Wukong: A Fast, Cost-Effective, and Easy-to-Use Serverless DAG Engine

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Benjamin Carver
Bachelor of Science
George Mason University, 2020

Director: Dr. Yue Cheng, Assistant Professor
Department of Computer Science

Spring Semester 2021
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my family.

Acknowledgments

First, I would like to thank my advisor, Dr. Yue Cheng. We began working together in the Spring of 2019 when I was still an undergrad. Dr. Cheng has been an incredible mentor; he has always remained extremely supportive and has already taught me so much. I look forward to continuing to work with him as I pursue my PhD. Additionally, I would like to express my appreciation to my committee members: Dr. Elizabeth White and Dr. Songqing Chen for their helpful comments and feedback on my thesis. I also owe Dr. White another huge "thank you" for all her support and help as my undergraduate academic advisor.

I would also like to thank my collaborators at Mason LEAP Lab: Jingyuan Zhang and Ao Wang. We've worked together on several projects, and your feedback, insight, and support has been extremely helpful. To that end, I would also like to thank Dr. Panruo Wu at the University of Houston, with whom I collaborated on WUKONG, for his insight and expertise. Finally, I would like to thank Ali Anwar, another collaborator of mine. His comments and feedback have been tremendously helpful when writing papers and preparing talks and presentation of my work.

Next, I would like to thank all of the friends who have accompanied me throughout my academic journey thus far. First of all, I consider myself immensely fortunate to have been randomly placed in the same room as Eric Terry my freshman year. Many of my most treasured memories from undergrad involve him, and his unyielding support helped me through many long days and nights of studying. I genuinely could not have asked for a better roommate. Thanks for everything, man. I would also like to extend my gratitude to several of my closest friends for their unwavering love, support, and companionship. First, Lansana Koroma and Ozod Kurbonov – you two are like brothers to me. Similarly, Yousif Alousi, Marcus deVos, Ian Trefen, Matthew Haley, Garrett Spencer, and Debi Das – y'all are some of the best friends a man could ask for. Thank you for always being there. And for all the fun, of course!

In closing, I would like to thank my family. To my parents, your unrelenting love and support has been invaluable. I am eternally grateful for all that you have done for me. I could not be where I am today without you both. I would also like to thank my grandmother, Elaine Carver, and my grandfather, Richard Ray Carver, for the same reasons. Next, my cousins, Alex and Kelsey Herbst – you two are like the brother and sister I never had. And to Jill and James, who have always been there cheering me on. The same goes for the California folks – Uncle Charlie, Aunt Jane, and my two other cousins, Sam and Addie. Finally, I would like to thank my girlfriend, Alison Gomeiz, who has been my rock for the past five years. You are a constant source of inspiration and happiness. Thank you for everything.

Table of Contents

List of Tables

Table Page

List of Figures

Abstract

WUKONG: A FAST, COST-EFFECTIVE, AND EASY-TO-USE SERVERLESS DAG ENGINE

Benjamin Carver

George Mason University, 2021

Thesis Director: Dr. Yue Cheng

Data analytics applications can often be modeled as a directed acyclic graph (DAG), where the nodes are fine-grained tasks and the edges are task dependencies. A DAG scheduler can be used to distribute the tasks to cloud computing resources where they can be executed in parallel to speedup workflow applications. Serverless computing is a cloud computing platform that enables the decomposition of traditionally monolithic, server-based applications into a collection of fine-grained cloud functions. Developers write the function logic while the service provider takes care of provisioning, scaling, and managing the back-end servers or virtual machines (VMs) that the functions run on. Creating a serverless-oriented DAG scheduler poses a major challenge, as executing complex, burst-parallel DAG jobs requires rapid scaling and high task throughput while minimizing data movement across tasks.

Despite these challenges, data analytics workloads are well-suited for serverless computing. The auto-scaling property of serverless computing platforms accommodates short tasks and bursty workloads, while the pay-per-use billing model of serverless computing providers keeps the cost of short tasks low.

In this thesis, we thoroughly investigate the problem space of DAG scheduling in serverless computing. Our goal is to demonstrate that serverless-oriented, parallel computing frameworks can support fast and efficient, DAG-based, parallel-computation workflows that are easy to deploy and manage. To accomplish this, we identify and evaluate a set of techniques to make DAG schedulers serverless-aware, and we implement these techniques in Wukong, a serverless DAG engine built atop AWS Lambda.

Our techniques and optimizations bring multiple benefits, including enhanced data locality, reduced network I/Os, automatic resource elasticity, and improved cost effectiveness. We show that when comparing Wukong to numpywren, a serverless system for linear algebra, Wukong achieves near-ideal scalability, executes parallel computation jobs up to 68.17× faster, reduces network I/O by multiple orders of magnitude, and achieves 92.96% tenant-side cost savings compared to numpywren, a serverless linear algebra library.

This thesis contains two modified, published papers along with an additional chapter describing the latest, work-in-progress version of Wukong. In Chapter 3, we describe and evaluate the initial prototype of Wukong. This first version of Wukong delivered competitive performance compared to a comparable, serverful Dask cluster. In Chapter 4, we present the second version of Wukong. We present a series of optimizations that greatly improve the cost-effectiveness and performance of Wukong. Finally, we present the current work-in-progress version of Wukong in Chapter 5. The latest version of Wukong is completely serverless, requiring no user-deployed or user-managed servers. As a result, this version of Wukong is extremely simple to use while still delivering competitive performance and cost-effectiveness.

This thesis establishes that serverless computing is an appropriate setting for creating a serverless DAG engine. We show that, by designing a DAG engine that takes into account both the benefits and the challenges of serverless computing platforms, it is possible to create a fast, cost-effective, and easy-to-use serverless parallel computing framework.

Chapter 1: Introduction

Serverless computing is an emerging cloud-computing model that enables a new way of building and scaling applications and services. A monolithic application that continuously runs on a server can be divided into a collection of fine-grained functions (also known as "Functions-as-a-Service (FaaS)"). These functions run only on demand, as a response to some event or by direct invocation from another function. Developers focus on writing the business logic of the functions in languages like Python, Go, or Java. The functions still run on backend servers and virtual machines (VMs); however, the cloud provider is responsible for provisioning and scaling resources, instantiating VMs, and performing most of the resource management. Thus, to the developers, the application appears to be "serverless."

Each serverless function has RAM and ephemeral disk space that it uses to store data objects. However, function executions are "stateless" – when a function completes, the data that it stored in RAM or on disk is not guaranteed to be available on the next invocation of the function. Function data that absolutely must be persistent must be stored externally in a database (e.g., Amazon Web Services (AWS) DynamoDB), cache (e.g., Redis), or object store (e.g., AWS S3). There is also a limit on how long a function can execute. On AWS Lambda, the current limit is 15 seconds.

A typical serverless application is backend data processing. For example, data objects in Amazon S3, a popular, cloud-based, object-storage service offered by AWS, can be processed using AWS Lambda, Amazon's serverless platform. Uploading an image file to S3 can trigger a Lambda function that will crop or resize the image. Thousands of images can be processed in parallel. AWS Lambda will dynamically scale the compute resources that are needed to execute the triggered functions.

Recently, researchers have developed more complex serverless applications. One such example is DAG (Directed Acyclic Graph)-based workflows. DAG-based workflows are

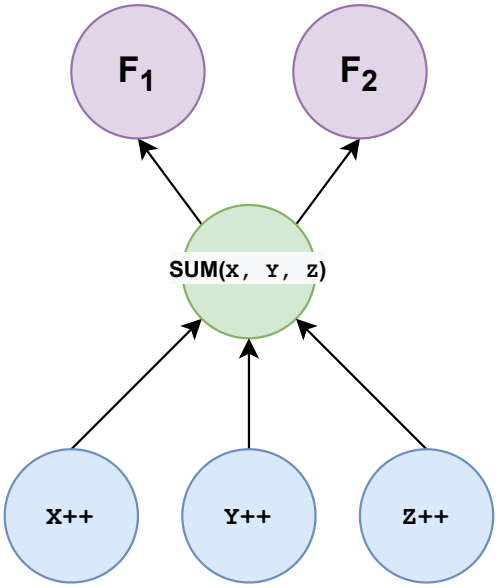


Figure 1.1: A simple DAG workload.

characterized by short, fine-grained, parallel tasks. They are often interactive jobs, which require very short response times. Specific examples of DAG workflows include data analytics, optimization algorithms, and real-time machine learning classifications, such as support vector machines (SVM). Due to the large number of short-running, potentially parallel tasks in these workflows, scheduling decisions must be made very frequently and scheduling delays cannot be tolerated.

Consider the DAG shown in 1.1. This DAG models a simple workload that takes as input three variables X, Y, and Z. In this workload, each variable is incremented by one. After this, the sum of the variables is computed and used as input to two fan-out tasks $F1$ and $F2$. The DAG captures the dependencies in the workload by using nodes to represent individual operations, or tasks, and directed edges to represent data dependencies between the tasks. The fan-in node representing the SUM operation depends on the output of each of the increment operations. Fan-out nodes $F1$ and $F2$ can be executed in parallel as can the increments of the three variables.

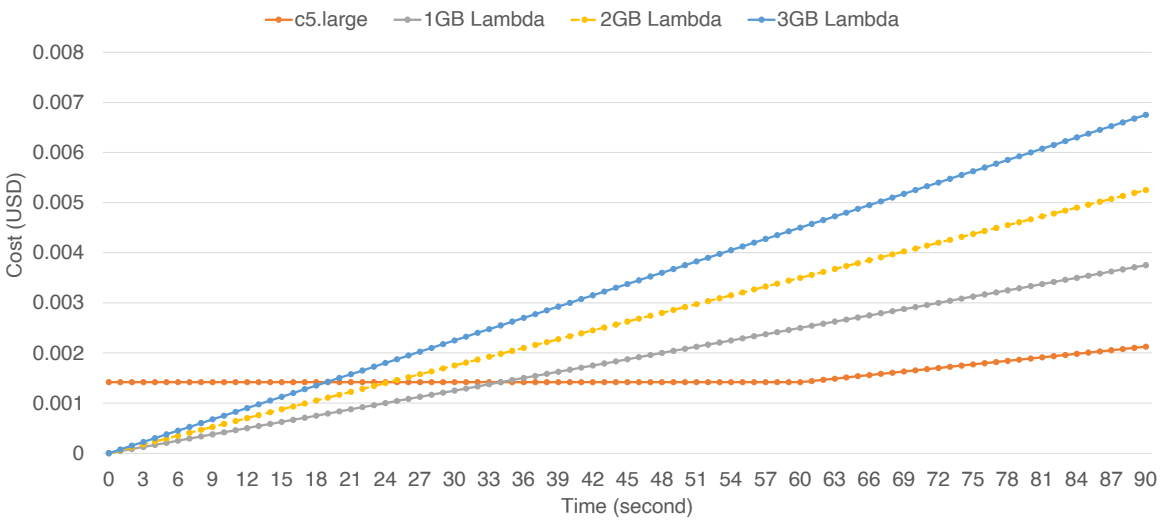


Figure 1.2: Cost comparison between a `c5.large` On-Demand EC2 instance and three different AWS Lambda configurations. This comparison assumes 125 Lambda functions are invoked each second for the entire duration of the workload.

Serverless platforms have several advantages that make them well-suited for DAG workflows. First, the large-scale parallelism and auto-scaling properties of serverless computing platforms give them a big advantage over continuously running servers for DAG workloads that are highly-variable or interactive. To deal with highly variable workloads, continuously running servers may be over-provisioned and thus spend more time idling than doing useful work. Serverless functions, on the other hand, can stop when there is no work to do and will incur a cost only when they are running. In order to satisfy the short response times of interactive workloads, serverless functions can be quickly scaled up to thousands of cores and scaled down when the workload finishes. Continuously running servers will either waste time idling between interactive workloads, or they will need to be started each time a workload is submitted, which wastes time waiting for the servers to startup. Additionally, serverless platform providers use a fine-grained, pay-per-use pricing model. For example, AWS Lambda charges users based on the number of function invocations ($0.20 per 1 million function invocations) they make and the time it takes for their functions to execute. (Function executions are rounding up to the nearest 1 millisecond).

Second, DAG workflows with short tasks can take advantage of the serverless platform's pricing model in order to keep costs low. 1.2 presents a cost-comparison between a continuously running `c5.large` EC2 instance using the AWS On-Demand pricing model and an AWS Lambda function configured to use 1, 2, or 3GB of memory. EC2 charges for execution time using per-second granularity with a sixty second minimum. Because of this, AWS Lambda can be considerably cheaper for short-running tasks, depending upon the number of invocations. Consequently, serverless computing is a promising platform for next-generation, large-scale DAG workloads in data analytics, data science, and high-performance computing (HPC).

Thesis Statement: Serverless-oriented, parallel computing frameworks can support fast and cost-effective, DAG-based, parallel-computation workflows that are easy to deploy and manage.

In order to realize this thesis, there are three high-level problems that serverless DAG-execution frameworks must address. The first problem is how to schedule the tasks in a DAG workload. A common approach to scheduling tasks is to use a single centralized scheduler that is responsible for scheduling all of the tasks in the DAG; however, a centralized task scheduler can quickly become a performance bottleneck, especially for large jobs with thousands of tasks.

We have developed an alternative approach called decentralized scheduling. Decentralized scheduling partitions the tasks in a DAG and assigns the tasks in each partition to a separate Lambda function. Each of the Lambda functions schedules its assigned tasks for execution, by itself, or by coordinating with other Lambda functions. Decentralized scheduling leverages the elasticity of serverless platforms and also the pay-per-use pricing model of Lambda functions. Decentralized scheduling is implemented in the first version of our DAG-execution framework, called Wukong 1.0. Wukong 1.0 is described in the paper "In Search of a Fast and Efficient Serverless DAG Engine" [2] presented in Chapter 3.

The second problem associated with building a DAG-execution framework is determining

the amount of task parallelism and the form of task communication that should be used to execute a DAG workload. Two tasks can be executed in-parallel if they do not depend upon one another. A DAG naturally expresses opportunities for parallelism at its fan-out points: fan-out tasks are independent and can be executed in parallel (e.g., tasks F1 and F2 in Fig. 1). However, parallel fan-out tasks must communicate their intermediate results to the succeeding fan-in tasks, which may result in an overall decrease in performance if communication is too costly. To address this problem, we developed a runtime technique for determining when a reduction in parallelism, and its associated overhead for communicating intermediate results, will increase performance. In addition, we coupled this technique with serverless, high-performance cloud storage that reduces communication delays at fan-in and fan-out points. These improvements are incorporated in WUKONG 2.0, which is described in Chapter 4 in our second paper, WUKONG*: A High-Performance Framework for Serverless Parallel Computing* [3].

The third problem associated with building a DAG-execution framework is making the framework easy to use. Existing serverless frameworks such as PyWren, numpywren, and the first two versions of our DAG-execution framework WUKONG 1.0 and WUKONG 2.0, all rely on multiple servers in some capacity. As a result, end-users are responsible for deploying and managing one or more servers, which often includes complicated container-orchestration systems such as Kubernetes.

In order to make our framework easier to use, we replaced the serverful code components of WUKONG 2.0, including the intermediate cloud storage, with serverless components that are simple to deploy and that require no server management. The final version of our framework, WUKONG 3.0, is completely serverless – WUKONG 3.0 does not require user-managed cloud servers, and WUKONG 3.0 can be run transparently from the user's personal desktop. We present the design and evaluation of WUKONG 3.0 in Chapter 5.

Chapter 2: Background and Related Work

2.1 Serverless Computing Primer

Serverless Computing handles virtually all the system administration operations to make it easier for users and developers to use a near-infinite amount of cloud resources including bundled CPUs and memory, object stores, and a lot more [4]. Developers structure serverless cloud applications as a collection of fine-grained cloud functions. Service providers provide a flexible interface for defining functions so that developers can completely focus on developing the core application logic; service providers in turn help automatically scale the function executions in a demand-driven fashion, hiding the tedious cluster configuration and management overheads from the users.

General Constraints and Limitations

Service providers place limits on the use of cloud resources in order to simplify resource management. For example, AWS Lambda users have to configure Lambda's memory and CPU resources in a bundle, rather than configuring them separately. The idea is that users lose some flexibility in selecting resources in order to make it easier for AWS to manage resources. Users can choose a memory amount between 128MB and 10,240MB. Lambda allocates CPU power linearly in proportion to the amount of memory configured. Each Lambda function can run at most 900 seconds and is forcibly returned when the function timeouts. In addition, Lambda only allows outbound network connections.

In addition to these limitations, serverless computing suffers from a "cold start" penalty [5–7][1] associated with container startups, though this penalty has been reduced over time. Service providers rely on container caching (i.e., warmed functions) to mitigate the impact

[1] "Cold start" refers to the first-ever invocation of a function instance.

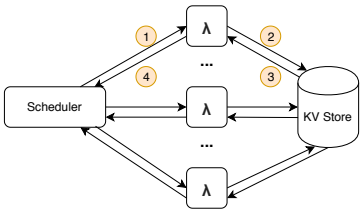


Figure 2.1: The strawman scheduler architecture. (1) The scheduler invokes a Lambda function, which establishes a TCP connection with the scheduler, executes the task, and then (2) sends the output results to the KV store. Once the Lambda function receives an `ACK` from the KV store (3), it notifies the scheduler (4) that the task is finished.

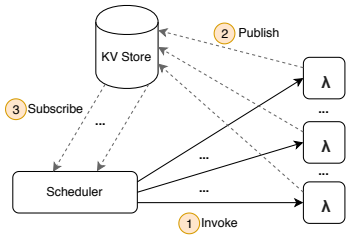


Figure 2.2: The pub/sub architecture. (1) The scheduler invokes a Lambda function, which executes the task, and then publishes the output results to the KV store in (2). The scheduler as the subscriber listens for messages on predefined channels, and gets notified in (3) whenever a Lambda function publishes the results to the KV store.

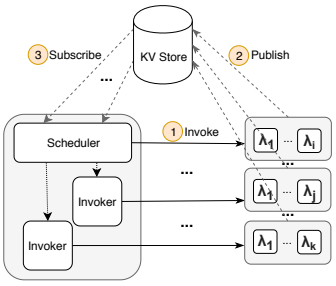


Figure 2.3: The parallel-invoker architecture. This architecture extends the pub/sub architecture in Figure 2.2 with a parallel-invoker that accelerates Lambda invocations by spawning multiple invoker processes in the scheduler to concurrently invoke Lambda functions.

of cold starts on elasticity. Another limitation that plagues the runtime performance of serverless applications is the lack of a quality-of-service (QoS) control. As a result, functions suffer from straggler issues [8]. *Therefore, an ideal serverless DAG framework must effectively workaround these limitations.*

Opportunities

Running DAG parallel jobs (e.g., distributed linear algebra, distributed data analytics, etc.) has long been challenging for domain scientists and data analysts due to a lack of access to high-performance computing clusters and the complexity of configuring, provisioning, and managing clusters. The emerging serverless computing model provides access to large computing clusters while relieving domain scientists and data analysts of tedious cluster administration tasks. *However, bridging the gap between serverless systems and DAG users requires friendly, fast, and efficient, DAG- and serverless-aware, frameworks.*

2.2 Serverless Workflow Management Frameworks

Existing serverless frameworks have been built using two main approaches. The first is a queue-based, master-worker approach in which the master orchestrates the workflow and submits tasks that are ready for execution to a queue. Workers are cloud functions that process these tasks in parallel when possible. For example, numpywren [9] is a serverless linear algebra framework. numpywren workers are implemented using PyWren [10], which is a framework for executing data-intensive batch processing workloads. In the second approach, the master directly invokes cloud functions to process ready tasks [11,12]. Examples of this include Sprocket [13] and ExCamera [14], which have been developed for serverless video processing.

This second approach is also used by general purpose serverless orchestration frameworks. Example frameworks include AWS Step Functions, Azure Durable Functions, Fission Workflows [15], and the framework in [16]. But these frameworks are not well-suited for supporting large, complex jobs, because they require manual workflow configurations (e.g., JSON) [17]. Lòpez et al. [17] evaluated AWS Step Functions and Azure Durable Functions with respect to their support for parallel execution, among other attributes. They found that the overhead for parallelism grows exponentially with the number of parallel functions for AWS Step Functions and Azure Durable Functions.

