Graphless: Toward Serverless Graph Processing

Lucian Toader, Alexandru Uta, Ahmed Musaafir, Alexandru Iosup
Vrije Universiteit Amsterdam, the Netherlands
L.Toader@atlarge-research.com, A.Uta@vu.nl, A.A.M.Musaafir@vu.nl, A.Iosup@vu.nl

***Abstract*—Our society is increasingly solving complex problems through the use of graph processing. Existing graph processing systems focus on performance, which allows addressing ever-larger and more complex problems. They also require uncommon expertise to properly deploy and utilize. To make graph processing generally accessible—to small and medium enterprises and institutions, to common research groups, to individuals—, in this work we design and implement the Graphless graph-processing system.**

Graphless is based on the serverless paradigm, which proposes to simplify computing by letting developers only focus on small, stateless functions, which are deployed and managed automatically. We address with Graphless the key challenge of combining the stateless functions assumed by serverless computing with the (opposite) data-intensive nature of graph processing. Graphless tackles this challenge through an architectural approach that allows it to deploy with push or with pull operation, and a collection of backend services, such as an orchestrator and a memory-as-a-service component.

We implement Graphless and conduct with it real-world experiments using Amazon Lambda for cloud-based serverless resources. Using the LDBC Graphalytics benchmark, we analyze Graphless, and compare its performance and operational cost with the graph-processing systems Apache Giraph (big data domain) and GraphMat (HPC). Overall, we show evidence Graphless provides performance and cost-efficiency similar to Giraph, for algorithms that can benefit from fine-grained elasticity, and lower than GraphMat, but is architecturally easier to deploy, and provides both push and pull operation.

1. Introduction

Graphs are useful for extracting meaning and expressing data connectedness at arbitrarily deep levels, streamlining the process of finding known and hidden relationships to solve complex societal problems [1], [2], [3]. So far, the graph processing community has focused on running graph algorithms on tightly-coupled, high-end infrastructure, using complex parallel [4] or distributed [5], [6] software. These approaches promise high performance, but effectively lock out the small and medium organizations that cannot afford investing in high-end infrastructure, or do not have the knowledge to operate complex graph-processing software. In the past few years, similar issues have led to the emergence of the serverless computing paradigm. Serverless systems require from the developer only simple, small functions, and a specification of how they interconnect; the operational concerns are abstracted away [7]. This gives serverless systems ease-of-use, fine-grained scalability, and, when using cloud resources, fine-grained pay-per-use billing. However, the serverless paradigm posits that functions are stateless, which goes contrary to the big data nature of graph processing. Toward a serverless graph-processing system, our work is the first to ask and to address the research question *How to combine serverless and graph processing into a serverless graph-processing system?*

Graph processing enables a wide array of applications. Use cases include social network analysis [1], [8], machine learning [9], data mining [10], bioinformatics [2], telecommunications [11], [12], fintech [13], and fraud detection [3]. Graph algorithms are generally iterative, data- and communication-intensive, and exhibit unpredictable computation and data-access patterns, leading to poor data-locality properties [14]. To address these challenges, modern graph processing runs in complex environments, such as parallel HPC systems, distributed commodity clusters, and elastic (commercial) clouds. Many large-scale graph processing platforms exist [15], [16], for big data processing, such as Apache Giraph [5] and GraphX [6], and for the HPC community, such as GraphMat [4].

The design, development, deployment, and management of the infrastructure and platform for graph processing, in short, the complex processing ecosystem [17], raises important challenges. First, managing such clusters and the processing system itself is non-trivial, requires expert knowledge, and incurs significant costs. Second, distributed graph processing systems generally run on a fixed number of compute nodes, making such systems brittle and unable to respond to changes in the workload by scaling-in or -out; the systems are either over- or under-provisioned. In practice, under-provisioning leads either to crashing or thrashing, and over-provisioning leads to resource waste due to the irregularity of graph processing workloads [18].

The commonly held view that large-scale graphs are only a problem for large organizations is inaccurate [19]. Therefore, leveraging cost-efficient deployments for graph-processing is an important problem. Recently, *serverless computing* has emerged as a paradigm for easy-to-use, cost-efficient deployment. Serverless is based on the *FaaS* layer of abstraction in cloud computing [7], aiming to separate operational from business concerns through full automation of provisioning and server management. Serverless shifts the operational responsibility to the cloud provider, is pay-per-

use billed in millisecond increments, and scales up and down automatically with fine granularity. This promises to allow developers to focus only on the software they are building. However, to achieve its promise of full automation, serverless posits that functions should be small, short-lived, and stateless. This contrasts with the data-related requirements of graph processing and poses a key conceptual challenge to serverless graph processing. In this work, we address this challenge, and the research question, with a two-fold contribution:

1) We design the Graphless serverless graph-processing system (Section 2). Graphless provides the key components for automating the operation of graph processing systems, which it combines to support two operational architectures for push- and for pull-based serverless graph processing.
2) We conduct real-world experiments with a prototype implementation of Graphless (Section 4). We compare Graphless with two leading graph-processing platforms, Giraph and GraphMat.

