaaSBoard incurs an additional cost for the Proxy. For tw, fr and rm, FaaSBoard achieves a lower monetary cost in almost all benchmarks, reducing costs by 7.4%–63.7%. However, for tw-BFS, FaaSBoard incurs a 72.9% higher cost due to its similar performance compared with FaaSGraph. This indicates that when running one-shot query, FaaSBoard has better cost efficiency for handling large graph datasets and intensive applications.

8.3 Detailed Evaluation

In this subsection, we examine each design choice and provide a detailed analysis on performance and cost efficiency. Unless specified otherwise, we follow all resource and execution configurations outlined in Section 8.1.

8.3.1 *Graph Loading.*

We begin by analyzing our graph loading mechanism. In our evaluation, we measure the loading time of fr-BFS, where 8 functions run concurrently to load the subgraphs. Figure 10 presents the loading time and estimated monthly cost for different storage approaches. The loading time is averaged over 10 consecutive runs, with a 1-hour interval between each run for the image-based approach. S3 and ElastiCache involve only storage costs, while the image-based approach addtionally involves the cost of invoking functions hourly over a month. We observe that the image-based approach significantly improves loading performance with only a slight increase in cost compared with S3. Additionally, the image-based approach delivers stable load performance with a CV of 0.088. ElastiCache is the most expensive option due to its memory-based design.

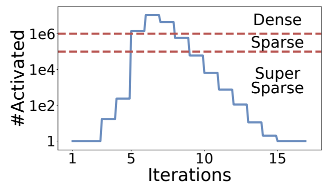
Figure 11 illustrates the load time for 1, 2, and 3 concurrent fr-BFS jobs. The load time is measured from the start of the invocation to the completion of loading of the last function. As concurrency increases, we observe that the load time for S3 increases due to its bandwidth bottleneck, while the image-based approach maintains stable load performance, as it can leverage bandwidth across multiple servers.

8.3.2 *Collective Communication.*

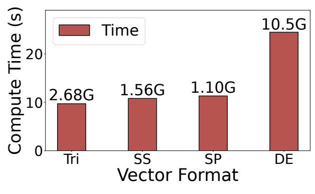
In this subsection, we assess the Proxy-based communication by comparing it with FMI [13]. As we cannot reproduce FMI's direct function connection behavior in AWS Lambda, we simulate it in FaaSBoard by implementing function-to-function communication via Proxy for send and receive operations. On top of this, we implement FMI's binomial tree broadcast and reduce.

Figure 12 illustrates the compute time speedup on tw, fr, and rm when applying Proxy compared to FMI, achieving a speedup of 1.05×-2.89×. Proxy-based communication has a significant impact on larger graphs like rm, as they are split into more partitions, resulting in a deeper binomial tree in FMI. Furthermore, the increased number of vertices amplifies communication volume. Figure 13 presents the execution breakdown of a single PR iteration on the tw graph. The communication accounts for a substantial portion in FMI setting, whereas it is significantly reduced in Proxy setting.

Next, we conduct a microbenchmark to examine the impact of shallow communication structure when using Proxy. We broadcast vectors of varying lengths to different numbers of functions and measure the broadcast time. The vector length is computed



(a) Format Selection



(b) Performance

Figure 16: Analysis of Trimode format on tw-BFS. The number indicates the total communication volume. (SS: Super Sparse, SP: Sparse, DE: Dense)

as $\#vertices * 4$ bytes for graphs lj and tw, which differ by approximately an order of magnitude. As shown in Figure 14, Proxy consistently outperforms FMI when using more than four functions. As the number of functions increases, Proxy maintains relatively stable performance due to its high bandwidth, whereas FMI exhibits a increase in broadcast time due to its deeper tree structure. However, at 16 functions, Proxy's network bandwidth becomes saturated, indicating the need for further scaling.

Next, we use tw-BFS to demonstrate the multiplexing capability of Proxy and the benefits of the trimode vector format. Figure 15 presents the compute time and monetary cost for running concurrent queries with a single Proxy, measured from invocation to the completion of the last query. We observe that the monetary cost increases sublinearly with the number of queries, as the cost of Proxy is amortized. However, at four concurrent queries, we experience a cost rise due to suboptimal performance. This is caused by the high communication volume generated by the dense format, leading to network congestion. To validate this, we increase the Sparse-Dense threshold to 10^7, rerun the experiment, and observe a reduction in both latency and cost.