Fission Workflows. Fission Workflows is built on top of the Fission [18] serverless framework for Kubernetes. Users define a DAG by creating a configuration file, which defines the tasks and their dependencies. The framework in [16] is built upon the HyperFlow [11] workflow engine. HyperFlow models its workflows using user-written JSON files, and thus is similar to Fission Workflows with respect to how workflows are represented. While manually composing a DAG configuration may work well for coarse-grained microservice-based workflow applications, manually implementing a complex, fine-grained workflow is nontrivial. For this reason, [15, 16] are not well-suited for supporting complex computing jobs implemented using high-level programming languages.

WUKONG uses neither a master-worker-queue approach nor a direct invocation approach. Instead, WUKONG adopts a decentralized approach in which the global DAG is partitioned into local subgraphs. Each WUKONG executor is responsible for scheduling and executing the tasks within its assigned subgraph in an autonomous manner. An executor assumes the master's role when it uses its assigned subgraph to determine when its tasks can be executed; an executor assumes a worker's role when it executes these tasks. WUKONG executors coordinate to ensure that the dependencies in the global DAG are satisfied.

Dask. Dask is an open-source parallel computing library for Python data analytics [10]. Standard Dask supports single-machine multi-threading and multiprocessing while Dask Distributed [6] provides support for distributed data analytics in Python.

Dask Distributed is a centrally-managed task scheduler in which a central scheduler manages a number of different workers. The scheduler is capable of handling requests from multiple users as it coordinates workloads across the worker processes. The workloads are internally stored as a DAG, which Dask itself generates based on the operations submitted by the users [6].

The Dask Distributed documentation [11] describes the criteria used by the scheduler to select a worker for a given task. For a task with no significant data dependencies and no restrictions (e.g. requiring a GPU), the scheduler will simply select the least-occupied worker. If the given task has restrictions, such as requiring a GPU, then the scheduler will narrow its search down to a group of workers that satisfy the given constraints. From here, the scheduler will favor workers that exhibit the largest amount of data locality with respect to the given task's data dependencies. In the case of a tie, the worker with the fewest tasks present (both in memory and tasks currently processing on the worker) will be selected.

The Dask Distributed documentation also describes the criteria used to select a task for scheduling and execution [11]. The Scheduler uses a number of heuristics to optimize various objectives, namely:

- service tasks in a first-in first-out fashion in order to make scheduling fast and simple
- execute tasks on the critical path first to reduce total runtime

- execute tasks that will enable the release of multiple data dependencies first, to free up memory resources
- execute related tasks to eliminate large "chunks" of work.

There are a few additional heuristics described. When a worker processes finishes execution of a task, the immediate dependents of that task receive top priority. This is done in the interest of eliminating "chunks" of related work before beginning new "chunks". In the case of a tie between selecting tasks, tasks with longer critical paths and a greater number of dependent tasks are favored. These designs are made at various points during executing, such as when a user submits a job to the scheduler [11].

Researchers have identified new stateful parallel applications for serverless computing. These efforts have lead to serverless parallel computing frameworks [8–14,19–22], which have been built using methods 2 and 3 described in section 2.2. However, none of them explicitly addresses the data locality issue of stateful serverless parallel applications. WUKONG goes beyond existing work with a novel *locality-aware* decentralized scheduling approach for complex DAG jobs.

funcX. funcX [23] is an open FaaS platform that enables high performance "serverless supercomputing" over existing HPC infrastructures. WUKONG is orthogonal to funcX and should be portable to existing commercial and open-source FaaS platforms [18, 24–27]. Porting WUKONG to other serverless platforms is part of our future work.

SAND. SAND [7] is a serverless platform that increases data locality by running some or all of the functions/tasks for a given workload within the same container on the same server, but as separate processes. WUKONG is a serverless application framework that increases data locality by executing multiple dependent tasks in the same WUKONG Executor function, and hence in the same process, container, and server. Thus, SAND improves data locality in the serverless platform layer, while WUKONG improves locality in the application framework layer running atop the serverless platform.

INFINICACHE [28] is a distributed memory cache that exploits the memory of serverless functions for object caching. WUKONG complements INFINICACHE in that WUKONG can

use InfiniCache to cache intermediate data.

ExCamera. ExCamera [14] is a video processing system designed to provide low-latency video processing using serverless computing. The system has two main components: a framework designed to execute computations on a commercial FaaS platform and a video encoder designed for fine-grained parallelism.

The authors of ExCamera identified five issues with AWS Lambda that made utilizing Lambda for their video processing system difficult. These issues include the long install-times for Lambda functions (especially when compared to invocation times), the inconsistency of Lambda start-times due to "cold starts", the Amazon-imposed concurrency limit for Lambda function execution, the fact that Lambda functions themselves cannot be addressed directly, and the limited execution time of AWS functions. In order to address these issues, the authors of ExCamera developed "mu", a library for orchestrating large-scale parallel computations on AWS Lambda. This library has three main components, which together help to mitigate the aforementioned problems.

First, mu uses a centralized coordinator, which is a long-lived process that is responsible for managing ExCamera workers. The coordinator uses RPC's and dependency-aware scheduling to interact with workers and manage tasks that require data from other tasks, respectively. These RPC's can direct workers to store or retrieve data from an external data store, initiate connections with other workers (via an intermediate server), and perform computations, among other things.

The workers themselves are invocations of the same AWS Lambda stateless function. At the beginning of their execution, workers will establish a connection to the coordinator so they may receive RPCs. Finally, in order to facilitate inter-worker communication, ExCamera makes use of a so-called "rendezvous server". The rendezvous server is a long-lived process that directs messages between workers.

Pywren Pywren [10] is a serverless data analytics platform that enables users to execute existing Python code at a large scale via AWS Lambda. Developed to address problems with modern cloud computing abstractions, Pywren leverages a stateless architecture with

stateless functions to create a massively parallel data-processing framework.

Pywren's main components include an execution runtime, scheduler and remote storage. A function is submitted to the scheduler, which provisions a container in which the function can be executed. The function's code and required data are serialized and stored in S3, where they are later retrieved by the serverless function responsible for executing the code. Once execution is finished, the result is stored in S3 at a predetermined key. The storage of the result in S3 signals that the job has finished executing.

A large body of research has explored distributed scheduling [29–33]. However, these solutions all target serverful scheduling with serverful deployment-specific optimization objectives. WUKONG targets serverless computing and takes advantage of the massive parallelism of serverless computing for high efficiency.

Parsl Parsl [33] is a "parallel scripting library" that augments Python code in order to directly encode parallelism. Users annotate Python functions, and Parsl uses these annotations to generate a task graph, which is internally represented as a DAG. The nodes of this DAG represent tasks while the edges represent dependencies between tasks, much like the DAG used by Dask. Parsl leverages this internal DAG representation to provide immediate task execution (i.e., as soon as a task is identified in the graph, it may be executed) and the ability for existing tasks to create new tasks dynamically during execution.

The main components of Parsl include the DataFlowKernel (DFK), Providers, and Executors. DFK acts as a central scheduler and is responsible for generating and managing the execution of tasks. DFK schedules a given task for execution once all of the task's dependencies have been resolved.

In order to accommodate the variety of different cloud providers, Parsl implements a simple uniform interface known as a provider. Providers act as an abstraction for submitting jobs, retrieving the status of jobs, and cancelling jobs. There are a number of different providers implemented by Parsl, including providers for local execution, various cloud providers, and providers for clusters and super computers.

Parsl executors extend Python's Executor class as defined in the *concurrent.futures*

library. Executors are responsible for executing jobs on compute resources (e.g. supercomputers, multi-core workstations, etc.). There are three types of executors:

- High Throughput Executor (HTEX)
- Extreme Scale Executor (EXEC)
- Low Latency Exector (LLEX)

Each executor is designed for a specific set of circumstances, though they all use the same common execution kernel. The kernel is responsible for the deserialization and execution of tasks. HTEX's are designed to execute tasks with a high throughput using a "pilot job model". They are capable of handling multi-day workloads and providing a high degree of fault tolerance. EXEX's are designed to utilize thousands of CPU cores and efficient network infrastructures associated with large supercomputers. Finally, LLEX's are light-weight executors designed for workloads that require low latency functions. Low Latency Executors' are not designed to provide high throughput or fault tolerance; instead, LLEX's minimize the round-trip latency for tasks.

Chapter 3: Wukong 1.0 - A Prototypical Serverless DAG Engine

The following chapter is comprised of the first paper published on WUKONG, *In Search of a Fast and Decentralized Serverless DAG Engine*. This paper appeared in the proceedings of International Parallel Data Systems Workshop (PDSW) 2019. There have been several modifications made to the paper. First, the background and related work sections have been removed. The content of those sections has been included in Section 2. Finally, the conclusion section has been renamed to *Chapter Summary*.

3.1 Abstract

PPython-written data analytics applications can be modeled as and compiled into a directed acyclic graph (DAG) based workflow, where the nodes are fine-grained tasks and the edges are task dependencies. Such analytics workflow jobs are increasingly characterized by short, fine-grained tasks with large fan-outs. These characteristics make them well-suited for a new cloud computing model called serverless computing or Function-as-a-Service (FaaS), which has become prevalent in recent years. The auto-scaling property of serverless computing platforms accommodates short tasks and bursty workloads, while the pay-per-use billing model of serverless computing providers keeps the cost of short tasks low.

In this paper, we thoroughly investigate the problem space of DAG scheduling in serverless computing. We identify and evaluate a set of techniques to make DAG schedulers serverless-aware. These techniques have been implemented in Wukong, a serverless, DAG scheduler attuned to AWS Lambda. Wukong provides decentralized scheduling through a combination of static and dynamic scheduling. We present the results of an empirical study in which Wukong is applied to a range of microbenchmark and real-world DAG applications. Results demonstrate the efficacy of Wukong in minimizing the performance overhead introduced by AWS Lambda — Wukong achieves competitive performance compared to a serverful DAG scheduler, while improving the performance of real-world DAG jobs by as much as 3.1× at larger scale.

3.2 Introduction

In recent years, a new cloud computing model called serverless computing [34,35] (Function-as-a-Service or FaaS)[1] has become prevalent, owing to OS-level (i.e., container-based) virtualization. Serverless computing enables a new way of building and scaling applications and services by allowing developers to break traditionally monolithic server-based applications into finer-grained cloud functions[2]; developers can thus focus on developing function logic without having to worry about provisioning, scaling, and managing traditional backend servers or VMs, which are notoriously tedious to maintain [36].

With their growth in popularity, serverless computing solutions have found their way into both commercial clouds (e.g., AWS Lambda, Google Cloud Functions, and IBM Cloud Functions) and open source projects (e.g., OpenLambda [37]). While serverless platforms were originally intended for event-driven, stateless applications [38], recent trend has demonstrated the usage of serverless computing in support of more complex applications.

One such example is DAG (directed acyclic graph) workflow-based data analytics applications. These applications are characterized by short, fine-grained tasks with large fan-outs [30,39,40]. For example, an analysis of Alibaba workload traces shows that more than 50% of the analytics batch jobs, tasks, and instances are finished within 10 seconds [40,41]. The auto-scaling property of serverless platforms makes these platforms well-suited for the short, fine-grained tasks and bursty, large fan-outs that characterize DAG-based workflows. In addition, FaaS providers charge users at a fine granularity – AWS Lambda bills on a per-invocation basis ($0.02 per 1 million invocations) and charges resource usage by rounding up the function's execution time to the nearest 100 milliseconds (ms). Workloads with short tasks can take advantage of this fine-grained pay-as-you-go pricing model to keep monetary costs low[3]. Consequently, serverless computing can be leveraged as a promising solution for next-generation large-scale DAG workloads in high-performance computing (HPC), data

[1] We use the term "serverless computing" and "FaaS" interchangeably.

[2] Without loss of generality, we use "Lambda functions" to represent cloud functions throughout.

[3] Pay-as-you-go in the context of serverless computing is essentially not paying for what are not being used.

analytics, and data sciences.

Moving DAG scheduling from a traditional serverful deployment to the emerging serverless platforms presents unique opportunities. In a traditional serverful deployment, the best practice is to utilize a logically centralized scheduler for managing task assignments and resource allocation under various objectives including load balancing, cluster utilization, fairness and so on. State-of-the-art serverful workflow schedulers include but are not limited to: MapReduce job scheduler [42], Apache Spark scheduler [43], Sparrow [30], and Dask [44]. In the context of serverless computing, however, the assumptions of the traditional serverful schedulers do not hold any more. This is because: (1) FaaS providers are responsible for managing the "servers" (i.e., where the task executors are hosted); and (2) serverless platforms typically provide a *nearly unbounded* amount of *ephemeral* resources. As a result, a hypothetical serverless DAG scheduler may not necessarily care about traditional "scheduling"-related metrics and constraints (such as load balancing and cluster utilization), as an individual task could be executed anywhere in the serverless data center that is essentially managed by the service provider.

Yet, designing a fast and efficient serverless-oriented DAG engine introduces challenges. First, a task needs to be *dispatched* to a Lambda function as fast as possible. With this in mind, a logically centralized scheduler would inevitably introduce a performance bottleneck, especially for short-task dominated workloads. Second, as already mentioned, serverless platforms come with inherent constraints including limited outbound-only network connectivity; as such, a workflow has to rely on an external storage system for storing intermediate data, which impacts data locality and incurs extra network communications. Researchers have developed solutions on serverless computing platforms for supporting parallel jobs [9, 10, 14]; however, these attempts do not fully investigate decentralized DAG scheduling for serverless computing. Therefore, current state-of-the-art demands a new serverless-native DAG framework optimized to minimize the network communication overhead while maximizing data locality whenever possible.

In this paper, we argue that a serverless DAG engine urgently demands a radical redesign, with a focus shifted away from the techniques optimized for traditional DAG schedulers targeting serverful deployments. To this end, we present WUKONG, a serverless-oriented, decentralized, data-locality aware DAG engine. WUKONG uniquely exploits the elasticity of the serverless platform (in our case AWS Lambda) and completely delegates the requirements of load balancing, fairness, and resource efficiency to the serverless platform. WUKONG is novel in that it realizes *decentralized scheduling* where a DAG is partitioned into sub-graphs that are distributed to separate task executors (implemented as an AWS Lambda runtime). Lambda runtimes schedule the tasks in their sub-graphs and cooperate (at "joins" on sub-graph boundaries) to dynamically schedule tasks that are in two or more sub-graphs but that must only be executed once. WUKONG also provides efficient storage mechanisms for managing intermediate data; however, according to our factor analysis in section 3.5.2, WUKONG's decentralized design has the most influence on its performance. The decentralized design minimizes communication overhead and improves scalability by increasing data locality.

Specifically, we make the following contributions:

- We thoroughly investigate the problem space of DAG scheduling in serverless computing,
- We identify and evaluate a set of techniques to make DAG scheduling serverless-aware,
- We design and implement WUKONG, a serverless DAG engine attuned to AWS Lambda,
- We evaluate WUKONG to validate its efficacy and design tradeoffs.

3.3 Motivational Study: A Journey from the Serverful to the Serverless

Prior to the emergence of the serverless computing model, DAG schedulers were designed to work with a finite number of compute and storage resources. These schedulers have to maintain a (complete or partial) global view of which tasks are running where, and use this view to optimize with respect to certain predefined objectives. Serverless computing, on the other hand, offers a nearly infinite amount of ephemeral resources, which are transparently managed by the service provider. Consequently, traditional schedulers would fail to utilize cloud resources optimally. WUKONG takes a radically different approach and is motivated by the above observations with the goal of improving the performance for task dispatching with respect to serverless platforms. In this section, we present our motivational study of designing a fast and efficient serverless DAG engine.

3.3.1 A Strawman Scheduler

We began our journey by implementing a centralized DAG scheduler, which simply parsed the user-defined job code, generated a DAG data structure, and sent off the DAG tasks to a group of Lambda functions for execution.

Our strawman scheduler was a modification of the Python-written Dask `distributed` scheduler. Dask is an open-source parallel computing library for Python data analytics [44]. Traditionally, Dask `distributed` executes tasks within so-called worker processes, each running as a long-lived server across a cluster of machines. In Dask, the scheduler sends tasks to the worker processes for execution. Worker processes run as long-lived servers across a cluster of machines. The Dask `distributed` scheduler uses a communication protocol to communicate with workers and balance their load with respect to certain optimization constraints, such as data locality and memory consumption. We reused the DAG and communication protocol modules from Dask, used AWS Lambda for task execution instead of worker processes, and disabled load balancing as load balancing is handled by AWS Lambda.

In a typical serverful distributed processing framework, worker processes can directly

communicate with each other using TCP. A worker process that needs to execute a task `T` may find that `T`'s input data is not stored locally. This worker will then issue TCP requests for `T`'s input data to the workers who executed the upstream tasks that output this data. In our serverless computing environment, Lambda functions are not allowed to accept inbound TCP connection requests. Due to this constraint, the upstream tasks in WUKONG would have to store their output data in external distributed storage (a key-value store or KV store in short), from which the dependent downstream tasks can read their input data and make progress. Figure 2.1 depicts the strawman approach.

3.3.2 Publish/Subscribe Model

While the centralized strawman scheduler worked, it suffered from several performance bottlenecks. The first performance bottleneck was due to the large number of concurrent TCP connection requests sent to the scheduler from the Lambda functions. A short-lived Lambda function will immediately request a TCP connection with the scheduler to acknowledge the completion of its task. This makes it easy for a pool of thousands of newly invoked Lambda functions to overwhelm the scheduler. This is not a problem for a serverful deployment, e.g., a statically deployed Hadoop cluster with hundreds of worker nodes that established TCP connections at cluster initialization phase.

To address this problem, we adopted a pub/sub (publisher/subscriber) approach (Figure 2.2). The pub/sub scheduler provided higher performance than the strawman scheduler, since sending task completion messages through pub/sub channels was more efficient than using a large number of concurrent TCP connections; also, the number of network hops was reduced. The pub/sub architecture was easy to integrate, since external storage was already being used to store intermediate results.

3.3.3 +Parallel Invokers

While the pub/sub approach had substantially improved network performance, the framework struggled to launch Lambda functions quickly enough for large, bursty workloads. This

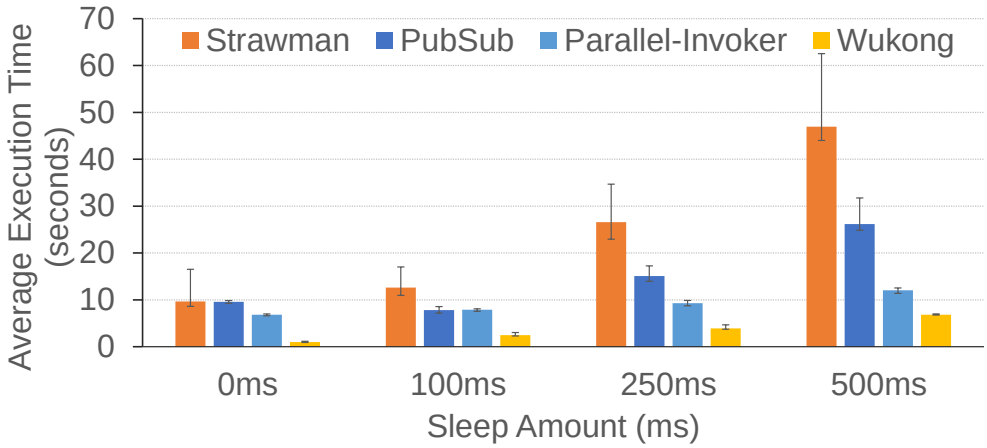


Figure 3.1: Performance comparison of different design iterations for Tree Reduction (TR). TR is a microbenchmark with a tree-like DAG topology [1], which combines neighboring elements until there is only one left. We ran TR with an initial array of 1024 numbers (i.e., 512 leaf tasks at the bottom of the DAG) on each system ten times and recorded the average (bars), and {min, max} (error bars). We intentionally added sleep-based delays in each task to simulate a compute task with a controllable duration.

is due to the large cost of invoking a Lambda function (e.g., invoking an AWS Lambda function takes about 50 milliseconds with the Boto3 AWS Python API). To scale-up Lambda invocation performance, we created a large number of dedicated Lambda-invoker processes co-located with the scheduler (Figure 2.3). When DAG task dependencies resolve, the scheduler evenly distributes task invocation responsibilities among multiple invoker processes, enabling (near-)linear speedup.