2. Graphless Design

We propose in this section the design of the Graphless serverless graph-processing system.

2.1. Requirements

We identify five key requirements for a serverless graph-processing system:

R1 *Target algorithms and graphs.* The system must support a broad range of applications from a variety of domains. Similarly to state-of-the-art graph-processing systems, the system must support the broad class of iterative graph algorithms, and graphs that are directed or undirected, weighted or unweighted, and can mutate at runtime.

R2 *Fine-grained elasticity.* To process efficiently the highly irregular workloads of graph processing, the system must be elastic [18], that is, be able to scale-up and -down seamlessly with the graph workload. Moreover, to avoid excessive overprovisioning, elasticity should be fine-grained.

R3 *Fast, scalable, low-latency remote memory.* Unlike the computation nodes used by other graph processing systems, FaaS platforms are generally stateless. To access graph data, they require a (remote) memory service with high throughput, low latency, and high elasticity.

R4 *Platform independence.* Users should be able to freely choose where to deploy the system based on their own requirements, such as performance *and* cost. Therefore, the system must be independent of its environment, whether it is a public cloud or an on-premises infrastructure.

R5 *High usability and automated resource management.* To satisfy a large and diverse user base, the system must require from its users only minimal and common knowledge of resource management. In particular, system deployment should be fully automated.

2.2. Architecture

Graphless is based on a serverless architecture, a form of event-driven architecture in which components are either fully-managed services or ephemeral containers that execute user-defined code and are triggered by events (state changes). It uses the Pregel computation model [20], a form of the Bulk Synchronous Parallel model [21], as its execution model.

To meet R4, the design of the architecture is modular, with each component exposing a platform-agnostic API. The memory, compute, storage, and queuing services can have their implementations swapped with custom built components or with managed services.

Requirement R5 is achieved through the serverless compute, managed distributed Redis (Elasticache[1]), and the easy-to-use programming model. Unlike other graph processing systems, deploying Graphless is simple. During our experiments, we had to migrate the system from one region of the AWS Cloud to another. The entire procedure took just minutes to complete.

2.3. Front-End

The user interfaces with the system through the front-end. This part exposes the functionality required for the user to implement arbitrary graph algorithms, through a high-level programming abstraction.

Programming Model Graphless provides a simple method of implementing graph algorithms through a vertex-centric high-level programming abstraction. In the vertex-centric paradigm, processing happens through parallel executions of a user-defined function for each individual vertex. Communication and data partitioning are transparent to the user. To implement a graph algorithm, users write a single function, which is executed for each active vertex; a vertex can vote to become inactive at any time during execution. Computation ends when all the vertices become inactive. The function has access to an API that exposes functionality for communication, update, aggregation, topology mutation, and execution termination.

API The user is required to implement the *Compute()* function, which will be executed for each active vertex in every iteration. Vertices communicate with each other via message-passing. Through the API, it is possible to retrieve and modify the edges and the value of the vertex and send messages to other vertices. The API allows users to create aggregators, that can hold global state for the duration of a superstep. Aggregators can be used to compute minimums and maximums, sum values, etc.

2.4. Back-End

The back-end is the part of the system that manages all the phases required to execute user-defined applications.

1. https://aws.amazon.com/elasticache/

Among its tasks are communication and superstep synchronization, data loading and partitioning, fault-tolerance, and function invocation.

Loader. The loader is responsible for the initialization phase of a graph computation. It parses the execution parameters and retrieves the input graphs from stable storage, passing them on to the memory engine; communication with the storage system takes place through a platform-agnostic API. The loader employs a fan-out pattern, splitting the work across multiple invocations of itself.

Orchestrator. This component is mainly responsible for data partitioning and synchronization. Once the graph is loaded in the memory engine, the orchestrator retrieves the metadata through the memory engine API, determines the number of workers that need to be launched in parallel based on configuration parameters, and partitions the data accordingly. The orchestrator ensures that the computation is evenly spread, so that all workers can finish processing their partition within the time constraints set by the FaaS platform. It initiates each superstep and determines when the processing has finished; it is also responsible for parsing and uploading the results to stable storage.

Function Manager. Each invoked worker function has a Function Manager that handles its execution. This component retrieves the vertex partition and messages assigned to a worker and makes them available to the user-defined function. The Function Manager also handles communication, vertex halting, and includes termination detection to determine when a superstep has ended.