Figure 16 evaluates the impact of the Trimode vector format. As shown in Figure 16a, the number of activated vertices in BFS fluctuates across iterations. Over 17 iterations, the algorithm utilizes the Dense format in 3 iterations, the Sparse format in 1 iteration, and the Super Sparse format in the remaining iterations, demonstrating the need of combining these formats. Figure 16b presents the compute time of tw-BFS when employing the Trimode approach compared to using a single format exclusively. While the communication volume is higher than that of the Super Sparse and Sparse modes, the Trimode configuration achieves superior performance by leveraging AVX instruction optimizations enabled by the Dense format. However, using only the Dense format increases communication overhead, leading to performance degradation. Therefore, a mixed-mode approach is necessary to achieve optimal performance.

8.3.3 *Graph Partitioning.*

In this subsection, we evaluate the performance of 1D, 2D-Cyclic, 2D-Staggered and our 2D-Balanced partition methods, covering the majority of chunk-based algorithms to the best of our knowledge. We implement all methods on FaaSBoard. We implement 1D partitioning by applying virtual cuts. Specifically, after determining the cut positions along the rows, we apply the same cuts to the columns at identical positions, and ensure that all generated matrices within the same row are assigned to the same function. This approach allows 1D partitioning to align with the broadcast-compute-reduce execution model described in Figure 2b. We use PR as our sample graph application following previous research [59].

Table 2: Comparison of graph partitioning on different graphs when running PR in terms of compute time (s).

Graph-App	1D	2D-Cyclic	2D-Staggered	2D-Balanced
tw-PR	45.6	OOM	OOM	43.1
fr-PR	66.0	60.3	63.3	58.4
rm-PR	224	OOM	OOM	66.7

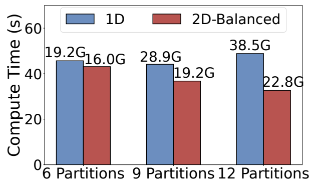


Figure 17: Performance with varying #partitions. The number indicates the total communication volume.

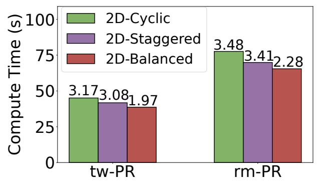
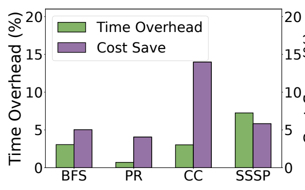


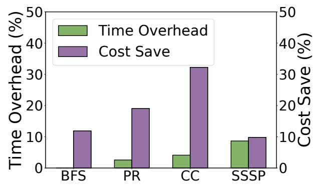
Figure 18: Performance with varing 2D methods. The number indicates the balance ratio produced.

Table 2 presents the compute time under different partition methods, with our 2D-Balanced partitioning proving to be the most effective. When comparing 1D and 2D-Balanced partitioning, the performance gap widens as the graph size increases. This is because larger graphs are divided into more partitions, leading to increased communication overhead in the 1D method. To validate this, we conduct a microbenchmark on tw-PR, where we split the graph into varying #partitions. Figure 17 presents the compute time and total communication volume. As the graph is split into more partitions—either because a single function cannot handle a large subgraph or to leverage additional computational resources for better performance—the 2D-Balanced partitioning consistently outperforms 1D methods and successfully maintains scalability. This advantage stems from the lower communication volume when using more partitions.

When comparing the 2D methods in Table 2, both 2D-Cyclic and 2D-Staggered fail to balance the workload, leading to out-of-memory (OOM) in tw and rm due to one function holding a significantly larger subgraph. To assess the impact on execution, we migrate FaaSBoard back to a local server (a single m4.10xlarge instance). In this setup, each process is responsible for a subgraph, and we restrict the number of available cores for each process to two. Proxy remains on the AWS Fargate. The only difference is that the memory limitation is removed. Figure 18 illustrates the compute time and balance ratio (i.e., the maximum #edges in a subgraph divided by the average #edges across all subgraphs) for tw and rm. The results demonstrate that 2D-Balanced is the most efficient method, as it achieves better load balancing across partitions. In summary, our experiments highlight the effectiveness of 2D-Balanced partitioning in execution and its ability to handle large graphs using small functions.



(a) Balanced Partitioning



(b) Unbalanced ($ratio$ = 2)

Figure 19: Compute time overhead and cost saving under Terminate-and-Respawn mechanism running on tw.