Figure 3.1 plots the average execution time achieved by each design iteration. We intentionally added a sleep-based delay to each task in order to simulate a compute task with controllable duration. For TR test with 0ms sleep delays, the performance difference between the strawman and pub/sub versions of our framework is roughly the same due to the fact that the TR workload is primarily dominated by the communication overhead of transferring the array over the network. As noted above, the parallel-invoker version is able to execute TR 24.2% faster than strawman and pub/sub. This is because TR is also characterized by a large number of leaf tasks. Specifically, the TR algorithm generates $\frac{n}{2}$ leaf

tasks, where n is the length of the input array. The parallel-invoker version can invoke the leaf tasks at a significantly higher rate than strawman and pub/sub; consequently, parallel-invoker performs better for workloads with a large number of leaf tasks. As the time span of each task increases, pub/sub start to show performance benefit against strawman, because a less number of TCP connections significantly reduces the amounts of IRQ requests which flood the strawman case. Parallel-invoker improves the performance against pub/sub, but is still sub-optimal due to network I/O overheads. The goal of WUKONG is to reduce the execution time of a DAG job, an optimal serverless DAG engine that dispatches DAG tasks with minimum runtime overhead.

One critical issue of the parallel-invoker architecture is its large commitment of resources to the centralized pub/sub scheduler throughout the whole course of the workload. Due to this, we moved to a decentralized scheduler design. This major design change came about as a result of a key observation, which was that the scheduler was only being used to launch downstream tasks as data dependencies of the DAG were resolved. Instead of scaling-up the invocation process of the centralized scheduler, each Lambda function could directly handle the responsibility of invoking downstream tasks without having to coordinate with the centralized scheduler. This lead to an effective, serverless-aware, scale-out design that is described next.

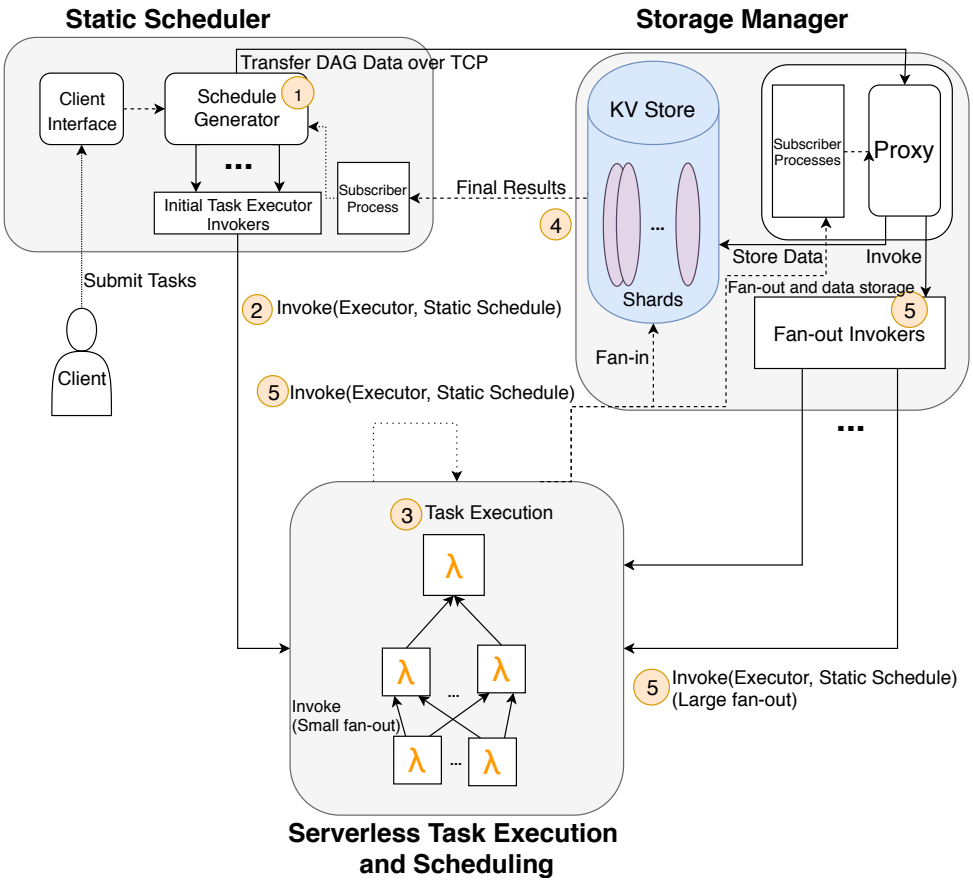


Figure 3.2: Overview of WUKONG architecture.

3.4 Wukong Design

In this section, we present the system design of WUKONG. We describe the high-level components and discuss the techniques used for static scheduling, task execution, dynamic scheduling, and managing storage.

3.4.1 High-Level Design

WUKONG consists of three major components: a static scheduler, a serverless task execution and scheduling runtime, and a storage manager. Figure 3.2 shows the high-level design of WUKONG. This figure reflects a major design revision to the Pub/Sub scheduler described

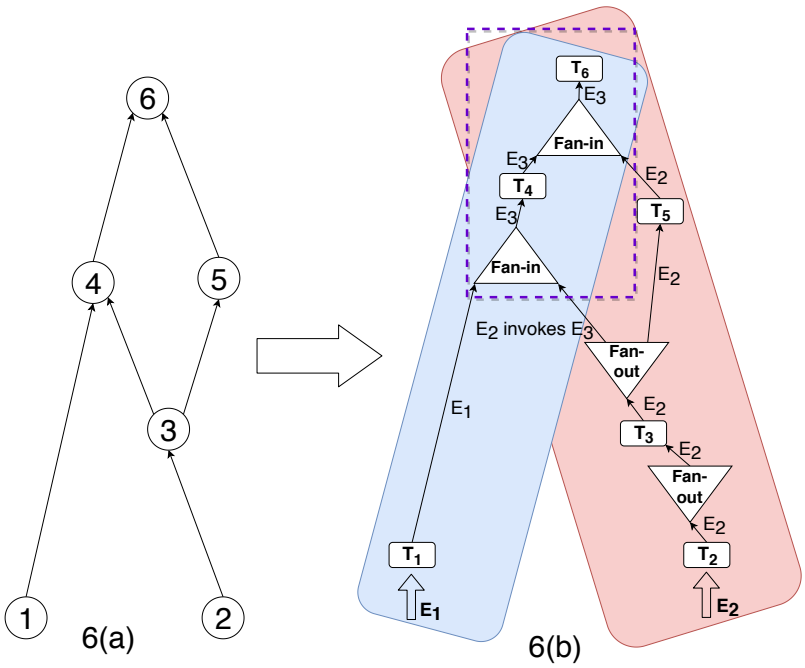


Figure 3.3: Static and dynamic scheduling.

at the end of section 3.3. This revision removed the requirement for Lambda functions to acquire downstream tasks from the KV store. We modified the scheduler to produce static schedules, where each schedule represents a sub-graph of the DAG. A static schedule contains all the task code and other required (static) information, e.g., data dependencies, for each task in the sub-graph. The scheduler now passes a static schedule to each Lambda function it invokes, meaning that each function starts with all of the task code that it may have to execute. This removed the necessity for Lambda functions to grab downstream task code from the KV store, which speeds up execution by decentralizing Wukong. Since static schedules contain the dependencies, scheduling operations for fan-in and fan-out processing can be done dynamically by the Lambda functions, which removes the need for a centralized scheduler to determine when data dependencies have been satisfied.

3.4.2 Static Scheduling

WUKONG users submit a Python computing job to WUKONG's DAG generator, which converts the job into a DAG. The static Schedule Generator generates static schedules from the DAG. For a DAG with n leaf nodes, n static schedules are generated. A static schedule for leaf node `L` contains all of the task nodes that are reachable from `L` and all of the edges into and out of these nodes. The data for a task node includes the task's code and the KV Store keys for the task's input data. The schedule for `L` is easily computed using a depth-first search (DFS) that starts at `L`. Figure 3.3(a) shows a DAG with two leaf nodes. Figure 3.3(b) shows the two static schedules that are generated from the DAG — `Schedule 1` is the nodes and edges in the region colored blue (left) and `Schedule 2` is the nodes and edges in the region colored red (right).

Static schedules are used to reduce the number of network I/O operations that must be performed by the Task Executors, which improves overall system performance. Instead of executing a single task and retrieving the next task from the KV store, Task Executors receive a static schedule of all of the tasks (including the task code) they may possibly execute.

A static schedule contains three types of operations: task execution, fan-in, and fan-out. Note that there is at least one fan-in or fan-out operation between each pair of tasks. To simplify our description, when tasks `T1` is followed immediately by task `T2` in a DAG and `T1` (`T2`) has no fan-out (fan-in), we add a trivial fan-out operation between `T1` and `T2` in the static schedule. This fan-out operation has one incoming edge from `T1` and one outgoing edge to `T2`, i.e., there is no actual fan-out. In Figure 3.3(a) and Figure 3.3(b), this is the case for tasks `T2` and `T3`.

A task operation may appear in more than one static schedule. In Figure 3.3(b), tasks `T4` and `T6` are both in `Schedule 1` and `Schedule 2`. This will create a scheduling conflict between the Task Executors that are assigned to schedule these tasks, since tasks `T4` and `T6` should be executed only once. There is not enough information available in the DAG to statically determine how to resolve scheduling conflicts so that execution time is minimized;

instead, conflicts are resolved by dynamic scheduling operations performed by the Task Executors. Note also that a static schedule does not map a given task `T` to a processor; this mapping is done dynamically and automatically by the AWS Lambda runtime when the Task Executor that will (eventually) execute task `T` is invoked. A static schedule only specifies a valid partial-ordering of the tasks it contains — tasks are to be executed in bottom-up order, starting with the leaf node in the static schedule. Dynamic scheduling during task execution imposes the remaining constraints on task order. The time and place that tasks are executed is determined at runtime.

3.4.3 Task Execution and Dynamic Scheduling

Execution starts when the scheduler's Initial Task Executor Invokers assign each static schedule produced by the Schedule Generator to a separate AWS Lambda function, which we refer to as a *Task Executor*, and invoke the set of initial Task Executors in-parallel. Each of these Task Executors performs the first operation in its static schedule, which is always to execute the leaf node task in its static schedule. In Figure 3.3(b), Task executors `E1` and `E2` execute leaf tasks `T1` and `T2`, respectively. An Executor will then execute the tasks along a single path in its static schedule, enforcing the static ordering of tasks along the path.

If Task Executor `E` executes a fan-out operation with only one out edge, the operation has no effect — Executor `E` simply performs the next operation in its schedule, which will be to execute the next task. A Task Executor may thus execute a sequence of tasks before it reaches a fan-out operation with more than one out edge or a fan-in operation. For such a sequence of tasks, there is no communication required for making the output of the earlier tasks available to later tasks for input. All intermediate task outputs are cached in the local memory of the the Task Executor.

If Task Executor `E` executes a fan-out operation with n (where $n > 1$) out edges, then `E` invokes $n - 1$ new Task Executors. The intermediate output objects that are needed by the new Executors are sent to the Storage Manager for storage in the KV Store, and the associated keys are passed to the invoked Executors as arguments. Each of the $n - 1$

Executors will be assigned a static schedule that begins with one of the $n-1$ out edges. Each of these (possibly overlapping) static schedules corresponds to a sub-graph of `E`'s static schedule. Executor `E` continues task execution and scheduling along the remaining out edge and executes the next operation encountered on this edge. We say that `E` becomes the Executor for one out edge and invokes Executors for the remaining $n-1$ out edges. In Figure 3.3(b), each fan-out operation has one edge labeled "becomes" and 0 or more out edges labeled "invokes". The label on an in or out edge also indicates the Executor that is performing the dynamic scheduling operations that involve that edge. For Executor `E2`, the first fan-out operation is trivial. On `E2's` second fan-out operation, `E2` becomes the Executor that will execute `T5` and invokes Executor `E3`.

As mentioned above, a fan-in operation represents a scheduling conflict between two or more Task Executors that are executing overlapping static schedules. If Task Executor `E` executes a fan-in operation with n (where $n > 1$) in-edges, then `E` and the $n-1$ other Task Executors involved in this fan-in operation cooperate to see which one of them will continue their static schedules on the out edge of the fan-in. The Task Executors that do not continue will send their intermediate output objects to the Storage Manager and stop. In Figure 3.3(b), the first fan-in operation of Executors `E1` and `E3` resolves the conflict between their static schedules. We assume that `E3's` fan-in operation is executed after `E1`'s fan-in operation; thus, `E1` stops and `E3` continues executing its static schedule at the out edge of the fan-out. At `E3's` next fan-in operation, which also involves `E2`, we assume `E3` executes its fan-in operation last so that `E2` stops and `E3` executes task `T6` and then stops.

Task Executors cooperate on fan-in operations for a fan-in `F` by accessing an associated dependency counter for `F` that is stored in the KV Store. This counter tracks the number of `F`'s input dependencies that have been satisfied during execution. When a Task Executor `E` finishes the execution of a task that is one of the input dependencies of `F`, Executor `E` performs an atomic increment operation on the dependency counter of `F`. The updated value of the dependency counter is then compared against the number of input dependencies of `F`. If the value of the dependency counter is equal to the number of input dependencies, then

all input dependencies of `F` have been satisfied and the task `T` on the out edge of `F` is ready for execution. In this case, task `E`, which executed the last dependent task of the fan-in, will continue its static schedule by executing `T`. If, instead, the value of the dependency counter is less than the number of input of `F`, then some input dependencies of `F` have yet to be satisfied. In that case, task `T` is not ready for execution, so `E` saves its intermediate output objects and stops. Notice that no Task Executor waits for any input dependencies of a fan-in to be satisfied. Note that AWS Lambda would bill Task Executors for wait time, which is why waiting is avoided.

For fault tolerance, we relied on the automatic retry mechanism of AWS Lambda, which allows for up to two automatic retries of failed function executions. In the future, we will investigate more advanced error handling mechanisms.

3.4.4 Storage Management

The Storage Manager performs various storage operations on behalf of the Task Executors and the Scheduler. At the start of workflow processing, the Storage Manager receives the workflow DAG and the static schedules derived from the DAG from the Scheduler.

Intermediate and Final Result Storage

Task Executors publish their intermediate and final task output objects to the KV Store. Final outputs are relayed to a Subscriber process in the Scheduler for presentation to the Client.

Small Fan-out Task Invocations

When a Task Executor performs a fan-out operation that has a small number of out edges, the Task Executor will make the necessary Executor invocations itself. However, sequentially executing a large number of invocations is time consuming so the Executor Task invocations are performed in parallel with assistance from the Storage Manager.

Large Fan-out Task Invocations

When a fan-out has a number of out edges that is larger than a user-specified threshold, the Task Executor publishes a message that is relayed to a Subscriber process in the Storage Manager, which passes the message to the Proxy. This message contains an ID that identifies the fan-out's location in the DAG. The Proxy uses the DAG and the fan-out ID to identify the fan-out's out edges in the DAG. This allows the Proxy, with the assistance of the Fan-out Invokers in the Storage Manager, to make the necessary Task Executor invocations, in parallel. The Proxy passes to each Executor its intermediate inputs (or their key values in the KV Store) and the Executor's static schedule.

3.5 Preliminary Results

We have implemented WUKONG using roughly 6,000 lines of Python code (about 2,500 LoC for the AWS Lambda Runtime, 2,400 LoC for the Storage Manager, and 1,100 LoC for the Static Scheduler). WUKONG currently supports AWS Lambda. Porting WUKONG to other public cloud and open source platforms is our work in progress.

Experimental Goals and Methodology.

The goals of our preliminary evaluation were to:

- identify and describe the main factors influencing the performance and scalability of WUKONG,
- compare WUKONG against the serverful Dask framework to determine whether WUKONG can achieve comparable performance, even with the inherent limitations imposed by AWS Lambda.

We compare the performance of WUKONG against Dask `distributed` on two different setups: a five-node EC2 cluster with each virtual machine (VM) running five worker processes and a local setup on a laptop with four worker processes.

We repeated the same tests on an easy-to-use laptop computer to further demonstrate that, with the same workload, WUKONG can achieve superior performance with minimal cluster administration effort.

Our evaluation was performed on AWS. The static scheduler ran in a `c5.18xlarge` EC2 VM and the KV Store was a Redis cluster partitioned across ten `c5.18xlarge` shards. The KV Store proxy was co-located on the same VM as one of the ten Redis shards. Each Lambda function was allocated 3GB memory with a timeout parameter set to two minutes.

Each node in the five-node cluster was an EC2 `t2.2xlarge` VM. We configured this cluster with general-purpose VMs to see if our serverless platform could match their performance. We opted to not configure a cluster of increased price and performance as we cannot

configure our AWS Lambda functions to match the processing power of such a cluster. Further, we cannot control for the various restrictions held in place by Amazon including the rate at which we can invoke Lambda functions, the memory allocated to Lambda functions (above 3GB), the network resources allocated to each function, etc.

The laptop was equipped with a two-core `Intel i5` CPU @ *2.30GHz*. Each Dask worker was allocated 2GB of laptop memory.

We describe the tested DAG applications as follows.

Tree Reduction (TR)

TR sums the elements of an array. TR repeatedly adds adjacent elements until only a single element remains. The implementation used here is general-purpose; it is not optimized for a highly distributed, serverless algorithm, serving as a microbenchmark for effectively evaluating serverless DAG engine.

General Matrix Multiplication (GEMM)

GEMM, as the core of many linear algebra algorithms, performs matrix multiplication. We evaluate the performance of Wukong for GEMM with two different matrix sizes: $10,000 \times 10,000$ and $25,000 \times 25,000$.

Singular Value Decomposition (SVD)

Two SVD workloads are used. The first workload computes the SVD of a tall-and-skinny matrix, i.e., a matrix with a significantly larger number of rows than columns (SVD1). The second workload computes the rank-5 SVD of an $n \times n$ matrix using an approximation algorithm (SVD2) provided by [45]. We use both SVD workloads as a real-world application to evaluate the performance of Wukong for increasingly large SVD problem sizes.

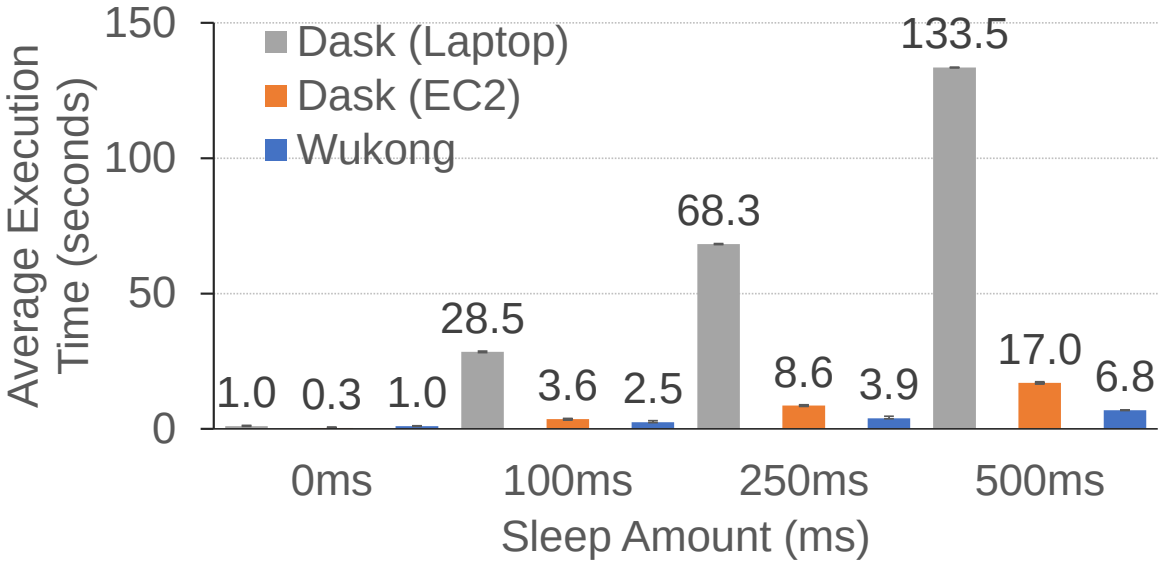


Figure 3.4: TR performance comparison.

Support Vector Classification (SVC)

SVC is a real-world machine learning application. We evaluate the performance of Wukong on SVC with increasingly large problem sizes. This workload is a benchmark that was retrieved from the publicly available Dask-ML benchmarks [46].

3.5.1 End-to-End Performance Comparison

As mentioned earlier, serverless computing suffers from cold starts. We address this issue by warming up a pool of Lambdas, which is the same strategy employed by ExCamera [14].

Due to AWS' planned cold-start performance optimizations for Lambdas running within a virtual private cloud [47], "cold start" penalties should not be nearly as large of an issue in the future.