Memory Engine To meet R3, we design the memory engine. It acts as an abstraction over a distributed set of Redis instances. In line with serverless architectures, we see the memory engine as a managed service - Memory-as-a-Service (MaaS). Through its API, vertices and messages can be stored, deleted, or updated, along with metadata required to keep track of state, such as the number of active vertices. Every Redis instance gets a partition of the data through sharding. Each new key is assigned to a partition using a consistent hashing function. Redis is an in-memory key-value store with sub-millisecond response times; it meets graph processing requirements, such as high-throughput and low-latency [22]. Amazon S3, a blob storage service, cannot meet the latency requirement and only supports eventual consistency. DynamoDB[2], a scalable NoSQL database, is prohibitively expensive for bursty workloads, such as graph processing. Moreover, it takes minutes to scale-up or -down, therefore it lacks the degree of elasticity that is required.

2.5. A1. Push-Based Architecture

In this section we discuss the design and implementation of a push-based architecture, depicted in Figure 1.

This architecture is push-based because there is a central orchestrator component that pushes to each worker the metadata for specific ranges of vertices that the worker must process. The orchestrator is the one responsible for starting new supersteps, partitioning the data, and detecting when the procesing has finished; the number of workers is determined based on the size of the data partition.

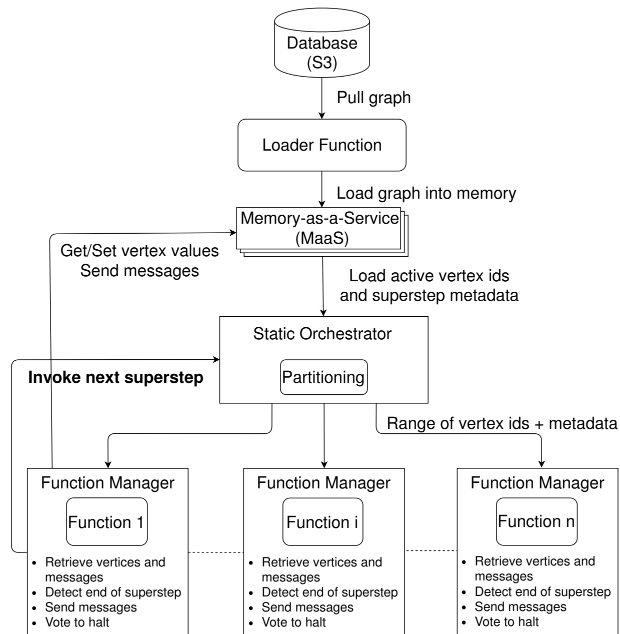

Figure 1. A1. Push-based architecture.

State persistence and communication between workers is done through the MaaS. The function manager provides all the functionality required for loading the partition data, message sending, and vertex updating and inactivation; it communicates with the MaaS whenever data needs to be sent or received. It also detects the end of a superstep and invokes the orchestrator function.

This architecture is pure FaaS, each worker function only runs for as long as it takes to process its payload. Fault tolerance is implemented in the form of function retries. If a worker function fails, it is simply restarted and retrieves its partition from the MaaS.

2.6. A2. Pull-Based Architecture

In this section we discuss the design and implementation of a pull-based architecture, depicted in Figure 2.

Instead of the orchestrator directly passing the partition metadata to the workers, it adds tasks for each worker in a queue. The invoked workers then pull the tasks from the queue along with the corresponding data from the MaaS.

In this architecture, along with data partitioning there is also compute partitioning. Each worker runs for the duration of the entire superstep; it is necessary to determine the optimal number of workers so that computation for each worker does not last longer than the function duration limit. Data partitioning consists of selecting an appropriate number of tasks that a vertex can pull from the queue at once so that computation is balanced across the workers. This architecture is no longer pure FaaS, since each function runs for the duration of a superstep.

2. https://aws.amazon.com/dynamodb/

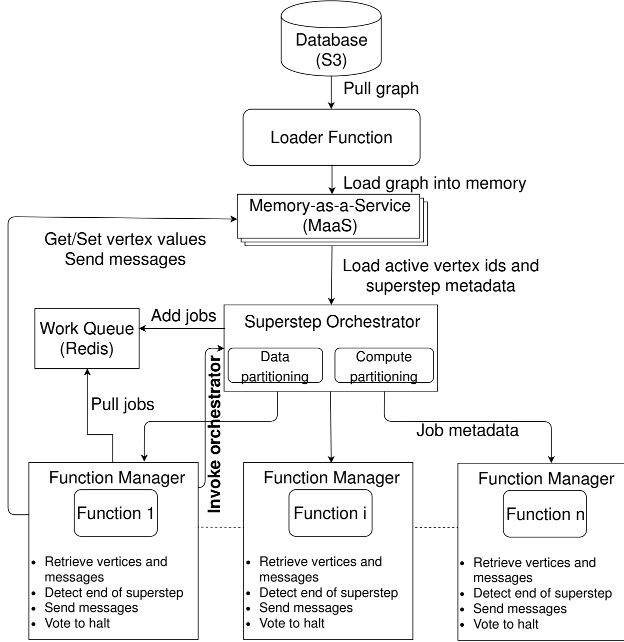

Figure 2. A2. Pull-based architecture.