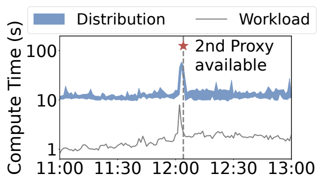
Finally, we evaluate the runtime across different partitioning methods using tw as an example. All algorithms are executed on a single m4.10xlarge instance. The 1D, 2D-Cyclic, and 2D-Staggered methods complete in under 1s due to their significantly lower complexity. In contrast, 2D-Balanced takes 38.8s to finish. However, we consider it acceptable as the partitioning algorithm runs only once and remains negligible compared to other essential preprocessing steps, such as building the CSR (408s), constructing and saving the partitioned CSR (31.4s), and packing and submitting the image to the repository (650s). Additionally, our optimizations of parallel and approximate binary search accelerates the narrowing of the decision boundary, significantly reducing algorithm runtime. For instance, the unoptimized algorithm requires 31 iterations and takes 263s to complete. Introducing parallelism reduces the runtime to 125s with 9 iterations, while further incorporating approximation cuts the runtime to 38.8s with only 2 iterations.

8.3.4 Terminate-and-Respawn. In this subsection, we evaluate the compute time overhead and cost savings of our mechanism compared to disabling it, using tw as the example graph dataset. Figure 19a shows that with default graph partitioning and the most balanced subgraphs, our mechanism achieves 4.0%-13.9% cost savings at the cost of 0.7%-7.2% execution overhead. This indicates that idle resources still exist with our load-balanced partitioning, and the Terminate-and-Respawn mechanism serves as a backup to further optimize resource and reduce costs. In Figure 19b, where we set the $ratio$ from Section 6.1 to 2, cost savings increase significantly because of more unbalanced workload, while the compute overhead is negligible in comparison.

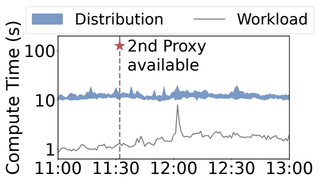
8.4 Trace Analysis

In this section, we evaluate our performance using a 2-hour trace from FaaSGraph [35] on peak hour. We select tw-BFS as our example graph workload. Since the graph size for tw is approximately 100X larger than the dataset analyzed in FaaSGraph, we scale down the original trace by a factor of 100 to ensure the experiment fits within our budget.

Figure 20a shows the compute time distribution over 2 hours. We observe a surge in compute time around 12:00, caused by the extended time to scale up a Proxy in Fargate, which takes approximately 43 seconds. Before query spike arrives, following the scaling policy in FaaSGraph [35], queries tend to queue to avoid the high proxy scaling overhead. However, when too many queries accumulate, a Proxy is finally scaled, resulting in even further delays as the FaaSBoard system response slowly in handling the surging workload.



(a) Proxy on Fargate



(b) Proxy with Faster Initialization

Figure 20: Compute time distribution under real-world workload. The number of Proxy is scaled from 1 to 2, with the exact time of scaling marked by a red star.

Compared to Fargate's container scaling time, we measure the time required to spawn a Proxy container on our local server to be approximately 1 second. To simulate a lower container initialization overhead in Fargate, we conduct an experiment where all proxies are kept alive but become available to the query scheduler after 1 second. The performance is shown in Figure 20b. We observe that FaaSBoard maintains steady performance even under fluctuating workloads. Notably, the second Proxy is scaled up more proactively due to its low overhead, and it is quickly deployed to handle graph processing queries. This highlights the importance of optimizing the initialization time of serverless container services on the cloud, in addition to function services, which can be addressed via many approaches such as checkpointing [4, 15, 25, 50] and container caching [8, 31, 55].

9 Conclusion

We present FaaSBoard, a graph processing system on serverless cloud services which adopts a disaggregated architectural design, integrating an autonomous-elastic computing mechanism with a multi-tier data system to optimize both resource and overall execution efficiency. By addressing the limitations of monolithic function architectures in exploiting intra-job elasticity and reducing communication overhead, FaaSBoard effectively overcomes the performance and cost inefficiencies prevalent in existing serverless graph processing systems. Experimental results show that FaaSBoard significantly outperforms FaaSGraph in terms of both compute performance and monetary cost.

References

[1] [n. d.]. iPerf. https://iperf.fr.

[2] Yousuf Ahmad, Omar Khattab, Arsal Malik, Ahmad Musleh, Mohammad Hammoud, Mucahid Kutlu, Mostafa Shehata, and Tamer Elsayed. 2018. LA3: A scalable link-and locality-aware linear algebra-based graph analytics system. *Proceedings of the VLDB Endowment* 11, 8 (2018), 920–933.

[3] Michael J Anderson, Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Theodore L Willke, and Pradeep Dubey. 2016. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 313–322.