We first examine the performance of TR (for a preliminary analysis that compares the various design iterations please refer to Figure 3.1 in section 3.3.3). As shown in Figure 3.4, Wukong greatly outperforms all previous versions of the framework. The decentralized scheduler reduces the network I/Os required to complete the workload; however, due to the

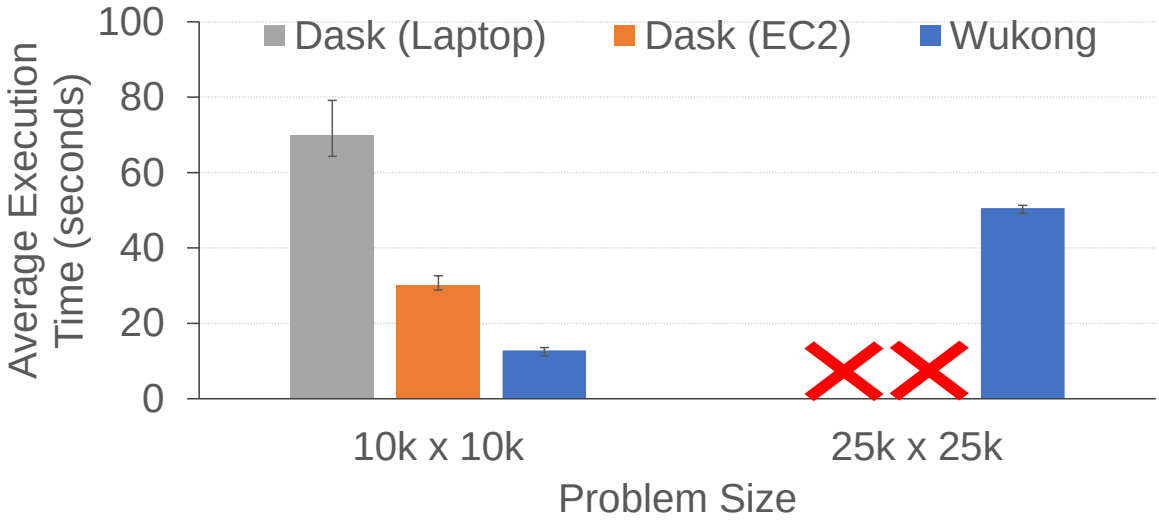


Figure 3.5: GEMM performance comparison.

extremely short-duration add operations used by TR with 0ms sleep delays, the communication overhead of transferring the underlying array greatly outweighs the performance gains from increased parallelism. This is why WUKONG achieves lower performance than Dask (EC2). WUKONG outperforms all other execution platforms when small sleep delays are added to each operation of the tree reduction. WUKONG executes 2.5× faster than Dask (EC2) in the case of 500ms delays. These small delays simulate additional work for each task. The results of this workload with added delays indicate that for workloads with longer tasks, the increased parallelism provided by WUKONG outweighs the communication overhead, demonstrating that our decentralized DAG scheduler incurs minimum overheads.

The results of our GEMM tests further demonstrate WUKONG's superiority in elasticity and performance. In the case of case of $10,000 \times 10,000$ matrix multiplication, WUKONG executed the workload more than twice as fast as Dask (EC2) and more than five times as fast as Dask (Laptop). Dask (EC2) could likely perform this workload faster if the cluster was larger, whereas for WUKONG, it leverages the large number of CPUs provided by AWS Lambda to elastically scale up the performance. When multiplying $50,000 \times 50,000$

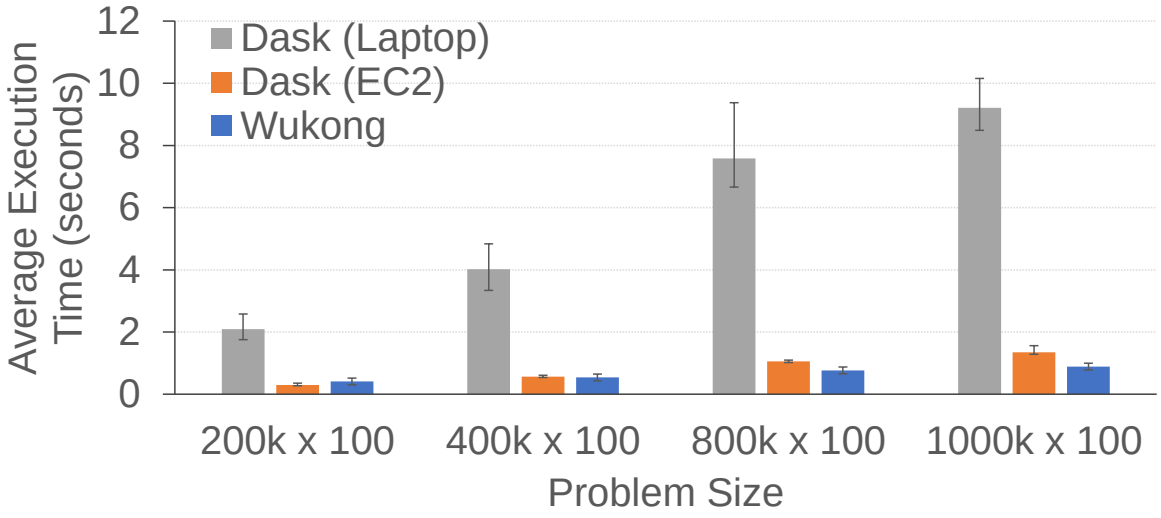


Figure 3.6: SVD1: SVD of tall-and-skinny matrix.

matrices, both setups of Dask (Laptop and EC2) suffered from out-of-memory (OOM) errors, failing to complete the job. Our analysis of GEMM on WUKONG indicates that these workloads were dominated by the communication overhead of transferring portions of the matrix to the Task Executors.

Next we analyze the performance of the two SVD workloads. For SVD1, we used the following numbers of rows: $200k$, $400k$, $800k$, and $1,000k$. Figure 3.6 shows that both Dask (EC2) and WUKONG were able to greatly outperform Dask (Laptop). For the first two problem sizes, Dask (EC2) out-performed WUKONG; however, as the problem size increased, the performance of WUKONG began to exceed that of Dask (EC2). This is because the parallelism from AWS Lambda began to outweigh the communication overhead of the workload. Even so, the overhead associated with network I/Os was a significant factor in the performance of this workload on WUKONG.

The dominance of communication in SVD was further demonstrated by the first three workload sizes of the SVD2 on a general $n \times n$ matrix (Figure 3.7). Dask (EC2) was faster than WUKONG for relatively smaller problem sizes, since the statically-deployed Dask `distributed` cluster supports direct worker-to-worker communication with less network I/O

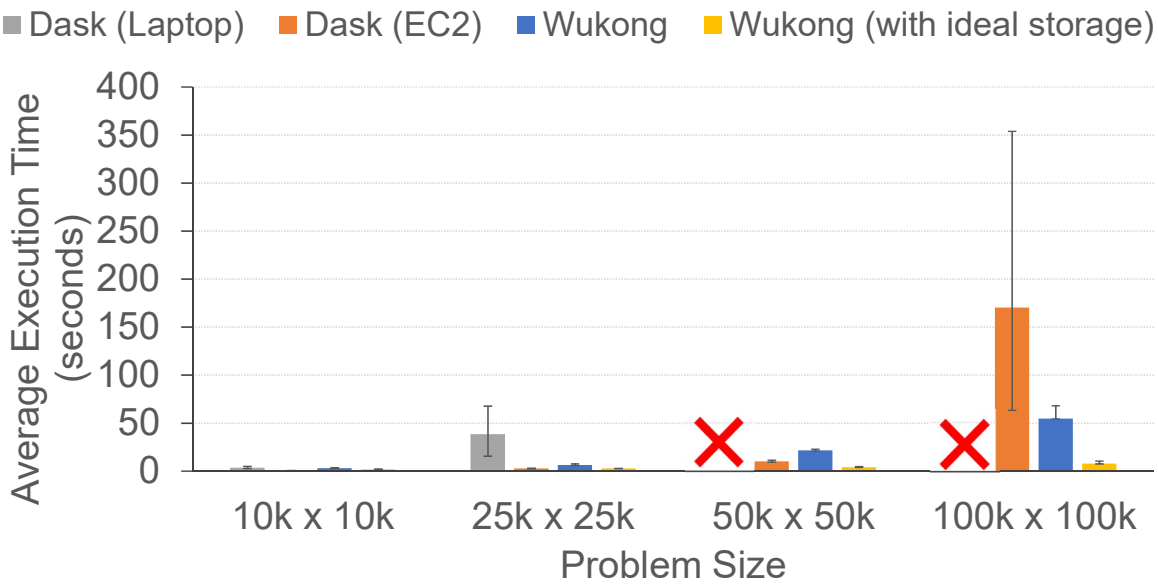


Figure 3.7: SVD2: SVD of general matrix.

overhead (especially for large intermediate results), and the CPU resources of the cluster did not yet become a bottleneck. Additionally, Dask (Laptop) suffered from OOM errors in the $50k \times 50k$ case and was unable to complete the workload. Finally, WUKONG executed the $100k \times 100k$ workload 3.1× faster than Dask (EC2), again because of the elasticity of WUKONG. WUKONG does not require extra administration effort for scaling out the computation capacity, whereas Dask (EC2) would do, thus imposing extra burden to the end users. The number of Lambda functions used for each of the workloads was 84, 480, 295, and 1082, respectively. The $50k \times 50k$ workload used less Lambdas than the $25k \times 25k$ workload due to the strategy used to partition the initial input data. Different input data partitioning strategies may introduce different parallelism-communication tradeoffs and affect scalability. We plan to investigate partitioning strategies as part of our future work.

Finally, we analyze the performance of SVC on WUKONG (Figure 3.8). We varied the SVC problem size (in this case, number of samples) over the values $100k$, $200k$, $400k$, and $800k$. While Dask (EC2) completes the job slightly faster than WUKONG for the smallest problem size, the performance of WUKONG begins to exceed Dask (EC2) as the problem

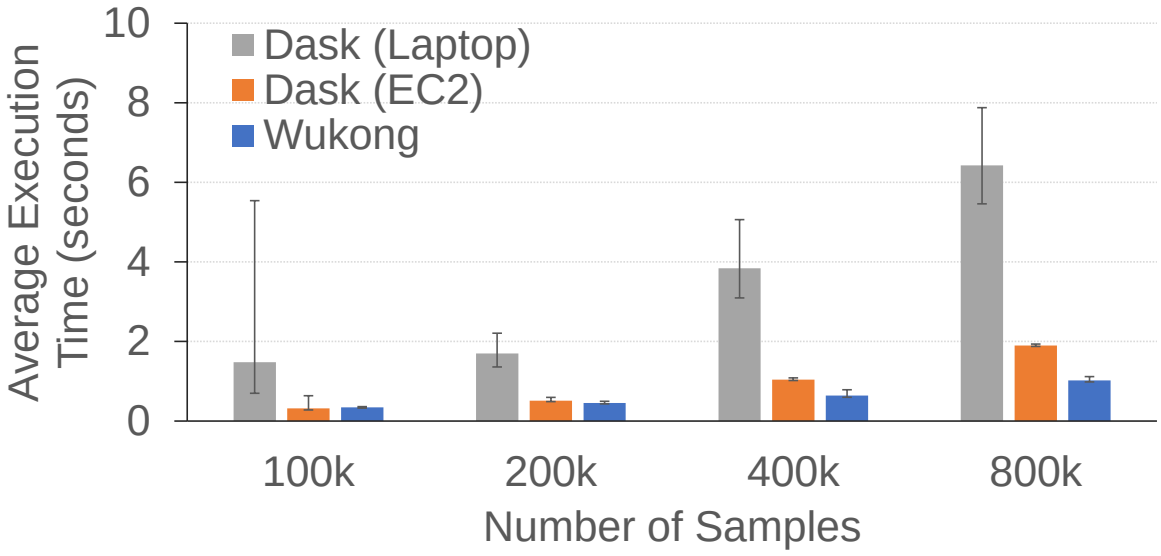


Figure 3.8: Performance comparison of SVC machine learning classification.

size increases. The performance gap increases as the problem size varied from $400k$ to $800k$. For a sample number of $800k$, WUKONG is able to execute the workload nearly $2\times$ as fast as Dask (EC2). This again strengthens our confidence that WUKONG can serve as a generic DAG engine for accelerating complex real-world applications such as machine learning.

3.5.2 Factor Analysis

WUKONG is able to effectively scale out to support large problem sizes and workloads. Figure 3.9 shows the amount that each major version of WUKONG contributed to the overall performance improvement from the original *Strawman* version to the current version. The most significant improvement came as a result of the decentralization of the Task Executors. Prior to Task Executor decentralization, Task Executors would only execute the task initially given to them by the static scheduler. Once decentralized, Task Executors instead retrieved new tasks from the KV Store each time they completed the execution of their current task.

Other significant improvements to the overall performance of WUKONG included the use of dedicated task invoker processes, which originated in the *Parallel Invokers* version,

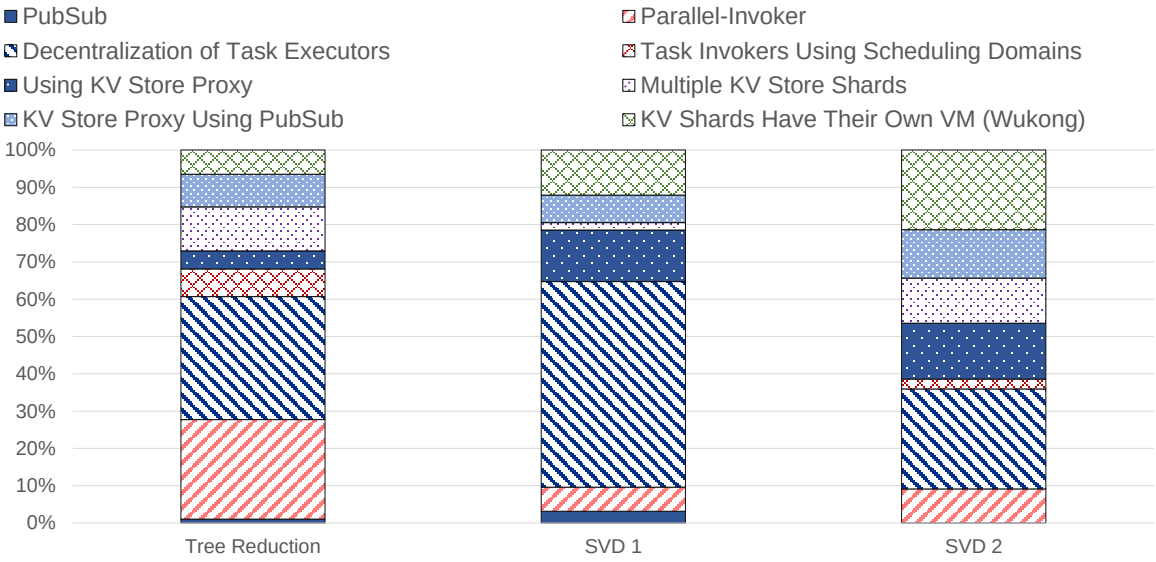


Figure 3.9: Contributing factors of different optimization techniques employed in WUKONG.

and the use of the KV Store Proxy to parallelize large task fan-outs. The effect of the KV Store Proxy varied depending on the workload since workloads that lacked high fan-outs would not actually utilize the proxy. Switching the communication protocol used by the KV Store Proxy from TCP to Redis PubSub also resulted in a fairly substantial performance improvements. Just as for the static scheduler, Redis PubSub enabled the KV Store Proxy to handle a higher volume of messages from Task Executors. Finally, running each KV Store shard on its own separate VM resulted in a significant performance improvement. Initially, all KV Store shards were running on the same VM, which resulted in resource contention for network bandwidth. Placing each shard on its own VM eliminated this bottleneck.

3.5.3 Overhead Quantification

The overhead associated with storing and retrieving large intermediate data values during workload execution is a major factor that impacts WUKONG's performance. For workloads characterized by short tasks and large communication overheads, WUKONG is not able to outperform Dask (EC2). This is most prevalent in the tree reduction workload without

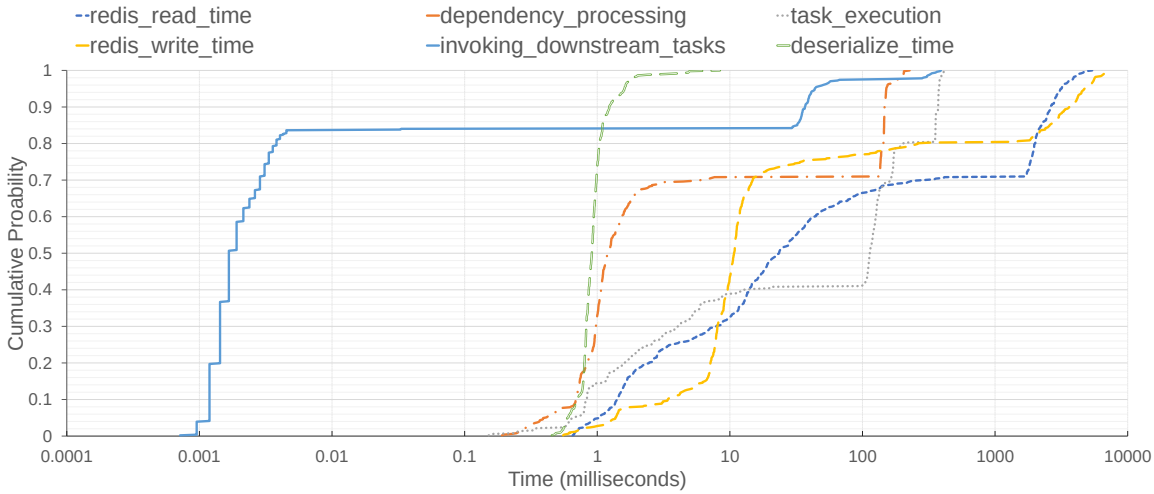


Figure 3.10: CDF breakdown of tasks in SVD2 with a $50k \times 50k$ matrix.

sleep delays, shown in Figure 3.4, and when computing the SVD of a square matrix, shown in Figure 3.7.

To quantify such I/O overhead, we conduct a detailed analysis with SVD2, by breaking down Wukong's execution duration into fine-grained factors. Figure 3.10 shows a latency distribution of individual tasks in SVD2 of a $50k \times 50k$ matrix. We observe that there were a small number of KV store read and write operations which took upwards of ten seconds to complete. While a majority of tasks did not experience such communication overhead, the long network I/Os experienced by a minority of the tasks have a large impact on the workload's overall performance.

In order to estimate the improvement in performance that Wukong could obtain if we were to use an ideally-fast (i.e., fully-optimized) intermediate storage, we executed a modified variant of SVD2 in which all array data was randomly generated each time it was used (instead of being written in and retrieved from the KV store). In Figure 3.7, the right-most (yellow-colored) bar shows the performance of Wukong with this ideal intermediate storage. While the performance of Dask (EC2) is still better than Wukong (with idea storage) for the smallest workload size, the performance is roughly the same in the $25k \times 25k$ case. Moreover, Wukong (with ideal storage) is able to perform 1.67× faster than Dask

(EC2) for the $50k \times 50k$ workload in this experiment. As discussed earlier, WUKONG (in its current form) is already able to outperform Dask (EC2) by over 115 seconds on-average for the largest problem size. When using an ideal KV store, WUKONG would execute the workload in 95.50% less time than Dask (EC2). These results highlight the magnitude by which network communication overhead negatively affects the overall performance of WUKONG.

Data locality is another key factor which influences the performance of WUKONG. Increased data locality enables Task Executors to carry out the workload without needing to retrieve dependent inputs from the KV Store. This reduces the communication overhead, thereby increasing the performance of the framework. The overall effect of data locality largely depends on the size of the data values kept in the Task Executor's local storage. Our analysis of the network I/O performance found that the transfer of intermediate data objects that were tens to hundreds of megabytes in size were the cause of longer execution times (as opposed to smaller intermediate data objects). Consequently, WUKONG is able to utilize the local data stores on Task Executors most effectively when large objects are stored.

3.5.4 Limitations

One limitation of our evaluation is that we did not compare WUKONG against other serverless DAG engines. This was because all of the systems use different representations for their DAG's, and these representations are large and complicated. Consequently, it is nontrivial to convert a DAG from one system to another. A comparison between serverful Dask and WUKONG is possible because they use the same DAG representation. We are currently investigating DAG representations of other serverless DAG engines so that we can make a thorough comparison between WUKONG and other frameworks and understand the pros and cons of their design decisions.