3. Experimental Setup

This section is focused on the experimental setup used to evaluate Graphless. The goals include validating the outputs produced by the system, testing its scalability, its cost-effectiveness, and whether the serverless paradigm in its current form is sufficient to produce a high performance, efficient, and easy-to-use graph processing system.

Running and Validation. We select LDBC Graphalytics, an industry-grade, state-of-the-art graph processing benchmark [23] to evaluate the performance of Graphless. The outputs of running the algorithms with Graphless are validated against trusted results. Graphalytics also retrieves metrics, such as makespan and processing time.

Selected Graph Algorithms and Datasets. The algorithms cover a broad range of application classes, such as statistics, traversal, components, and distance calculation: Breadth-First Search (BFS), Weakly Connected Components (WCC), PageRank (PR), Community Detection by Label Propagation (CDLP), and Single-Source Shortest Paths (SSSP). We run these algorithms on the largest real-world weighted graph from the LDBC Graphalytics datasets. The graph is obtained from the gaming community and consists of 61,170 vertices and 50,870,313 edges.

Environment. The Graphless prototype runs on Amazon Cloud, using AWS Lambda as the FaaS platform providing the compute, EC2 virtual machines to run Redis instances for the memory engine, and S3 as stable storage. One of the objectives of this system is to automate resource management as much as possible; for experimentation purposes, we run Redis instances on manually managed EC2 instances. However, Amazon provides Elasticache, a managed Redis service that can be used as a drop-in replacement. Graphless is implemented in Go[3], a compiled, type-safe, garbage-collected programming language.

We chose AWS Lambda as the compute building block because it is the most mature serverless platform currently available. Functions are invoked asynchronously, using the Invoke action from the Lambda API with the InvocationType parameter set to Event. Lambda has a set of resource limits, which can be seen in Table 1. CPU and network bandwidth are proportionally allocated with memory.

TABLE 1. AWS LAMBDA RESOURCE LIMITS.

Resource	Limits
Memory	128MB - 3008MB (64MB increments)
CPU	(2 * m) / 3328MB, where m is the memory [24]
Temporary storage	512MB
File descriptors	1024
Execution duration limit	900s

The experiments involving Giraph and GraphMat are performed on Google Cloud using a 16-node configuration. Each node has 16 vCPUs and 60 GB of memory. A vCPU is a hardware hyper-thread of a 2.3 Ghz Intel Xeon E5 v3.

The main metrics used to evaluate the performance of Graphless are the following:

Processing Time. This metric represents the time required to execute one run with an algorithm, without including any platform-specific overhead, such as resource allocation, graph preprocessing, and loading the dataset or uploading the results to stable storage.

Cost of Execution. One of the objectives of our system is to be accessible to a broad variety of users, including individuals, such as researchers and small and medium enterprises. Cost is a major pain point for any type of user, but for those with limited capital, a high cost prohibits them from using a system entirely. For Graphless, the cost is represented by the sum of the cost incurred by AWS Lambda, as billed by the provider[4], and the cost incurred for provisioning the Redis storage VMs.

4. Experimental Results

This section is focused on the experimental evaluation of the Graphless prototype. The experimental setup is the one described in Section 3.

Main Findings:

1) Graphless achieves fine-grained elasticity.
2) Graphless is network bound; communication accounts for the majority of the execution time for all algorithms.
3) Graphless is cost-effective for algorithms that are not communication-intensive and can benefit from fine-grained elasticity.

3. https://golang.org/
4. https://aws.amazon.com/lambda/pricing/

4.1. Scalability Experiments

We examine how the components of the system, as well as the system as a whole, scale with load.

Memory Scalability. We assess the scalability of the memory engine by using a fixed number of one thousand maximum concurrent functions and varying the number of Redis instances that the memory engine can make use of.

This experiment shows that the system is not compute-bound, but is network-bound at this scale. The number of functions does not change, but we see clear improvements when adding new memory instances. We observe from Figure 3 that processing time improves with each addition of new instances. Communication between vertices seems to be the source of the bottleneck. This is confirmed by the fact that BFS, the least communication-intensive algorithm shows the least improvement; CDLP and PR, which are the most communication-intensive algorithms, show the most improvement. Processing time improvement is up to 41% for CDLP when moving from four to sixteen instances. The other algorithms experience improvements ranging from 21% to 36%.