[4] Lixiang Ao, George Porter, and Geoffrey M Voelker. 2022. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 730–746.

[5] Lars Backstrom, Daniel P. Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the Twelfth ACM SIGKDD International Conference*

on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006. ACM, 44–54. doi:10.1145/1150402.1150412

[6] Thomas Bodner, Theo Radig, David Justen, Daniel Ritter, and Tilmann Rabl. 2025. An Empirical Evaluation of Serverless Cloud Infrastructure for Large-Scale Data Processing. *arXiv preprint arXiv:2501.07771* (2025).

[7] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 315–328.

[8] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

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

[10] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

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

[12] Rishan Chen, Mao Yang, Xuetian Weng, Byron Choi, Bingsheng He, and Xiaoming Li. 2012. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*. 1–13.

[13] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. 2023. Fmi: Fast and cheap message passing for serverless functions. In *Proceedings of the 37th International Conference on Supercomputing*. 373–385.

[14] Timothy A Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–25.

[15] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[16] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[17] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 363–376.

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

[19] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*. 152–166.

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

[21] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Ditto: Efficient serverless analytics with elastic parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 406–419.

[22] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. 2023. Executing microservice applications on serverless, correctly. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 367–395.

[23] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. 2022. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 697–713.

[24] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 427–444.

[25] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective checkpoint orchestration for serverless hot-starts. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 298–316.

[26] Dimitrios Koutsoukos, Renato Marroquín, Ingo Müller, and Ana Klimovic. 2025. Adaptive data transformations for QaaS. (2025).

[27] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*. ACM, 591–600. doi:10.1145/1772690.1772751

[28] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 741–761.

[29] Yilong Li, Seo Jin Park, and John Ousterhout. 2021. MilliSort and MilliQuery: Large-Scale Data-Intensive Computing in Milliseconds. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 593–611.

[30] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. 2020. Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 399–408.

[31] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2022. Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing. In *2022 USENIX annual technical conference (USENIX ATC 22)*. 69–84.

[32] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. 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*. 782–796.

[33] Mengfan Liu, Wei Wang, and Chuan Wu. 2025. Optimizing distributed deployment of mixture-of-experts model inference in serverless computing. *arXiv preprint arXiv:2501.05313* (2025).

[34] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray CC Cheung, and Jianfei He. 2023. In-network aggregation with transport transparency for distributed training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 376–391.

[35] 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 *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 385–400.

[36] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. 2021. {SONIC}: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 285–301.

[37] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. {ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 303–320.

[38] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[39] Jasmina Malicevic, Amitabha Roy, and Willy Zwaenepoel. 2014. Scale-up graph processing in the cloud: Challenges and solutions. In *Proceedings of the Fourth International Workshop on Cloud Data and Platforms*. 1–6.

[40] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. 2020. Graphite: A NUMA-aware HPC system for graph analytics based on a new MPI* X parallelism model. *Proceedings of the VLDB Endowment* 13, 6 (2020), 783–797.

[41] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. 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*. ACM, 115–130. doi:10.1145/3318464.3389758

[42] Matthew Perron, Raul Castro Fernandez, David Dewitt, Michael Cafarella, and Samuel Madden. 2023. Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–25.

[43] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. 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*. ACM, 131–141. doi:10.1145/3318464.3380609

[44] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. 2021. Scaling distributed machine learning with {In-Network} aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 785–808.

[45] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 205–218.

[46] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless linear algebra. In *Proceedings of the 11th ACM symposium on cloud*

computing. 281–295.
[47] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. 2024. Vexless: a serverless vector data management system using cloud functions. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
[48] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. 2021. Dorylus: Affordable, scalable, and accurate {GNN} training with distributed {CPU} servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 495–514.
[49] Lucian Toader, Alexandru Uta, Ahmed Musaafir, and Alexandru Iosup. 2019. Graphless: Toward serverless graph processing. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 66–73.
[50] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*. 559–572.
[51] Michael Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. 2021. Boxer: Data analytics on network-enabled serverless platforms. In *11th annual conference on innovative data systems research (CIDR 2021)*. ETH Zurich.
[52] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. Sync or async: Time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Notices* 50, 8 (2015), 194–204.
[53] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.
[54] Jaewon Yang and Jure Leskovec. 2012. 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*. IEEE Computer Society, 745–754. doi:10.1109/ICDM.2012.138
[55] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 335–350.
[56] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1489–1504.
[57] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus:{NIMBLE} task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 653–669.
[58] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.
[59] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 301–316.