3.6 Wukong 1.0 Chapter Summary

We have presented WUKONG, a high-performance DAG engine that implements decentralized scheduling by exploiting the elasticity and scalability of AWS Lambda to reduce network I/O overhead and improve data locality. Our evaluation shows WUKONG is competitive with a traditional serverful DAG scheduler Dask and demonstrates that decentralizing task scheduling contributes a significant portion of the improvement in overall performance. As part of our future work, we are exploring new techniques to fundamentally improve the performance of intermediate storage for serverless DAG workloads.

WUKONG is open sourced and is available at: `https://github.com/mason-leap-lab/Wukong`.

Acknowledgments.

We thank our shepherd, Shadi Ibrahim, and the reviewers for their valuable feedback. This work is sponsored in part by NSF under CCF-1919075 and an AWS Cloud Research Grant.

Chapter 4: Wukong 2.0 - Achieving High Performance and Cost-Effectiveness

WUKONG 1.0's contribution is a decentralized DAG-scheduling technique. Decentralized scheduling partitions the input DAG into sub-DAGs that are cooperatively scheduled by a fleet of Lambda executors. Our evaluation of WUKONG 1.0 showed that it achieves competitive performance and low costs (compared to serverful Dask); however, WUKONG occasionally experiences long communication delays at fan-out points. These delays occur during network I/O operations that access the Redis intermediate data storage. In particular, reads and writes of large intermediate data objects account for a majority of the execution time for workloads such as SVD2 (i.e., Single Value Decomposition (SVD) of an $n \times n$ matrix). In addition, Redis is unable to support large-scale workloads in which hundreds or thousands of Lambdas are concurrently performing I/O operations.

The following example illustrates how communication delays can occur at fan-out points. Refer again to 1.1. Fan-out tasks $F1$ and $F2$ can be executed in parallel. The Executor for task `Sum(A)` can invoke a new Executor for executing task $F1$ and another new Executor for executing task $F2$. Before it invokes these new Executors, the Executor for `Sum(A)` must output the computed sum to intermediate storage, so that the sum can be input by the Executors for tasks $F1$ and $F2$. However, it is possible that the communication delay that results from writing and then reading the sum from intermediate storage may far exceed the execution time that is saved by executing tasks $F1$ and $F2$ in parallel. If so, then execution time can be reduced by assigning fan-out tasks $F1$ and $F2$ to the Executor for task `Sum(A)` instead of to two new Executors. This reassignment of tasks is referred to as "clustering" tasks `Sum(A)`, $F1$, and $F2$.

The task clustering techniques we have developed reduce the amount of parallelism in order to avoid large communication delays. Clustering tasks increases data locality

and substantially reduces network I/O. Communication delays that cannot be avoided can be reduced by coupling task clustering with serverless, high-performance cloud-storage to improve the throughput and latency of intermediate storage. These improvements are implemented in WUKONG 2.0, which is described in our second paper, WUKONG: A High-Performance Framework for Serverless Parallel Computing.

4.1 Abstract

Executing complex, burst-parallel, directed acyclic graph (DAG) jobs poses a major challenge for serverless execution frameworks, which will need to rapidly scale and schedule tasks at high throughput, while minimizing data movement across tasks. We demonstrate that, for serverless parallel computations, decentralized scheduling enables scheduling to be distributed across Lambda executors that can schedule tasks in parallel, and brings multiple benefits, including enhanced data locality, reduced network I/Os, automatic resource elasticity, and improved cost effectiveness. We describe the implementation and deployment of our new serverless parallel framework, called Wukong (Locality-Aware, Decentralized Serverless scheduling), on AWS Lambda. We show that Wukong achieves near-ideal scalability, executes parallel computation jobs up to 68.17× faster, reduces network I/O by multiple orders of magnitude, and achieves 92.96% tenant-side cost savings compared to numpywren.

4.2 Introduction

In recent years, a new cloud computing model called serverless computing or Function as a Service (FaaS) [34] has emerged. Serverless computing enables a new way of building and scaling applications and services by allowing developers to break traditionally monolithic server-based applications into finer-grained cloud functions. Developers write function logic while the service provider performs the notoriously tedious tasks of provisioning, scaling, and managing the backend servers that the functions run on [36].

Serverless computing solutions are growing in popularity and finding their way into both commercial clouds (e.g., AWS Lambda, Google Cloud Functions, and Azure Functions) and open source projects (e.g., OpenWhisk). While serverless platforms were originally intended for event-driven, stateless applications [38], a recent trend is the use of serverless computing for more complex, stateful, parallel applications.

Some types of compute- and data-intensive applications are inherently parallelizable and can be structured as a directed acyclic graph (DAG) of short, fine-grained tasks [21, 30, 39, 40]. The large-scale parallelism and auto-scaling services provided by serverless platforms makes them well-suited for such kinds of burst-parallel fine-grained tasks that characterize DAG-based parallel computation workflows. Burst-parallel applications include data analytics [40], optimization algorithms [48], and real-time machine learning classifications such as support vector machines (SVM) [49–51]; these applications typically demand low-latency scheduling [30] with large-scale parallelism [52].

FaaS providers charge function execution time at a fine granularity – AWS Lambda bills on a per-invocation basis. Workloads with short tasks can take advantage of this fine-grained pay-per-use pricing model to keep monetary costs low. Consequently, serverless computing can be leveraged by next-generation, burst-parallel workloads in high-performance computing (HPC) and data analytics.

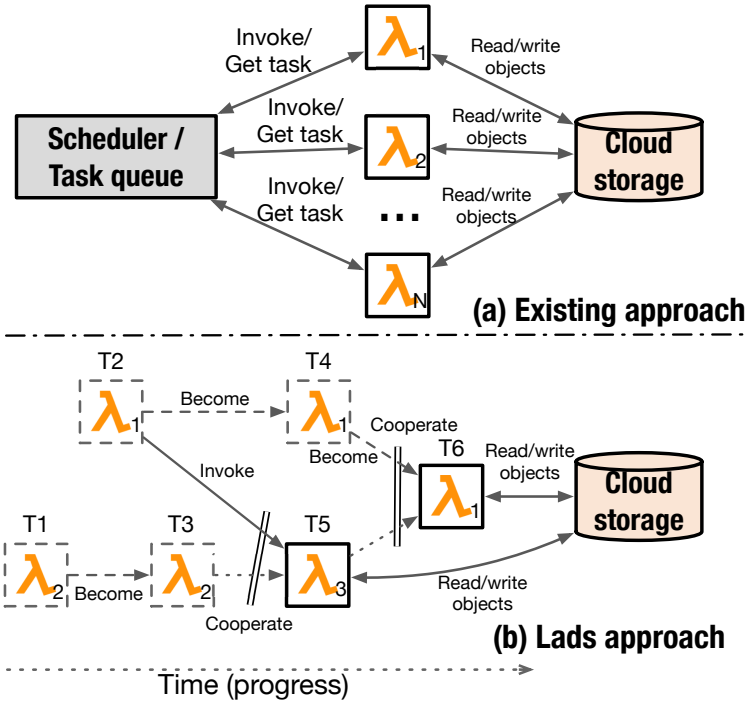


Figure 4.1: In (a), the central scheduler tracks all task completions, updates all task dependencies, and identifies all ready tasks. The scheduler dispatches ready tasks to Lambda executors; or the scheduler deposits ready tasks into a shared work queue, and a pool of Lambda executors contend for the queued tasks. Intermediate task inputs and outputs are stored outside of the executors, which reduces data locality. In (b), task scheduling is performed by a fleet of Lambda executors that schedule and execute their assigned tasks in parallel and cooperate to ensure that task dependencies are satisfied. Intermediate task inputs and outputs may be stored inside the executors, which increases data locality. This approach also enables fine-grained and automatic Lambda resource elasticity, as Lambda executors finish assigned tasks and return (e.g., Lambda 2).

Migrating such applications from a traditional serverful deployment to a serverless platform presents unique opportunities. Traditional serverful deployments rely on existing workflow management frameworks such as MapReduce [42], Apache Spark [43], Sparrow [30], and Dask [44] to provide a logically centralized scheduler for managing task assignments and resource allocation. The scheduler traditionally has various objectives, including load balancing, maximizing cluster utilization, ensuring task fairness, and so on. However, a traditional serverful scheduler is not required by serverless computing. This is because: (1) FaaS providers are responsible for managing the "servers" (i.e., where the task executors

are hosted); and (2) serverless platforms typically provide a *nearly unbounded* amount of *ephemeral* resources. As a result, a hypothetical serverless parallel computing framework may not necessarily care about traditional "scheduling"-related metrics (such as load balancing and cluster utilization), since the framework has no control over where tasks are executed. (The service provider, of course, cares about these metrics.)

Yet, designing an efficient serverless-oriented parallel computing framework introduces unique challenges. First, while serverless platforms (e.g., AWS Lambda) promise to offer superior elasticity and auto-scaling properties, the serverless invocation model imposes non-trivial scheduling overhead. Unlike a typical serverful parallel framework where the central scheduler directly communicates with each worker process using TCP [42,53], in a serverless setup, the scheduler can dispatch tasks to serverless workers in one of three ways.

Figure 4.1(a) depicts a high-level overview of all three methods. In method #1, the scheduler invokes a Lambda function (using the HTTP protocol) to dispatch the task code and execute the task. Note that with this method, there is a one-to-one association between tasks and Lambda functions. Given an average invocation overhead of 50 ms (typical for AWS Lambda functions), the scheduler could quickly become a performance bottleneck, especially for large and complex jobs with thousands of tasks. These observations indicate that a naive attempt to simply port an existing serverful DAG framework to serverless computing will be unsuccessful. In order to create a performant, cost-effective serverless DAG engine, new techniques must be developed to fully take advantage of the characteristics of the serverless platform.

In method #2, the scheduler launches short-lived[1] Lambda executors as workers that establish TCP connections with the scheduler and receive RPC requests for task processing. Example frameworks include ExCamera [14] and PyWren [10]. Task executors within these frameworks may execute several tasks as opposed to just one as with the first method.

Similarly, in method #3, the scheduler places tasks in a shared work queue. These tasks are retrieved from the queue by serverless executors; state-of-the-art systems such

[1]Lambda functions may run up to 900 seconds in AWS cloud.

as numpywren [9] launch stateless Lambda executors that connect to a centralized shared queue and constantly retrieve tasks from the queue. Frameworks using this method may have a component separate from the central scheduler that is responsible for invoking the AWS Lambda executors. This is sometimes referred to as a "provisioner". In the latter two approaches, as shown in Figure 4.1(a), a tightly synchronized central scheduler tracks all task completions, updates all task dependencies, and identifies any ready tasks. The scheduler dispatches ready tasks to Lambda executors, or the scheduler deposits ready tasks into a shared work queue, and a pool of Lambda executors contend for the queued tasks. Intermediate task inputs and outputs are stored externally, which reduces data locality.

The second challenge is that serverless platforms come with inherent constraints, including bandwidth-limited, outbound only network connectivity; therefore, serverless workflows must rely on external cloud store for intermediate data storage and exchange, which creates excessive data movement overhead. Data locality enhancement is thus critical for minimizing communication costs.

Researchers have developed serverless parallel computing frameworks that support parallel job processing [9, 10, 14]; however, these solutions do not fully address the aforementioned performance issues of efficient scheduling and data locality, which leads to long scale-out delays, sub-optimal performance, and higher monetary cost. To this end, we design and build a new serverless parallel computing framework called WUKONG[2]. WUKONG is a serverless-oriented, decentralized, locality-aware, and cost-effective parallel computing framework. *The key insight of* WUKONG *is that partitioning the work of a centralized scheduler (i.e., tracking task completions, identifying and dispatching ready tasks, etc.) across a large number of Lambda executors, can greatly improve performance by permitting tasks to be scheduled in parallel, reducing resource contention during scheduling, and making task scheduling data locality-aware, with automatic resource elasticity and improved cost effectiveness.*

Scheduling is decentralized by partitioning a DAG into multiple, possibly overlapping,

[2] `https://mason-leap-lab.github.io/Wukong`

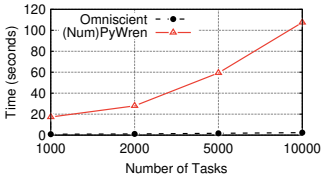


Figure 4.2: (Num)PyWren scaling tasks on AWS Lambda.

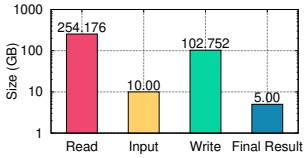


Figure 4.3: Numpywren GEMM read and write amplification.

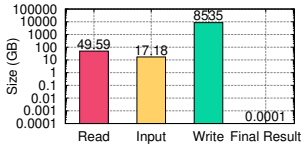


Figure 4.4: Numpywren TSQR read and write amplification.

subgraphs. Each subgraph is assigned to a task Executor (implemented as an AWS Lambda function runtime) that is responsible for scheduling and executing tasks in its assigned subgraph. This decentralization brings multiple key benefits:

Enhanced data locality and reduced resource contention: Decentralization improves the data locality of scheduling. Unlike PyWren [10] and numpywren [9], which require executors to perform network I/Os to obtain each task they execute (since numpywren's task executor is completely stateless), Wukong preserves task dependency information on the Lambda side. This allows Lambda executors to cache intermediate data and schedule the downstream tasks in their subgraph locally, i.e., without constant remote interaction with a centralized scheduler.

Harnessing scale and local optimization opportunities: Decentralizing scheduling allows an Executor to make local data-aware scheduling decisions about the level of task granularity (or parallelism) appropriate for its subgraph. Agile executors can scale out compute resources in the face of burst-parallel workloads by partitioning their subgraphs into smaller graphs that are assigned to other executors for an even higher level of parallel task scheduling and execution. Alternately, an executor can execute tasks locally, when the cost of data communication between the tasks outweighs the benefit of parallel execution.

Automatic resource elasticity and improved cost effectiveness: Decentralization does not require users to explicitly tune the number of active Lambdas running as workers and thus is easier to use, more cost effective, and more resource efficient.

We make the following contributions in this paper.

- We thoroughly explore the problem space of serverless parallel computing framework

design. For a range of parallel computation applications, we identify issues of the state-of-the-art serverless frameworks—task scheduling, data locality, and resource efficiency (monetary cost effectiveness).

- We present the design and implementation of WUKONG, a new serverless parallel computing framework that solves the identified issues. WUKONG synergizes a set of optimization techniques, including decentralized scheduling, task clustering, and delayed I/O. These techniques together achieve near-ideal scalability, reduce data movement over the network, enhance data locality, and improve cost effectiveness.
- We evaluated WUKONG extensively on AWS. Our results show that WUKONG reduces network I/O by many orders of magnitude and achieves up to 68.17× higher performance than numpywren, while reducing the monetary cost by as much as 92.96%.

4.3 Background and Motivation

4.3.1 Why Serverless?

Serverless Computinghandles virtually all system administration tasks, making it easier for developers to use a near-infinite amount of cloud resources, including bundled CPUs and memory, object stores, and a lot more [4]. Service providers provide a flexible interface for defining serverless functions, which allows developers to focus on core application logic. Service providers in turn auto-scale function executions in a demand-driven fashion, hiding tedious server configuration and management tasks from the users.

General Constraints and Limitations

Service providers place limits on the use of cloud resources to simplify resource management. Take AWS Lambda for example: users configure Lambda's memory and CPU resources in a bundle. Users can choose a memory capacity between 128MB–3008MB in 64MB increments. Lambda will then allocate CPU power linearly in proportion to the amount of memory configured. Each Lambda function can run at most 900 seconds and will be forcibly stopped when the time limit is reached. In addition, Lambda only allows outbound TCP network connections and bans inbound connections and the UDP protocol.

Opportunities

Running large-scale, burst-parallel computation jobs has long been challenging for domain scientists due to the complexity of configuring, provisioning, and managing compute clusters [54, 55]. By taking over system administration and automatically providing capability to launch thousands of processes with no advance notice, the emerging serverless computing model *seems* to provide a foundation that will attract domain scientists and data analysts. *However, to fully unleash the potential of serverless computing, an efficient serverless-optimized parallel computing framework is needed.*

4.3.2 Challenges

We build on our experience with serverless frameworks to synthesize the performance requirement for an ideal, serverless, parallel computing framework, and discuss why current solutions are not able to meet these performance requirements of burst-parallel applications at both the task scheduling and the data locality level.

Challenge to Rapidly Scale Out

A family of burst-parallel computation jobs (e.g., data analytics [30], machine learning classifications [50], etc.) are dominated by short-lived tasks with a span ranging from hundreds of milliseconds (ms) to tens of seconds [10, 30, 40, 51]. Such applications pose a difficult scheduling challenge to the serverless computing platforms. This is because, while serverless computing promises to deliver elastic auto-scaling feature in response to bursts of concurrent workloads, serverless function invocations incur non-negligible overhead, thus creating a scheduling bottleneck with slow scaling out.

PyWren is a state-of-the-art serverless execution engine [10]. PyWren enables users to program MapReduce-like applications on serverless platforms such as AWS Lambda. Numpywren [9] is a system for linear algebra built on top of PyWren. Numpywren uses PyWren's existing infrastructure to deploy their own serverless task executors, which run as a user-defined function within PyWren's own Lambda executors. In order for numpywren to scale the size of their Lambda cluster, numpywren invokes additional PyWren executors (using PyWren's own API) from their central scheduler. Based on this design, numpywren relies heavily on PyWren for Lambda scaling and management.

Figure 4.2 shows PyWren's ability to schedule large numbers of no-op tasks on AWS Lambda-based executors. Ideally, an omniscient serverless scheduler should be able to take full advantage of the massive parallelism offered by serverless computing and rapidly scale to thousands of Lambda executors in seconds in response to bursty, highly parallel workloads. PyWren uses a centralized scheduling approach, where it employs 64 threads for task scheduling and invocation; it takes almost 2 minutes to scale out to 10,000 Lambda

executors[3]. To make it worse, a serverless framework like PyWren cannot always keep thousands of Lambda executors actively running (unlike the worker servers to a typical serverful parallel framework such as MapReduce [42]), so it has to constantly invoke many Lambdas in response to bursts of job tasks. *This serves to illustrate the failure of existing serverless execution frameworks to fully utilize serverless computing's elastic auto-scaling property.*

Challenge of Excessive Data Movement

Parallel applications require intermediate data exchange among tasks. Direct task-to-task data communication is naturally supported in traditional serverful parallel computing frameworks such as MPI [54], MapReduce [42], and Dask [44]. However, data exchange in serverless applications may be supported only indirectly, through the use of remote cloud storage systems.

Consider the data exchanged during the execution of $25k \times 25k$ GEMM (general matrix multiplication) and $8,192k \times 128$ TSQR (tall skinny QR) on numpywren. Figure 4.3 and Figure 4.4 present a comparison between pure input and output sizes and the amount of data transferred during the two workloads respectively. For GEMM, the total quantity of data read is more than $25\times$ the size of the input data while the total quantity of data written is more than $20\times$ the size of the output. This trend is further exemplified by TSQR. While the amount of data read is only $2.88\times$ greater than the input data size, the amount of data written over $65M\times$ greater than the output size. This is because numpywren and PyWren adopt a stateless Lambda executor design where a task can be dispatched to any Lambda executor; once dispatched to a Lambda, the task simply performs the following four steps: 1) reads its input data (the intermediate data generated by one or multiple upstream tasks) from cloud storage (numpywren uses S3), 2) performs computation, 3) writes the intermediate results (as output) to the cloud storage, and 4) returns. While the stateless design seems to be a good fit for serverless platforms, it does not preserve data

[3]As did in [14], we also performed warmup operations to make sure each Lambda invocation does not incur a cold start [56].

locality, which results in excessive data movement. *This stresses a strong need for reducing data movement and increasing data locality in serverless frameworks.*

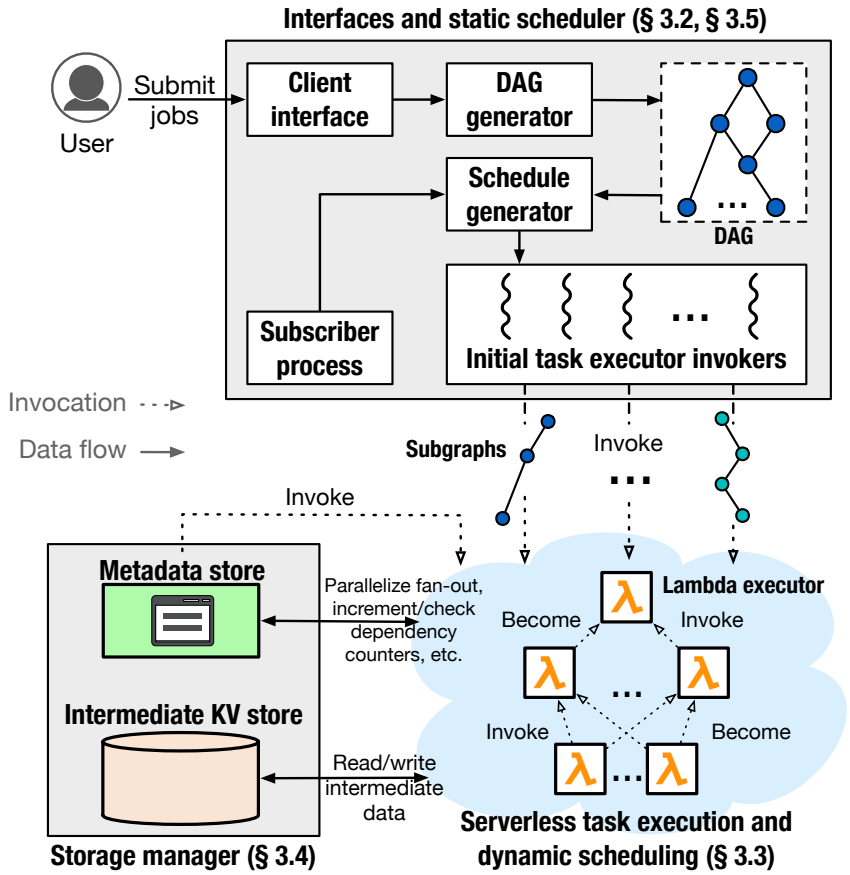


Figure 4.5: Overview of WUKONG architecture.