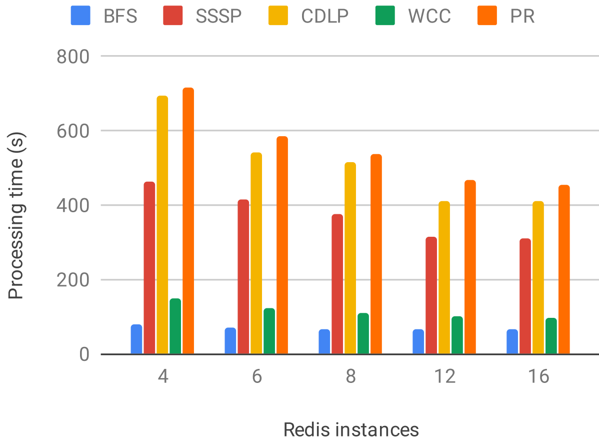

Figure 3. Memory engine scalability.

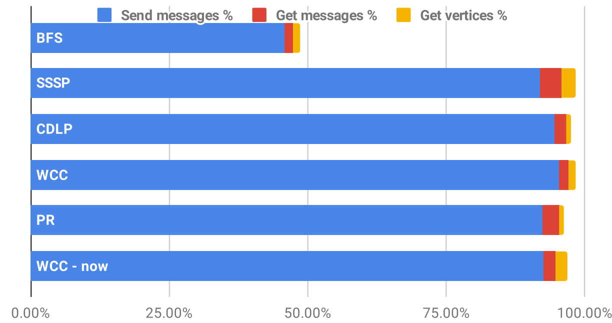

Figure 4. Communication impact on processing time using 12 Redis instances.

Figure 4 provides further insight into the problem. Excluding BFS, communication time is more than 90% of the aggregated execution time for the functions, the majority of which is message sending. Results were similar for all other runs, regardless of the number of instances. Since functions are stateless and cannot communicate directly with each other, the high number of messages for algorithms such as PR and CDLP becomes a problem. For example, during the execution of PR, each vertex sends messages along each of its neighbouring edge and all vertices are active during every superstep.

Compute Scalability. We assess the compute scalability of the system with a fixed number of Redis instances, using the setup that provided the best results from the memory scalability experiment: 16 Redis instances, each pinned to a physical CPU core. We vary the maximum number of concurrent functions for each separate execution of the five algorithms.

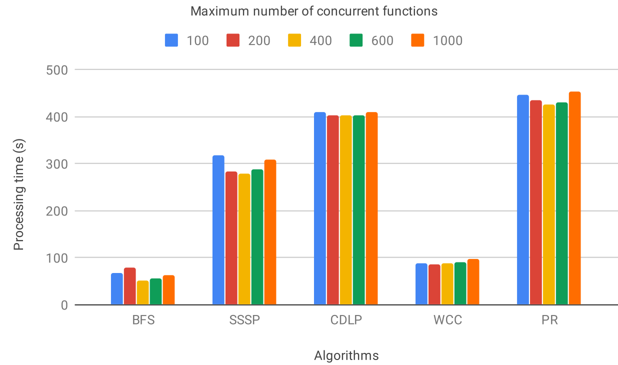

Figure 5. Compute scalability.

The results of the experiment confirm the findings of the memory engine scalability experiment: the system is not compute-bound, but is network-bound. From Figure 5 we determine that the best run used a maximum number of 400 concurrent functions. Further increasing the number of functions leads to increased contention when attempting to access the memory engine, slowing down the entire execution. This is evidenced by the fact that 200 functions perform better on almost all algorithms (excluding BFS) when compared to 600 concurrent functions.

To meet R2 from Section 2, the system needs the ability to seamlessly scale-up or -down with the unpredictable graph workloads. Our experiments show that fine grained elasticity is achieved. The execution patterns of each algorithm are easily identifiable in Figure 6, where we can see the number of workers constantly adjusting to match the workload. A sharp rise in the number of workers marks the start of a new superstep, as the remaining active vertices are partitioned to a fresh set of workers. PR exhibits a similar pattern to CDLP; for both algorithms, all vertices are active during each superstep, therefore utilizing the maximum number of functions. The number of workers drops sharply as the computation finishes for each vertex. For SSSP, it is clearly observable when it approaches convergence, as the worker invocation pattern becomes increasingly smaller.

These worker invocation patterns help explain why al-

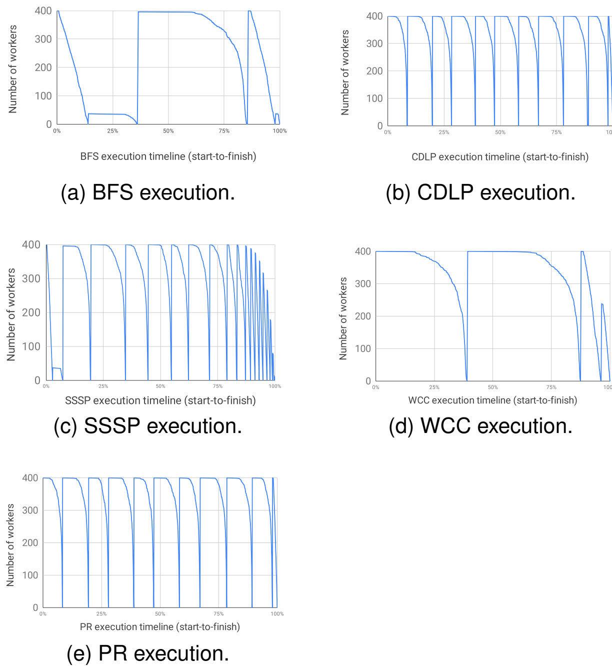

Figure 6. Fine-grained elasticity during execution of algorithms.

gorithms such as PR and CDLP have significantly higher processing times on our system than BFS and WCC. They require a higher number of supersteps to complete and make use of the maximum amount of workers, since every vertex is active during each superstep. The communication overhead is much higher than for BFS and WCC, straining our network-bound system.