4.4 Wukong Design

In this section, we present the system design of WUKONG. This design is motivated by the observations that existing serverless parallel frameworks are slow to scale out and are bottlenecked by excessive data movement (section 4.3).

4.4.1 High-Level Design

Figure 4.5 shows the high-level design of WUKONG. The design consists of three major components: a static schedule generator which runs on Amazon EC2, a pool of Lambda Executors, and a storage cluster.

Scheduling in WUKONG is decentralized and uses a combination of static and dynamic

scheduling. A static schedule is a subgraph of the DAG. Each static schedule is assigned to a separate Executor. An Executor uses dynamic scheduling to enforce the data dependencies of the tasks in its static schedule. An Executor can invoke additional Executors to increase task parallelism, or cluster tasks to eliminate any communication delay between them. Executors store intermediate task results in an elastic in-memory key-value storage (KVS) cluster (hosted using AWS Fargate [57]; see section 4.4.4) and job-related metadata (e.g., counters) in a separate KVS that we call metadata store (MDS).

4.4.2 Static Scheduling

WUKONG users submit a Python computing job to the DAG generator, which uses the Dask library to convert the job into a DAG. The Static-Schedule Generator generates *static schedules* from the DAG. For a DAG with n leaf nodes, n static schedules are generated. A static schedule for leaf node `L` contains all of the task nodes that are reachable from `L` and all of the edges into and out of these nodes. The data for a task node includes the task's code and the KVS keys for the task's input data. The schedule for `L` is easily computed using a depth-first search (DFS) that starts at `L`. Lambda Executors notify the static scheduler when a final result has been stored in Redis by sending a message to the static scheduler's subscribe process. Upon receiving a message, the static scheduler will download final results and return them to the user automatically.

Figure 4.6(a) shows a DAG with two leaf nodes. Figure 4.6(b) shows the two static schedules that are generated from the DAG: `Schedule 1` (blue) and `Schedule 2` (green).

A static schedule contains three types of operations: task execution, fan-in and fan-out. To simplify our description, when DAG task `Tx` is followed immediately by task `Ty`, and `Tx (Ty)` has no fan-out (fan-in), we add a trivial fan-out operation between `Tx` and `Ty` in the static schedule. This fan-out operation has one incoming edge from `Tx` and one outgoing edge to `Ty`, i.e., there is no actual fan-out. In Figures 4.6(a) and (b), this is the case for DAG tasks `T2` and `T3`.

A fan-in task `T` may depend on tasks that will be executed by different Executors, e.g.,

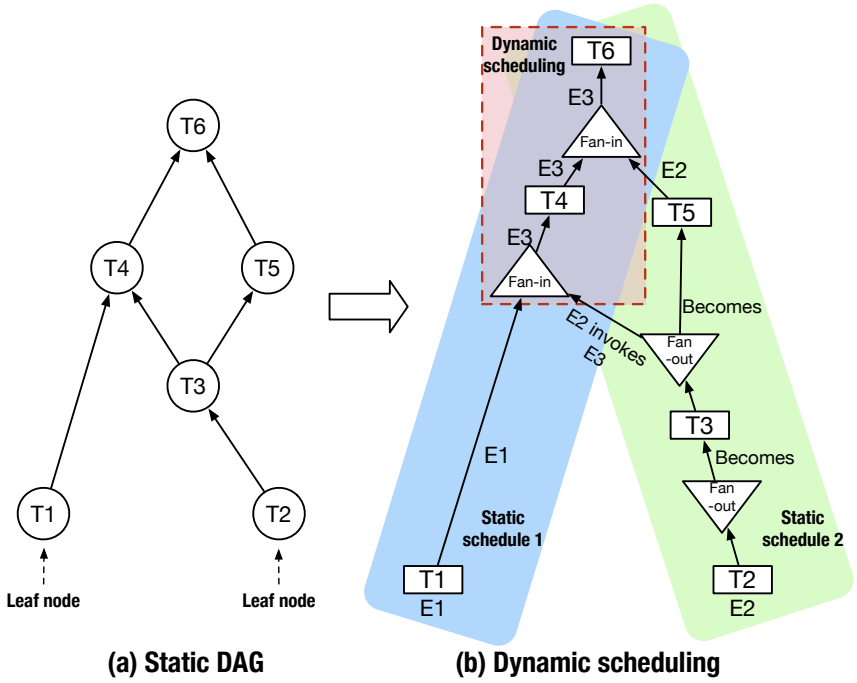


Figure 4.6: Static DAG (a) and dynamic scheduling (b). WUKONG's Executors coordinate in the area inside the dashed box in (b) using dynamic scheduling. "T1" denotes Task 1. "E1" denotes Lambda Executor 1.

task `T4` in Figure 4.6. The dynamic scheduling technique described below ensures that `T`'s data dependencies are satisfied and that `T` is executed by only one Executor. Note also that a static schedule does not map its tasks to processors; this mapping is done automatically by the AWS Lambda platform when an Executor function instance is invoked with the static schedule and placed on a VM by AWS Lambda.

4.4.3 Task Execution & Dynamic Scheduling

Task Execution. WUKONG workflow execution starts when the static scheduler's Initial-Executor Invokers assign each static schedule produced by the Static-Schedule Generator to a separate Executor. Recall that each static schedule begins with one of the leaf node tasks in the DAG. The Initial-Executor invokes these "leaf node" Executors in parallel. After executing its leaf node task, each Executor then executes the tasks along a single path through its schedule. An Executor may execute a sequence of tasks before it reaches

a fan-out operation with more than one out edge or it reaches a fan-in operation. *For such a sequence of tasks, there is no communication required for making the output of the earlier tasks available to later tasks for input.* All intermediate task outputs are cached in the Executor's local memory with inherently enhanced data locality. Executors also look ahead to see what data their tasks may need, which allows them to discard data in their local memory that is no longer needed.

Furthermore, Executors can increase parallelism by scheduling and invoking new Executors to execute the task targets of a fan-out. A group of Executors that reach a common fan-in node decrease parallelism by scheduling one of them to execute the fan-in task and the rest to stop. WUKONG uses dynamic scheduling to resolve conflict on the fly. More importantly, *this dynamic scheduling of the tasks in an Executor's static schedule leads to a decentralized scheduling model that naturally fits serverless computing, eliminates the need for a centralized scheduler processes that check data dependencies and invoke ready tasks with improved scalability.*

Dynamic Scheduling for Fan-Out Operations. For fan-out operations there are two cases:

Case 1: none of the n (where $n > 1$) fan-out edges is a fan-in edge. Then `E` "becomes" the Executor for one of the fan-out's tasks, say `T`, by executing `T`, and `E` "invokes" an Executor for the other fan-out tasks.

Case 2: one or more of the fan-out edges is also a fan-in edge. For example, fan-out node 3 in Figure 4.6(a) has a fan-out edge that is a fan-in edge to node 4. The selection of a "becomes" edge for `E` is based on the immediate availability of the tasks targeted by the fan-out edges. If no task target's dependencies are satisfied then no task target is immediately available for execution and none of the fan-out edges can be selected as `E`'s "becomes" edge (the fan-in edges have fan-in operations that will be executed next); otherwise, one of the fan-out edges for the available target tasks is selected as the "becomes" edge.

An intermediate objects needed for input by an invoked Executor is passed to the Executor as an argument if the size of the object is less than the maximum allowed argument

size (256K);otherwise, the object is sent to the Storage Manager, and the associated KVS keys are passed to the invoked Executors as arguments.

Each of the $n-1$ Executors invoked by `E` is assigned a static schedule that begins with one of the $n-1$ fan-out edges. Each of these (possibly overlapping) static schedules corresponds to a sub-graph of `E`'s static schedule. Executor `E` continues task execution and scheduling along the remaining fan-out edge and executes the operation encountered on this edge.

In Figure 4.6(b), fan-out edges are labeled either "invokes" or "becomes" to indicate whether the Executor invokes a new Lambda Executor to execute a fan-out task or executes the fan-out task itself.

Since Executor invocations, which are in the form of AWS Lambda function invocations, incur a high overhead (e.g., invoking an AWS Lambda function takes about 50 milliseconds with the Boto3 AWS Python API), we use a number of dedicated Executor-Invoker processes that are co-located with the Static Scheduler (Figure 4.5). When an Executor performs a fan-out operation, and a large number of new Executors must be invoked, the Executor delegates the invocations to the Static Scheduler. The Static Scheduler evenly distributes task invocation responsibilities among the Invoker processes, enabling (near-)linear speedup over sequential invocations.

Dynamic Scheduling for Fan-in Operations. If Task Executor `E` executes a fan-in operation with n (where $n > 1$) in-edges, then `E` and the $n-1$ other Executors involved in this fan-in operation cooperate to see which one of them will continue their static schedules on the out edge of the fan-in (e.g., node 4 in Figure 4.6).

For a fan-in operation executed by `E` for fan-in task `T`, `E` atomically gets and updates a value in the KVS that tracks the number of `T`'s input dependencies that have been satisfied during execution There are two cases:

Case 1: all of the input dependencies of `T` have been satisfied. Then E continues its static schedule by executing `T`

Case 2: all of the input dependencies of `T` have not been satisfied. Then `E` sends the

intermediate object needed by T to the Storage Manager.

In Figure 4.6(b), each fan-in task is labeled with the Executor that executed the task.

Task Clustering. Storing and retrieving large intermediate objects can be very costly. WUKONG implements task clustering to avoid large object storage.

Task Clustering for Fan-Out Operations. If the output object of some fan-out task T executed by Executor E is larger than a user-defined threshold t (e.g., 200 MB), E will try to cluster the target tasks of T's fan-out edges, i.e., E will execute the target tasks whose dependencies are satisfied instead of invoking new Executors for these tasks. For example, the Executor that executes task C can also execute tasks F and G to avoid the time and cost of communicating task C's large object output to tasks F and G. In cases like this, the fan-out edges will have multiple edges labeled "becomes".

Task Clustering for Fan-In Operations. If E executes a fan-in operation for task T and the input dependencies of a single task T have not been satisfied, then E sends the intermediate object needed by T to the Storage Manager. If this object is large, E then rechecks T's input dependencies. If T's input dependencies became satisfied while the large intermediate object was being stored, E becomes the Executor for T, This avoids the communication delay that would have occurred when T retrieved the large object from storage, but not the delay that occurs when R stored the object. The delay that occurred when the large object was stored by E can be avoided if the storage operation can be delayed until after T's input dependencies become satisfied.

Suppose that when Executor E executes the fan-out operation it identifies multiple fan-out tasks that are also fan-in tasks and that have input dependencies that are not satisfied. E will then execute any ready fan-out tasks it finds and delay the decision about how to handle the fan-out tasks that are not ready. (There may be many fan-out tasks that are not also fan-in tasks or that are fan-in tasks with satisfied input dependencies.)

Delayed I/O. After Executor E executes the ready tasks, E will recheck whether the input dependencies of the unready tasks have since become satisfied. If so, the newly ready tasks can be executed. If at least one unready task becomes ready, the ready task can be executed

and the unready tasks can be rechecked again, and so on, for a configurable number of times. Our profiling indicates that it is almost always better to wait until all of the unready tasks become ready to avoid the very large communication delay the occurs when many large objects are stored and retrieved potentially at the same time. A pattern of simultaneous large object writes and reads occurred often in the DAG workloads that we executed. If unready tasks remain when this process stops, `E` must send the intermediate objects needed by the unready tasks to storage; however, as described above, `E` can possibly avoid the communication delay associated with retrieving the objects from storage by checking the unready objects one more time.

4.4.4 Storage Management

WUKONG's Storage Cluster is built atop AWS Fargate [57], a serverless container engine that can be elastically scaled out/in. The Storage Cluster includes a number of AWS Fargate tasks, each of which hosts a Redis instance, and a KV Store Proxy Service. The KV Store Proxy is executed within an EC2 VM along with an additional instance of Redis used exclusively for storing static schedules and dependency counters. The user can configure the size of the Fargate cluster to dynamically accommodate workloads of different sizes. The user simply specifies how many nodes they would like, and WUKONG ensures that these nodes are created and available. The Fargate nodes are used for the storage of intermediate data generated during a workload's execution. We opt to use Redis instead of S3 as Redis can provide both high bandwidth for large object workloads and high IOPS for small object workloads [8, 58], whereas S3's IOPS is throttled. However, modifying WUKONG to use S3 is trivial.

The KV Store Proxy performs various storage operations on behalf of the Task Executors and the Scheduler. At the start of a workflow, the Storage Manager receives the workflow DAG and the static schedules derived from the DAG from the Scheduler.

Intermediate and Final Result Storage. Task Executors publish their intermediate and final task output objects to the KV Store. Final outputs are relayed to a Subscriber process in the Scheduler for presentation to the Client.

```
1  def add(x,y):
2    time.sleep(0.5)
3    return x + y
4
5  L = range(8)
6  while len(L) > 1:
7    L = list(map(delayed(add),\
8             L[0::2], L[1::2]))
9
10 L[0].compute()
```

Figure 4.7: TR code.

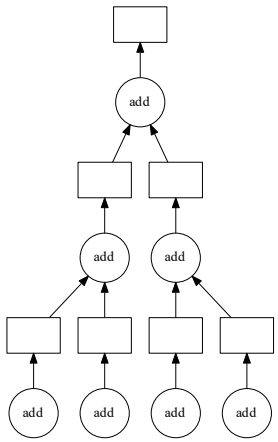


Figure 4.8: TR DAG.

Small Fan-out Task Invocations. When a Task Executor performs a fan-out operation that has a small number of out edges, the Task Executor makes the necessary Executor invocations itself.

Large Fan-out Task Invocations. When a fan-out has a number of out edges that is larger than a user-specified threshold, the Task Executor publishes a message that is relayed to a Subscriber process in the Storage Manager that then passes the message on to the Proxy. This message contains an ID that identifies the fan-out's location in the DAG. The Proxy uses the DAG and the fan-out ID to identify the fan-out's out edges in the DAG. This allows the Proxy, with the assistance of the Fan-out Invokers in the Storage Manager, to make the necessary Task Executor invocations, in parallel. The Proxy passes to each Executor its intermediate inputs (or their key values in the KV Store) and the Executor's static schedule.

4.4.5 Programming Model

WUKONG reuses Dask's Python programming interfaces [59] (including high-level APIs, such as Dask libraries and data structures, and low-level APIs, such as `Dask.delayed`) for implementing parallel programs. In general, any code written for Dask should execute

on WUKONG. Figure 4.7 shows an example code snippet that implements tree reduction (TR) (a detailed description of TR is in section 4.5.1). WUKONG also reuses Dask's DAG generator which translates high-level Python code into a DAG [60]. Figure 4.8 depicts the DAG generated for TR with an 8-element array.

4.4.6 Fault Tolerance

For fault tolerance, we relied on the automatic retry mechanism of AWS Lambda, which allows for up to two automatic retries of failed function executions. Investigating better fault tolerance scheme is part of our future work.

4.5 Evaluation

Implementation. We have implemented WUKONG using roughly $12k$ lines of Python code ($5,349$ LoC for the AWS Lambda Executor Runtime, $3,057$ LoC for the Storage Manager, and $3,577$ LoC for the Static Scheduler). We use the Dask library [44] to generate DAGs to use as the input computation graph for WUKONG.

4.5.1 Experimental Goals and Methodology

Our evaluation was performed on AWS. The static scheduler ran in an `r5n.16xlarge` EC2 VM. We used a scale-out Redis cluster (Multi-Redis) as WUKONG's intermediate storage. The Fargate nodes within the storage cluster were each configured to have 30 GB of memory and 4 vCPUs. For the Multi-Redis setup, WUKONG used 75 nodes for the storage cluster, as we've determined this value to be both performant and cost-effective for many workloads. The MDS proxy was co-located on the same `r5n.16xlarge` VM as a Redis instance that was used for storing static schedules and dependency counters. Each Lambda function was allocated 3 GB of memory and a maximum execution time of seven minutes.

We compared the performance of WUKONG against both Dask distributed (which we refer to simply as "Dask") and numpywren [9] / PyWren [10]. We chose to compare our performance against Dask as both WUKONG and Dask use the exact same input DAG and the same algorithms for their computations. We compared the end-to-end performance of WUKONG against numpywren, and we compared the scalability of WUKONG against PyWren, which is numpywren's underlying Lambda execution framework.

We chose to compare WUKONG against numpywren because both are serverless DAG execution frameworks; however, there are significant differences between WUKONG and numpywren. One difference is that WUKONG uses Dask DAGs, which explicitly encode tasks and their dependencies. Numpywren, on the other hand, uses an implicit DAG representation for its programs, which are all implemented in the LAmbdaPACK language for linear algebra [9]. The nodes of the DAG that represent a LAmbdaPACK program are generated on-demand (at runtime).

Another difference is that numpywren uses AWS S3 as its intermediate data store. In order to make the comparison between WUKONG and numpywren more fair, we modified numpywren to use a single instance of Redis as its intermediate object store. We compared this version of numpywren, which we refer to as "Numpywren Single Redis", with a modified version of WUKONG that also uses a single instance of Redis for intermediate data storage. In addition, we compared numpywren S3 against "WUKONG Multi-Redis", which uses a Fargate Redis cluster for its intermediate object store. This second comparison was performed in order to show the optimal performance achieved by each system.

Finally, numpywren only supports linear algebra algorithms and only several of these algorithms are also available in Dask. As a result, we are limited in which workloads we can run on both WUKONG and numpywren.

To better understand WUKONG's performance, we compared WUKONG against 2 different Dask configurations. Both configurations used the same amount of CPU (2,000 cores) and memory (3,000 GB memory). This was the largest VM cluster that we could configure. The first configuration consisted of 1,000 2-core 3GB workers running on 125 `c5.4xlarge` 16-core 32 GB VMs. Each VM had eight workers running on it. The second configuration consisted of 125 workers; each worker exclusively ran on a `c5.4xlarge` VM and was allocated 16 cores and 24 GB memory. The first configuration was selected so that each worker was approximately as powerful as the AWS Lambda functions used by WUKONG; it represents a *worst-case scenario* that stresses the Dask scheduler with many workers and incurs high communications due to the lack of data locality, emulating a serverless environment with centralized scheduling. The second configuration represents a *best-case scenario* where the relatively more powerful workers could process larger data with improved data locality and significantly reduced communications.

Additionally, we used the exact same input data partitions for Dask and WUKONG. We scaled the problem size for each benchmark by having each (Dask, WUKONG, or numpywren) worker assigned with (at least) one partition of the input data. As such, a test used only a fraction of the 2,000 cores of the Dask cluster, until the problem size was large enough

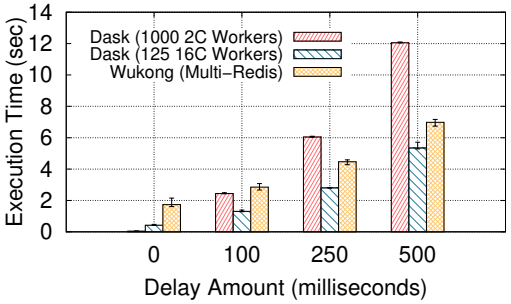


Figure 4.9: TR.

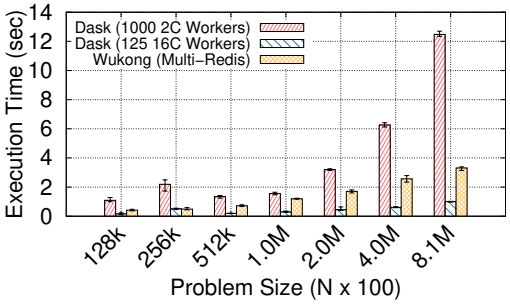


Figure 4.10: SVD1.

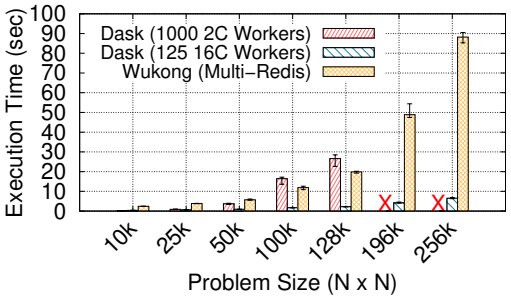


Figure 4.11: SVD2.

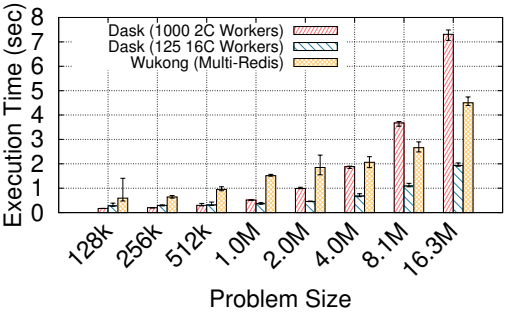


Figure 4.12: SVC.

to occupy all the resources. The largest number of partitions (of input data) used during our evaluation was $4,096$ for 16 M TSQR (which will be described shortly). This number of partitions does not overwhelm either Dask configurations as TSQR's DAG features large fan-ins in the middle of the job. The large fan-ins result in the number of tasks allocated to each worker decreasing as the workload progresses.

In our evaluation results, each data point is the average of *ten* runs. The error bars on the graphs of the results indicate the biggest and smallest results obtained. WUKONG is easy-to-use as it exposes only two configuration knobs to the end users—the input partition size and the number of Fargate nodes. (A sensitivity analysis of these two configuration knobs is omitted due to space constraint.)

We evaluated the following parallel applications.

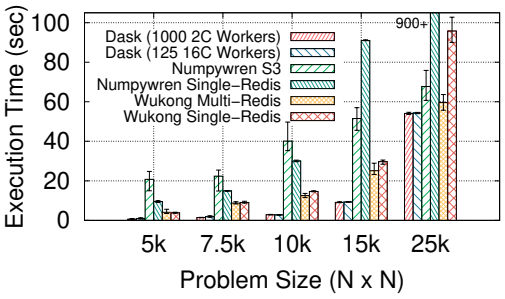


Figure 4.13: GEMM.

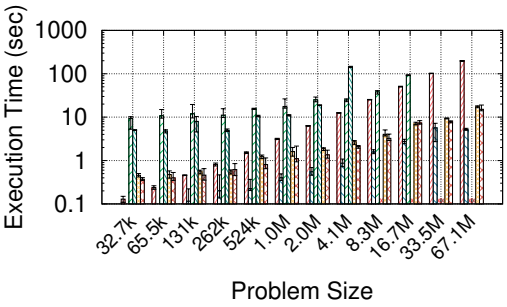


Figure 4.14: TSQR (log-scale).

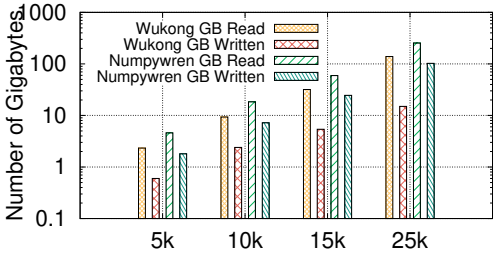


Figure 4.15: GEMM I/O (log).

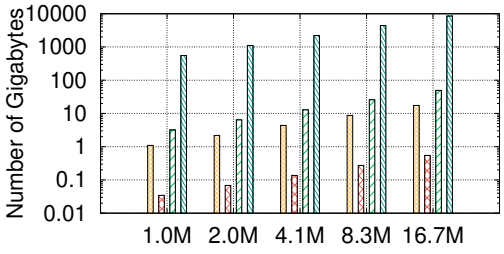


Figure 4.16: TSQR I/O (log).

Tree Reduction (TR): TR sums the N elements of an array using a total of $N - 1$ operations performed over multiple passes. Each pass adds adjacent elements, in parallel, until after $log(N)$ passes only a single element remains. TR serves as a microbenchmark for evaluating the effect of task granularity and parallelism on performance. See Figure 4.7 and Figure 4.8 for an example of the Python code snippet and generated DAG.

Singular Value Decomposition (SVD): We evaluated two variants of SVD. The first variant (SVD1) computes the SVD of a tall skinny matrix. The second variant (SVD2) computes the SVD of a square (i.e., $n \times n$) matrix using an approximation algorithm provided by [45]. Note that Dask's SVD algorithm differs considerably from numpywren's, and thus a direct comparison between WUKONG and numpywren for SVD is impossible. For reference, WUKONG can execute SVD $256k \times 256k$ in 88 seconds on average while [9] reports an average of 77,828 seconds for the same problem size for SVD.

Support Vector Classification (SVC): SVC is a machine learning workload. The benchmark we used is publicly available in the Dask-ML documentation [46].

General Matrix Multiplication (GEMM): GEMM performs matrix multiplication, an important component of many linear algebra algorithms.

Tall-and-Skinny QR Factorization (TSQR): This workload performs a QR factorization of a tall skinny matrix.

4.5.2 End-to-End Performance Comparison

TR. The size of the array used for TR was 1024. We also intentionally added a delay to each task of TR. This delay simulates an increased amount of work performed per task. We varied the amount of delay between 0–500 ms. Figure 4.9 shows that both configurations of Dask outperforms WUKONG by a large margin for the base case. This is because Dask uses TCP to dispatch the $1024/2 = 512$ `addition` tasks to workers, whereas for WUKONG the overhead of scaling out to 512 Lambda executors dominates. As we add increasing amounts of delay to each task, the performance gap between Dask and WUKONG decrease. Once the delay is 250 ms or more, WUKONG executes the workload faster than the $1,000$-worker Dask

cluster, as WUKONG uses decentralized scheduling to rapidly scale out to 512 workers. This experiment shows an interesting tradeoff between per-task execution time and serverless scaling cost – WUKONG performs best when each task performs a non-trivial amount of work, as this compensates for the overhead of spinning up additional Lambdas.

SVD. We tested seven problem sizes for both SVD1 and SVD2 (see Figure 4.10 and Figure 4.11). For nearly all problem sizes, WUKONG out-performed the $1,000$-worker Dask cluster, completing the job anywhere from 62.02% to 69.09% faster than Dask. This performance difference results from the benefits of WUKONG's decentralized scheduling techniques, which greatly reduce the overhead of executing tasks on a large number of workers. The $1,000$-worker Dask was bottlenecked by its central scheduler due to the increasing load from the one thousand workers.

The 125-worker Dask cluster consistently outperformed WUKONG. This is because each Dask worker is provisioned with more resources (i.e., more CPUs, greater network bandwidth, a larger amount of RAM per worker, etc.), which significantly increases data locality and reduces cross-worker communications. More importantly, for SVD2, WUKONG was able to scale to considerably larger problem sizes than what the 1,000-worker Dask cluster was capable of handling (crosses in Figure 4.11). The results demonstrate WUKONG's ability to provide competitive performance with traditional serverful frameworks, while also scaling effectively for increasingly large problem sizes. WUKONG's ability to scale effectively here is largely because of its use of task clustering and delayed I/O. These techniques dramatically reduce data movement and ensure all downstream tasks which depend on large data are executed on the Task Executor that already has the data in-memory. A quantitative analysis of the benefits of these techniques is given later in section 4.5.5.

SVC. Figure 4.12 shows SVC's performance. As with the previous benchmark, Dask was able to perform better for smaller problem sizes; however, when we increased the problem size, the performance gap between the two frameworks decreased. As the scale of the problems increased to $4.0M$ samples and beyond, WUKONG began to scale more effectively than the $1,000$-worker Dask cluster. The large parallelism of WUKONG enabled the framework to

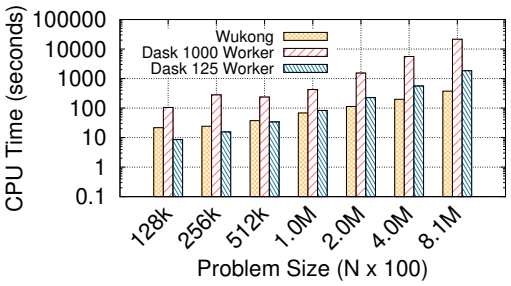


Figure 4.17: SVD1 CPU time.

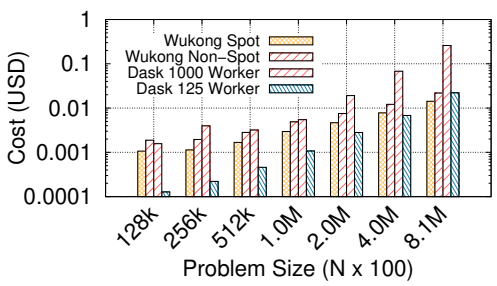


Figure 4.18: SVD1 cost.

complete the workload in 46.45% less time; however, the 125-worker Dask cluster executed the workload roughly 50.56% faster than WUKONG. This is likely due to both the higher network bandwidth and increased data locality of the Dask workers.

GEMM. Like TR, the results of our GEMM experiments identify a workload that is difficult to execute in a serverless environment. As shown in Figure 4.13, WUKONG achieved worse performance than Dask. This is because GEMM natively requires a number of large data objects to be communicated between tasks before the computation can begin. Due to the limited bandwidth available to the intermediate data storage, both WUKONG and numpywren experience long delays during this phase of the workload.

WUKONG's performance greatly exceeded that of both numpywren configurations for all problem sizes. For the largest problem size, WUKONG (single Redis shard) executed the workload 89.76% faster than numpywren (single Redis shard). WUKONG (with Fargate) was 51.51% faster than numpywren (with S3) for $15k \times 15k$ matrices because WUKONG reduced the amount of data read from and written to the intermediate KVS, and this reduction on I/Os directly translates to performance improvement. As shown in Figure 4.15, WUKONG read 49.39% less data than numpywren for the smallest problem size and 45.24% for the largest; WUKONG reduced the amount of data written by as much as 85% for the largest problem size.

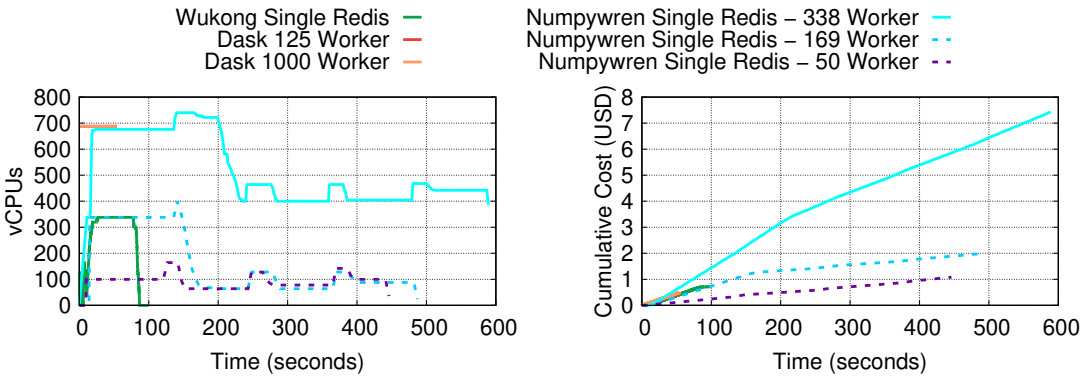
TSQR. Figure 4.14 shows that WUKONG outperformed both numpywren (with S3) and

numpywren (single Redis shard) for all problem sizes. For the $4.1M \times 128$ matrix, WUKONG (single Redis shard) was executing the workload 68.17× faster, or in 98.53% less time, than numpywren (single Redis shard); and WUKONG (Fargate) is 9.19× faster, or in 89.11% less time, than numpywren (S3). Numpywren (single Redis shard) failed to execute the next larger problem size, and the largest workload numpywren (S3) was able to execute was $16.7M \times 128$. For this workload, WUKONG was 13.36× faster, executing the workload in 92.51% less time. This again, is because of the significantly reduced reads and writes. WUKONG's "become" functionality allowed serial tasks along a single subgraph path to execute locally on the same Lambda Executor; whereas numpywren randomly assigned high priority tasks (which were ready to execute) to any stateless Lambda executor. As a result, numpywren wrote $16,027\times$ more data for the smallest problem size and $15,631\times$ more data for the largest problem size, which resulted in dramatic performance degradation (Figure 4.16).

4.5.3 CPU Time and Monetary Cost

Figure 4.17 presents a comparison between WUKONG (Multi-Redis) and both Dask clusters on their total CPU time (core seconds) for SVD1. Note that this comparison only considers cores actively used by Dask for each problem size. WUKONG uses more core seconds than Dask-125 for the first three problem sizes, as Dask-125 finishes the job significantly faster than WUKONG. For $1.0M$ and above, WUKONG uses less core seconds than Dask-125. Dask-1000 incurs the longest CPU time as it is the slowest of the three. The disparity between the two frameworks increases as the problem size grows.

Figure 4.18 presents a comparison of the monetary cost to execute SVD1 for varying problem sizes. This cost analysis only considers the Dask VMs actively used for each given workload size. At first, WUKONG is more expensive than Dask-125. As the problem size increases, the cost of executing the workload on WUKONG increases at a much slower rate than the cost of running the workload on Dask-125. By the largest problem size, the price



(a) CPU usage comparison.

(b) Cost comparison.

Figure 4.19: GEMM CPU usage and cost timeline.

of executing the workload on WUKONG is equal to that of executing the workload on Dask-125. Additionally, WUKONG achieves faster performance, lower cost, and more efficient CPU usage than Dask-1000 using non-spot pricing for all except the smallest problem size. These results demonstrate WUKONG's pay-per-use property.

Figure 4.19 shows a comparison between various configurations of WUKONG Single Redis, Dask, and numpywren Single Redis on the number of vCPUs and cumulative cost used during GEMM $25k \times 25k$. These configurations include both Dask configurations and three configurations of numpywren Single Redis. By design, numpywren allows users to specify the initial[4] number of Lambda executors (workers) for a job. We configured numpywren to use 50, 169, and 338 workers. Numpywren-169 was selected because the maximum concurrency achieved by WUKONG during this workload was 169 Lambdas. We tested a scaled-out version of numpywren that used twice as many starting workers (338) as well as a scaled-down version using 50 workers. Notably, numpywren-50 was the fastest configuration, followed by numpywren-169 and finally numpywren-338. This suggests that increasing the number of numpywren's parallelism significantly increases contention, possibly at the framework's central queue or scheduler, leading to performance degradation.

[4]One can configure the starting number of executors, a maximum number of executors, and the policy used to scale the number of executors dynamically during execution. We opted to select the default scaling policy for all numpywren runs.

Figure 4.20: TSQR CPU usage and cost timeline.

WUKONG was both cheaper and used less vCPUs during the execution of the workload than all three numpywren configurations. Specifically, WUKONG was 33.47% cheaper and 77.57% faster than the best-performing numpywren configuration. More importantly, WUKONG is autonomous and does not require users to explicitly tune the parallelism, which improves the usability.

Similarly, Figure 4.20 shows a comparison for TSQR $4.0M$. The first configuration of numpywren used 128 workers while the second used 256 workers. These are based on the number of leaf tasks in WUKONG's workload (512), though we found that using 512 numpywren workers resulted in worse performance. WUKONG used less CPU resources and was significantly cheaper than all other frameworks. It also out-performed all other frameworks except for Dask-125. Specifically, WUKONG completed the workload in 87.41% less time and 92.96% more cheaply than the best-performing numpywren configuration (i.e., 14.22× cheaper and 7.94× faster). Lastly, while WUKONG did not out-perform Dask-125, WUKONG was 95.67% cheaper than Dask-1000 and 45.70% cheaper than Dask-125 for this workload. WUKONG did not reach more than 106 vCPUs during the workload's execution as many of the leaf tasks performed only a small amount of work before writing their data to the KVS and exiting. That is, by the time additional leaf tasks are scheduled,

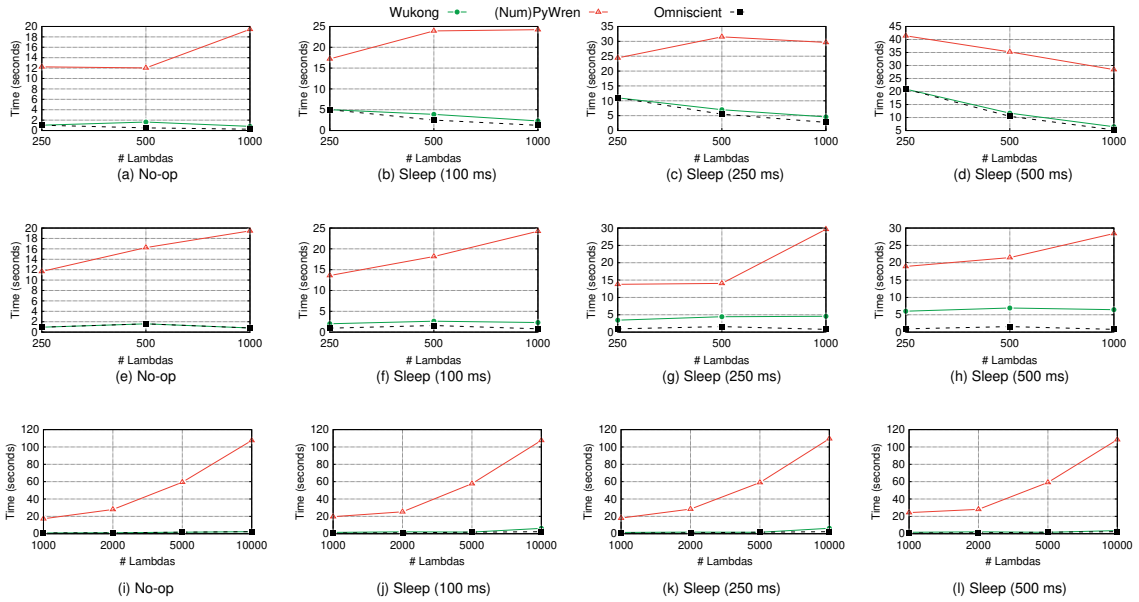


Figure 4.21: Figure 4.21(a)-(d) – strong scaling: time (Y-axis) to execute 10,000 tasks over N Lambda executors (X-axis). Figure 4.21(e)-(h) – weak scaling: time to execute 10 tasks per Lambda. Figure 4.21(i)-(l) – serverless scaling: time to execute N task on N Lambda. For each row, plots are for (from left to right) tasks of 0, 100, 250, and 500 ms.

previously-invoked leaf tasks were already finishing their execution, forming a scheduling pipeline.

4.5.4 Elastic Scaling

We next evaluate WUKONG's scalability on traditional strong / weaking scaling and serverless scaling metrics, and compare it against (Num)-PyWren. The maximum concurrency we got from Amazon was 5,000 concurrent Lambdas. In this experiment, for both (Num)-PyWren and WUKONG, all the executors in the Lambda pool got warmed up before accepting task requests, eliminating the cold start concerns. The Lambda pool scaled from zero.

Strong Scaling. For strong scaling, we varied the number of Lambdas to be launched to execute 10,000 tasks. In order to simulate workloads with various computation loads, we added a delay of 100 ms, 250 ms, and finally 500 ms to each task. Each test was repeated three times. Figure 4.21(a)–(d) show that, in all cases, WUKONG exhibited near-ideal

strong scaling behavior, scaling significantly better than (Num)-PyWren in all cases, thereby demonstrating the effectiveness of our decentralized scheduling mechanism. It was not until the 500 ms delay case that (Num)-PyWren exhibited a downward trend in execution time as the number of Lambda executors scaled. This is because: 1) 500-ms tasks tend to run longer, and 2) with more Lambda executors and a fixed total amount of tasks, each Lambda gets assigned less number of tasks.