4.2. Comparison of the Architectures

In this section we compare the two implemented architectures of the Graphless prototype, one with a push-based model and the other with a pull-based model. We compare running each architecture with the best configuration determined from the previous experiments: 400 concurrent functions and 16 Redis instances.

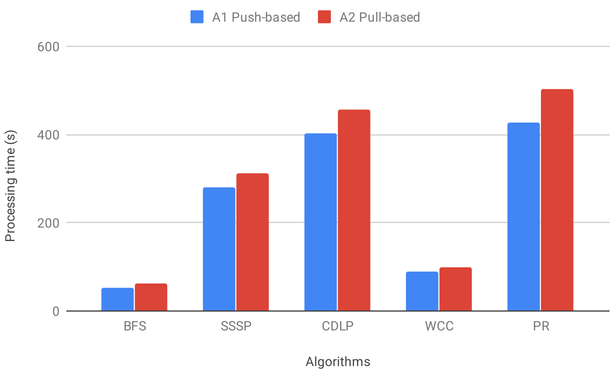

Figure 7. Processing time comparison of a push-based and a pull-based architecture.

For our dataset, the push-based architecture performs better with all five algorithms. The pull-based architecture ends up being slower due to the extra overhead added by inserting and retrieving tasks from the queue service.

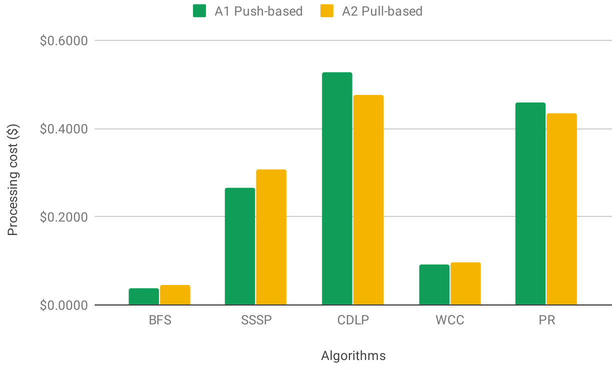

Figure 8. Processing cost comparison of a push-based and a pull-based architecture.

When it comes to cost, the push-based architecture is cheaper for BFS, SSSP, and WCC. Surprisingly, although the pull-based architecture is slower for all algorithms, it is actually more cost-effective for PR and CDLP. While the processing time is higher than the push-based architecture, the aggregated function execution time is lower, therefore the processing cost is also lower. The higher processing time is caused by the orchestrator taking longer to execute; every superstep it has to push tasks in the queue service for all vertices.

4.3. Comparison with Other Graph Processing Systems

In this section we compare Graphless with two other graph processing systems, Apache Giraph and GraphMat. Experiments are run with Giraph and GraphMat deployed on 16 nodes.

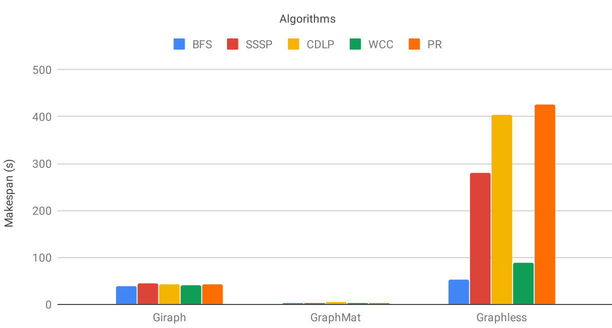

Figure 9. Makespan comparison for three graph processing platforms: Giraph, GraphMat, and Graphless.

The experiments show that Graphless is orders of magnitude slower than the other systems for most algorithms. The best result is 25% slower than Giraph for BFS. Compute power is not a problem with the underlying serverless

platform, which is able to spin up thousands of concurrent workers on demand. As shown in the previous experiments, communication is the issue, taking up the majority of the execution time. This is especially a problem for communication-intensive algorithms, such as PR and CDLP.

Even though GraphMat is much faster than both Giraph and Graphless, it performs extensive preprocessing of the dataset before computation. It was observed that preprocessing time often dominates the total execution time of an algorithm [25] and that processing time is often just a fraction of the makespan [23].

In the case of Graphless, the makespan is equal to the processing time. Unlike other graph processing systems, where partitioning is usually only done once in the beginning of an execution, repartitioning has to be done every superstep due to the statelessness of the workers.

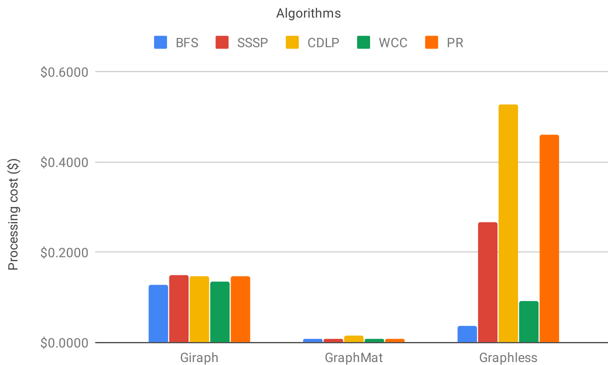

Figure 10. Processing cost comparison for three graph processing platforms: Giraph, GraphMat, and Graphless.

Figure 10 depicts the cost of running all five algorithms with the three graph processing systems. Graphless is an order of magnitude cheaper for BFS than Giraph; it is also cheaper than Giraph for WCC. Graphless has a significantly higher processing cost than Giraph or GraphX when running communication-intensive algorithms, such as PR and CDLP. GraphMat is the cheapest to run for all algorithms when taking makespan into account. However, as previously stated, it also spends time preprocessing the dataset; this would also be billed when running GraphMat on virtual machines in the cloud.

4.4. Discussion

The experiments prove that Graphless can be cost-effective when running algorithms with lighter communication requirements, such as BFS. Provided that the communication bottleneck is overcome, Graphless can be cost-effective for all five algorithms; computation is cheap if we are not waiting on communication and the majority of the cost comes from the memory service. Recently, researchers have started to develop elastic storage systems for serverless data analytics [26]. Using such a system, serverless graph processing systems can benefit from fully automated resource management, therefore reducing cost. Decreasing the number of sent messages between vertices would significantly improve performance and cost for all algorithms. In its current state, the system is not competitive for communication-intensive algorithms when compared to existing graph processing systems, such as Giraph and GraphMat. Network variability is a big issue, since performance will vary greatly between different functions, leading to imbalance. In worst-case scenarios, bandwidth for a Lambda function can drop to as low as 30Mbps [24].

5. Related Work

In this section, we contrast our contribution with related work. Overall, ours is the first work combining graph processing with the serverless paradigm.

Elastic graph processing: Closest to our work, the JoyGraph elastic graph-processing system [18] uses traditional VM-based infrastructure. However, the JoyGraph process of starting-up and stopping VMs adds significant performance and cost overhead, and requires stopping computation and repartitioning data. In contrast, Graphless offers much finer-grained scalability, due to serverless computing, and provides a memory-as-a-service component for data persistence.

Serverless processing: Also close to our work are the pioneering studies of serverless computing applied to processing in general, and in particular to big data. PyWren [27] is a serverless (big) data processing system, which can implement MapReduce-like jobs. However, PyWren offers no support for graph processing: it cannot support iterative computation and loading the Python runtime is too slow for short-lived functions. Similarly, Flint [28] and a streaming approach [29] are serverless data-processing platforms with useful features, but do not support graph processing.

Performance for serverless storage and communication: This serverless performance bottleneck is starting to be addressed by the community. Pocket [26] is an elastic serverless storage system built on the premise that using cloud-storage services as remote memory for serverless data analytics cannot provide sufficient performance. Similarly, [30] analyses the suitability of using serverless computing for network intensive applications.

6. Conclusion and Future Work

To make available graph processing to a wider audience than the current specialized tools do, in this work we have focused on leveraging the emerging serverless paradigm for graph processing.

We have proposed Graphless, the first serverless graph-processing system. Graphless proposes a set of backend components that automate the deployment and operation of graph processing, and an architectural approach that supports both push and pull operations. Among the key challenges, Graphless addresses the need to reconcile the assumption of stateless functions that serverless makes with the need to access and store data that is typical of graph

processing. Graphless achieves this through a dedicated memory-as-a-service component. Overall, the serverless nature of the Graphless system ensures automated resource management, fine-grained scalability, and, when deployed on FaaS cloud platforms such as Amazon Lambda, fine-grained pay-per-use billing model.

We prototype Graphless and evaluate it through real-world experiments using LDBC Graphalytics. Our results show that the system is not compute-bound, but network-bound, with the majority of the execution time being spent on communication. When compared to big data platforms, such as Apache Giraph, the system is similar in performance and cost-effective for algorithms that can benefit from fine-grained elasticity. Expectedly, Graphless is not competitive with HPC platforms, such as GraphMat.

For the future, we plan to continue our architectural work through exploration with more diverse architectures. We will also focus on performance engineering for individual components, such as graph-processing-aware elastic storage and communication—these are key to making Graphless competitive also with HPC platforms.

References

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, “One trillion edges: Graph processing at facebook-scale,” *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1804–1815, 2015.

[2] G. A. Pavlopoulos, M. Secrier, C. N. Moschopoulos, T. G. Soldatos, S. Kossida, J. Aerts, R. Schneider, and P. G. Bagos, “Using graph theory to analyze biological networks,” in *BioData Mining*, 2010.

[3] G. Wang, S. Xie, B. Liu, and P. S. Yu, “Review graph based online store review spammer detection,” in *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, 2011, pp. 1242–1247.

[4] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, “Graphmat: High performance graph analytics made productive,” *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, 2015.

[5] “Apache Giraph,” http://www.giraph.apache.org, last accessed: 5 March 2018.

[6] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, “Graphx: A resilient distributed graph system on spark,” in *First International Workshop on Graph Data Management Experiences and Systems*, 2013, pp. 2:1–2:6.

[7] E. van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, “A spec rg cloud group’s vision on the performance challenges of faas cloud architectures,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 21–24.

[8] L. Zhu, A. Galstyan, J. Cheng, and K. Lerman, “Tripartite graph clustering for dynamic sentiment analysis on social media,” in *SIGMOD*, 2014, pp. 1531–1542.

[9] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich, “A review of relational machine learning for knowledge graphs,” *Proceedings of the IEEE*, vol. 104, no. 1, pp. 11–33, 2016.

[10] X. Yan and J. Han, “gspan: graph-based substructure pattern mining,” in *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, 2002, pp. 721–724.

[11] K. M. Alzoubi, P.-J. Wan, and O. Frieder, “Message-optimal connected dominating sets in mobile ad hoc networks,” in *Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*, 2002, pp. 157–164.

[12] B. An and S. Papavassiliou, “A mobility-based clustering approach to support mobility management and multicast routing in mobile ad-hoc wireless networks,” *International Journal of Network Management*, vol. 11, no. 6, pp. 387–395, 2001.

[13] N. Wang, D. Li, and Q. Wang, “Visibility graph analysis on quarterly macroeconomic series of china based on complex network theory,” vol. 391, p. 65436555, 2012.

[14] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, “Challenges in parallel graph processing.” vol. 17, pp. 5–20, 2007.

[15] N. Doekemeijer and A. L. Varbanescu, “A survey of parallel graph processing frameworks,” *Delft University of Technology*, p. 21, 2014.

[16] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, “Scalable graph processing frameworks: A taxonomy and open challenges,” *ACM Comput. Surv.*, vol. 51, no. 3, pp. 60:1–60:53, 2018.

[17] A. Iosup, A. Uta, L. Versluis, G. Andreadis, E. van Eyk, T. Hegeman, S. Talluri, V. van Beek, and L. Toader, “Massivizing computer systems: A vision to understand, design, and engineer computer ecosystems through and beyond modern distributed systems,” in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, July 2018, pp. 1224–1237.

[18] A. Uta, S. Au, A. Ilyushkin, and A. Iosup, “Elasticity in graph analytics? a benchmarking framework for elastic graph processing,” in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 381–391.

[19] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, “The ubiquity of large graphs and surprising challenges of graph processing,” *Proc. VLDB Endow.*, vol. 11, no. 4, pp. 420–431, 2017.

[20] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A system for large-scale graph processing,” in *SIGMOD*, 2010, pp. 135–146.

[21] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[22] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, “Understanding ephemeral storage for serverless analytics,” in *ATC*, 2018, pp. 789–794.

[23] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardto, H. Chafio, M. Capotă, N. Sundaram, M. Anderson, I. G. Tănase, Y. Xia, L. Nai, and P. Boncz, “Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms,” *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1317–1328, 2016.

[24] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking behind the curtains of serverless platforms,” in *ATC*, 2018, pp. 133–146.

[25] T. David, R. Guerraoui, and V. Trigonakis, “Everything you always wanted to know about synchronization but were afraid to ask,” in *SOSP*, 2013, pp. 33–48.

[26] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic ephemeral storage for serverless analytics,” in *OSDI*, 2018, pp. 427–444.

[27] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, “Occupy the Cloud: Distributed Computing for the 99,” in *SoCC*, 2017, pp. 445–451.

[28] Y. Kim and J. Lin, “Serverless data analytics with flint,” *CoRR*, vol. abs/1803.06354, 2018.

[29] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, “A serverless real-time data analytics platform for edge computing,” *IC*, vol. 21, no. 4, pp. 64–71, 2017.

[30] A. Singhvi, S. Banerjee, Y. Harchol, A. Akella, M. Peek, and P. Rydin, “Granular computing and network intensive applications: Friends or foes?” in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017, pp. 157–163.