Weak Scaling. For weak scaling, we executed ten tasks per worker and varied the number of Lambda executors from 250 to $1,000$. As shown in Figure 4.21(e)–(h), WUKONG exhibited near-ideal weak scaling behavior for all sleep delays and worker configurations. Notably, thanks to the decentralized scheduling, WUKONG was able to scale to $1,000$ Lambda executors much more effectively than (Num)PyWren, which experienced increasingly large delays as the number of Lambdas increased.

Serverless Scaling. Finally, we test serverless scaling – executing N tasks on N Lambda executors, with each Lambda effectively executing one single task. Figure 4.21(i)–(l) plots the results. We observe that (Num)-PyWren took an increasing amount of time to finish executing N tasks on N Lambdas, and executing $10,000$ tasks took almost two minutes. In contrast, WUKONG rapidly scales out to $10,000$ tasks in few seconds, almost approaching the behavior of an omniscient serverless execution framework. This again demonstrates the efficacy of WUKONG's decentralized scheduling.

4.5.5 Factor Analysis

Task Clustering and Delay I/O. Finally, we look at the impact of task clustering and delaying I/O on WUKONG's performance. Clustering and delay I/O prevent tasks with large intermediate data from writing their data to Redis. Instead, these tasks attempt to execute downstream tasks locally. Any downstream tasks that cannot immediately be executed because their dependencies are not satisfied are put into a queue. The dependencies of the queued tasks are routinely rechecked until the dependencies are satisfied or a maximum delay time is reached. By delaying I/O, more tasks can be clustered and expensive network

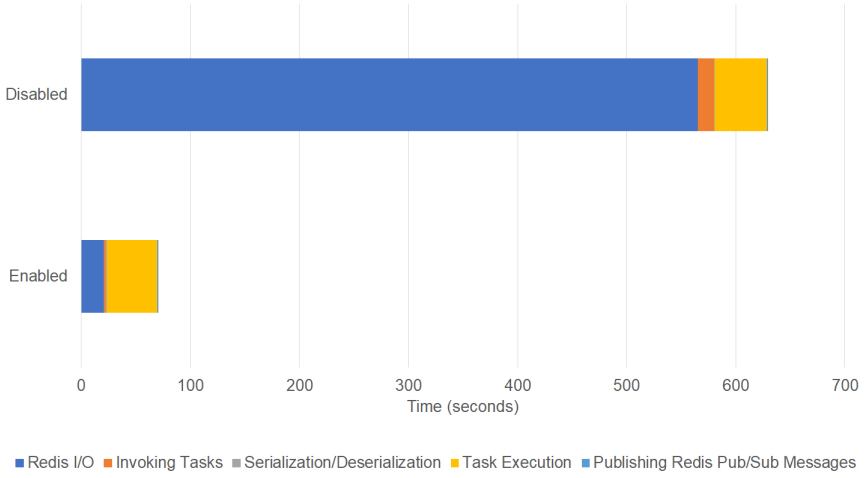


Figure 4.22: SVD2 50k x 50k aggregated execution time breakdown with and without task clustering and delay I/O.

I/Os data can be avoided, decreasing the workload runtime dramatically.

Figure 4.22 displays two aggregations of time for the activities performed for SVD2. In both cases, "publishing messages", "task execution", and "serialization/deserialization" each took roughly the same amount of time (in aggregate); however, the difference between the times for task invocation and especially Redis I/O is significant. With the two optimizations disabled, task invocations and Redis I/O made up an aggregate 14.80 and 565.21 seconds, respectively. When the optimizations were enabled, task invocation took an aggregate 2.05 seconds while Redis I/O took just 20.36 seconds. There is 7.21× more aggregate time spent invoking tasks and 27.76× more aggregate network I/O performed with clustering disabled.

We analyze Wukong's performance by breaking down the performance gap between a baseline and Wukong with all optimizations enabled (Figure 4.23). The use of the Fargate multi-Redis storage cluster results in a 20.85% performance improvement over using AWS ElastiCache for intermediate data storage. When using Fargate, the I/O performed during the workload is spread across a large number of Redis instances, resulting in reduced

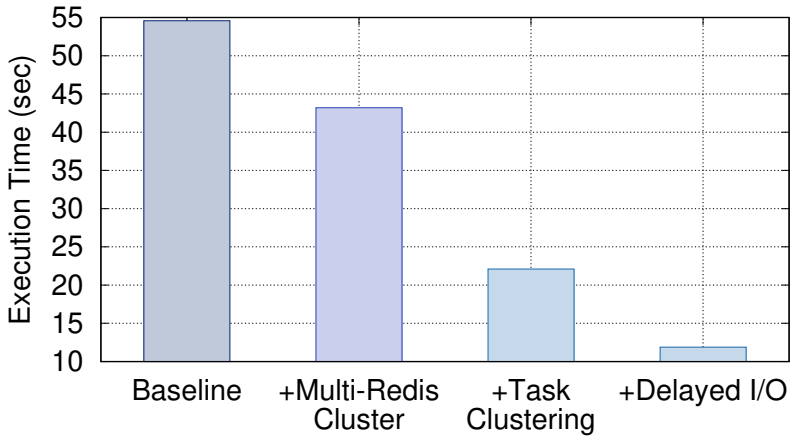


Figure 4.23: Contributions of optimizations to Wukong's performance of SVD2.

network contention and consequently reduced I/O latency. (Using a large number of ElastiCache instances is cost prohibitive.) When clustering (without delayed I/O) is enabled, performance improves by another 48.82% as a significant amount of large object I/Os is eliminated. Finally, enabling delayed I/O results in a 46.21% improvement relative to the use of clustering and Fargate alone. Overall, Wukong is 4.6× faster when all optimizations are used, demonstrating the effectiveness of these techniques.

4.6 Discussion and Lessons

WUKONG is not intended to replace established, serverful processing frameworks such as Spark [43] and TensorFlow [61]. Rather, because WUKONG is less powerful but easier to use than these serverful frameworks, WUKONG targets users who lack a strong CS background, but who work on lighter-weight and smaller computing-related problems, in areas such as data analytics and machine learning, particularly those whose solutions can be implemented using numerical Python libraries.

Our initial attempt to port Dask to a serverless platform was unsuccessful due to the poor performance of the resulting framework [2]. There were several major bottlenecks in this original design. For one, the use of a centralized scheduler to assign tasks to Lambda executors was too slow for all but the smallest workloads. Specifically, the centralized scheduler was unable to process thousands of concurrent network connections with Lambda executors. Additionally, the centralized scheduler struggled to rapidly invoke Lambda functions when scaling out for large workloads. Finally, Lambda executors spent an overwhelming majority of their time reading and writing intermediate data. [2] presents a preliminary study about these performance bottlenecks.

To address these performance issues, we developed our decentralized scheduling technique. While this technique significantly improved performance, there were still bottlenecks due to a lack of data locality. Specifically, reading and writing very large intermediate objects dominated end-to-end execution time, particularly for workloads such as SVD. To address these bottlenecks, we developed task clustering and delayed I/O. Though these techniques have been used in serverful contexts [32, 62], they had never been utilized in a serverless context. With the addition of these two techniques, large object reads and writes were eliminated, which greatly improved performance and resource utilization.

4.7 Wukong 2.0 Chapter Summary

We have presented WUKONG, a new serverless parallel computing framework that uses locality-enhanced, decentralized scheduling (atop AWS Lambda), task clustering, and delayed I/O to achieve high performance, near-ideal scalability, and data locality while being cost-effective. Our evaluation demonstrates the effectiveness of decentralized scheduling, task clustering, and delayed I/O in reducing both the execution time and cost of executing workloads. Further, we have shown that WUKONG exhibits near-ideal scaling behavior, reduces the execution time by as much as 98.53% compared to numpywren, and reduces network I/O by multiple orders of magnitude. Finally, WUKONG can reduce costs by upwards of 92.96% and 95.67% compared to numpywren and Dask, respectively.

Chapter 5: Wukong 3.0 - Going Completely Serverless

Wukong 2.0 improved on Wukong 1.0 by adding a pair of scheduling techniques - task clustering and delayed I/O - that reduce the amount of parallelism in order to avoid large communication delays. The Wukong 1.0 storage component was also redesigned in Wukong 2.0. Wukong 2.0 manages a cluster of AWS Fargate nodes that each run an instance of Redis. The Fargate cluster used by Wukong 2.0 is more elastic than the single Redis node used by Wukong.

Wukong 2.0 supports fast and efficient, DAG-based, parallel-computation workflows. However, Wukong 2.0 workflows may not be easy to deploy and manage, especially for users who lack a strong CS background. This includes domain scientists who work on lighter-weight and smaller computing-related problems in areas such as data analytics and machine learning.

The final version of our DAG-execution framework, called Wukong 3.0, is very easy to use. We utilize a peer-to-peer (P2P) communication library [63] that uses special techniques to circumvent AWS Lambda's outbound-only network connectivity constraint and enable direct communication between Lambda functions. This P2P communication service is serverless – no deployment or management effort is required by the user. We also re-implemented the remaining serverful components of Wukong 2.0, including the DAG generator and leaf task invokers, as serverless functions.

Wukong 3.0's performance is generally slower than that of Wukong 2.0. This is due to the overhead associated with establishing P2P connections and the fact that transferring data via P2P is slower than doing so using Redis. In order to speed-up Wukong 3.0's performance, we optimized the Lambda Executors. First, the Lambda Executors were rewritten to be multithreaded and to use asynchronous I/O, which enables data transfer and data processing to be performed concurrently. Second, Lambda Executors are now

multi-threaded to take advantage of the fact that Lambda Executors can be configured to run on up to six cores.

The remainder of this chapter will be spent describing Wukong 3.0. In Section 5.1, we describe the effort required to deploy and manage Wukong 2.0. Wukong 2.0 uses two types of servers. Internal servers are servers that execute Wukong 2.0 code components. These internal servers may use one or more external servers, which are servers that are managed by AWS and that run AWS code. Wukong 2.0's internal and external servers require varying degrees of user effort to deploy and manage. In Section 5.2, we describe the changes that were made to Wukong 2.0 code components so that most of the Wukong 2.0 internal and external servers could be removed, and we present the new Wukong 3.0 system architecture. Wukong 3.0's Executors no longer store intermediate data on external Fargate servers; instead, Executors exchange intermediate data using peer-to-peer (P2P) communication. The design of the P2P service is presented in Section 5.3. In Section 5.4, we describe the deployment and management of Wukong 3.0, which is a pure-serverless DAG-execution framework. In Section 5.5, we compare the performance of Wukong 3.0 and Wukong 2.0. In general, Wukong 3.0 is roughly 70% slower on the benchmarks we tested, but Wukong 3.0 is very simple to use.

5.1 Deployment and Management of Wukong 2.0

WUKONG 2.0 was designed to be executed on the AWS cloud platform. The serverless executors are implemented as AWS Lambda functions. The Static Scheduler, Metadata Store, and Key-Value-Store Proxy components are designed to be run within AWS EC2 virtual machines. Finally, the elastic storage component of WUKONG 2.0 is an AWS Fargate cluster that is controlled by the Static Scheduler. The various serverful components of WUKONG can be categorized as **internal servers** and **external servers**. The internal servers run various WUKONG 2.0 code components, while the external servers are part of AWS Fargate and AWS Lambda. That is, the external servers are not directly executing WUKONG code.

Wukong 2.0 Internal Servers

WUKONG 2.0 uses three internal servers. The first is used to run the WUKONG 2.0 front-end Client Interface code as shown in Fig. 4.5. Users submit their Python-written Dask programs to this front-end client. The second internal server is used to run the Static Scheduler code components. These components include the DAG generator, the static scheduler generator, the subscriber processes, and the initial task invokers. These components all run on an AWS EC2 VM. The third and final internal server is used to run the code component of the Storage Manager, the Key-Value-Store Proxy. This component also runs on an AWS EC2 VM.

Conceptually, there are four external servers used by WUKONG 2.0. The first external server is the Metadata Server, which stores workload metadata including static schedules, dependency counters, and other dependency information used to manage the fan-in operations in a DAG. The final results of a workload are also stored in the Metadata Server before being delivered to the client. (It is worth noting that the Metadata Server is typically housed within the same virtual machine as the Key-Value-Store Proxy, but this is not required.) The second external server is comprised of the virtual machines on which the AWS Fargate nodes run, each each of which contains a Redis server used to store intermediate data. The third external server is the virtual machines on which S3 AWS S3 (Simple Storage Service)

runs. WUKONG 2.0 uses AWS S3 to dynamically install additional runtime dependencies (e.g., large Python modules required for machine learning workloads). In this way, AWS S3 allows WUKONG to circumvent the 512MB AWS Lambda function deployment limit. The forth and final external server refers to the virtual machines on which the AWS Lambda Executors run. The virtual machines of S3 and Lambda are abstracted away from the user and are fully managed by AWS.

Wukong 2.0 Deployment

The deployment of WUKONG 2.0 requires several AWS services to be configured: AWS Fargate, AWS Lambda, AWS EC2, and AWS VPC. In order to reduce the complexity of the deployment process, WUKONG 2.0 provides an automated script for deploying the required AWS infrastructure, including the 3 internal and 1 external EC2 VMs. As of right now, the script does *not* automatically create the AWS Fargate cluster. To make running WUKONG 2.0 easier to deploy, the framework can trivially be configured to use a single instance of Redis running on an EC2 virtual machine in place of the AWS Fargate cluster. This is done via a simple argument passed through Python code.

WUKONG 2.0 also provides an AWS Serverless Application Model (SAM) application to automate the deployment of the WUKONG 2.0 Lambda functions. The SAM application allows WUKONG 2.0 to automatically retrieve the information that is required for executing workflows, such as the names of the WUKONG 2.0 Lambda functions. Users can modify a configuration file to indicate whether the Static Scheduler should use user-created Lambda functions or automatically discover the Lambda functions created by the AWS SAM template (default).

Users deploy S3 by creating an S3 bucket and then specifying the name of the bucket in the WUKONG 2.0 configuration file.

Wukong 2.0 Management

WUKONG 2.0 management consists of starting and stopping servers before and after using the framework. In order to use WUKONG 2.0, the user must start the three internal and the external virtual machine for the Metadata Store before running any WUKONG 2.0 workloads.

These virtual machines must then be stopped when all workloads are finished and the user is done using WUKONG 2.0. Additionally, though the Static Scheduler will automatically start the AWS Fargate nodes, the user must manually shut down the nodes. This can be done by calling the appropriate function of the Static Scheduler programmatically (i.e., in Python code) or through the AWS web console or CLI tools.

Typically, the EC2 virtual machines enter the `RUNNING` state within a few seconds. The AWS Fargate cluster takes longer to start up. The exact amount of time depends on how many Fargate nodes must be started, as specified by the user. When using the default number of AWS Fargate nodes (75 at the time of writing), the start-up period typically lasts between one and three minutes. All Fargate nodes must enter the `RUNNING` state before a workload may be executed on WUKONG.

The use of deployment scripts and serverless computing make WUKONG 2.0 easy to use. Beyond the initial deployment and setup of the required AWS infrastructure, the only management that users are responsible for is starting and stopping the EC2 servers and the Fargate nodes. However, domain scientists who lack a strong CS background may find WUKONG 2.0 harder to use. This is largely because of the steep learning curve of AWS. The web-based AWS consoles are complicated and are not easy to navigate, and AWS' extensive cloud terminology takes time to learn. Users must select an EC2 pricing model, which specifies how compute capacity is charged. Users must also select EC2 server instances from a large number of instance types, each type having a different combination of CPU, memory, storage, and networking capacity. Choosing the right instance type requires some knowledge about computer architecture and networking.

5.2 Wukong 3.0 Architecture and Design

5.2.1 Removing the Internal and External Servers

To make Wukong easier to deploy and manage, we removed Wukong 2.0's dependency on internal and external servers that required some management effort by the user. Wukong 3.0 users have a choice about where to run the front-end client. They can execute the front-end client on the same personal computer that they use to connect to AWS. Alternatively, they can deploy and manage an EC2 virtual machine to run the client, as was done in Wukong 2.0.

The Static Scheduler code components have been redesigned and rewritten to run exclusively as serverless functions. The same has been done for the Key-Value-Store Proxy. Consequently, the three internal EC2 virtual machines that were used in Wukong 2.0 are no longer required. In addition, the Wukong 2.0 Lambda Executors have been rewritten to use a P2P (message-passing) communication system rather than relying on external storage (i.e., the AWS Fargate cluster).

The AWS Fargate cluster and its associated VMs are not used in Wukong 3.0. Thus, the Metadata Store and the Key-Value-Store Proxy external servers do not exist in Wukong 3.0. Instead, Wukong 3.0 Executors transfer metadata and intermediate data via a P2P communication protocol. P2P requires the service of an internal server that runs the Coordinator Service shown in Fig. 5.1. However, the Coordinator Service is fully-managed by Wukong 3.0, so it requires no management effort on the part of the user. Wukong 3.0's P2P protocol is described in Section 5.3. Wukong 3.0 retains the use of AWS S3 for storing metadata/metrics and, in certain cases, intermediate data, as described in 5.3.4.

5.2.2 Wukong 3.0 System Components

Fig. 5.1 shows the serverless system architecture for Wukong 3.0. The Wukong 2.0 Static Scheduler was converted into a collection of serverless Lambda functions referred to as the DAG Generator in Fig. 5.1. In Wukong 2.0, users would submit their Dask Python code to a long-running scheduler process, which would convert this code into a DAG, perform a series of depth-first searches to divide the DAG into subgraphs, and then pass these subgraphs to the initial task invoker processes. In Wukong 3.0, the front-end client packages-up the user's Dask Python workload code and submits the package to the DAG Generator Lambda function. This function generates the input DAG using

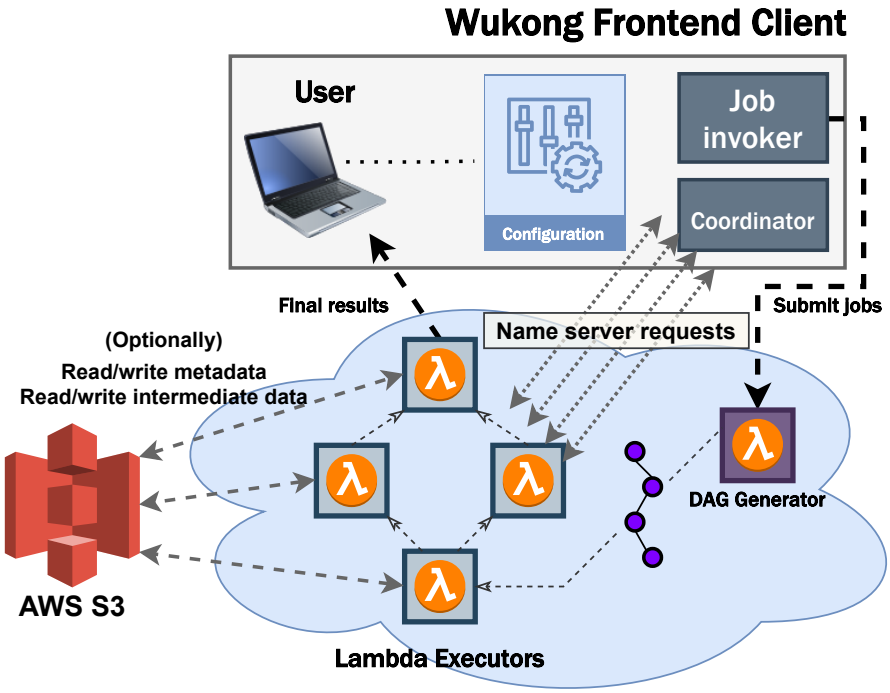


Figure 5.1: WUKONG 3.0 Architecture.

Dask and then invokes a series of Lambda Executors, assigning each Executor a package containing one or more unique leaf tasks of the input DAG.

When the Executors begin running, they first inspect their invocation package to determine how many leaf tasks, if any, they have been assigned. If an Executor has been assigned one or more leaf tasks, then the Executor performs a DFS for each leaf task in order to break up the DAG into subgraphs, which are also referred to as static schedules. The Executor then executes the tasks in its static schedule, just as it did in WUKONG 2.0.

The Coordinator service is used as a name server to facilitate peer-to-peer connections between Lambda functions. The exact process by which Lambdas establish connections to one another is described in 5.3. The Coordinator is also used in dependency management and dependency tracking (i.e., determining when downstream tasks are ready-to-execute on the basis of whether or not their data dependencies are available). In order for the Coordinator to perform dependency management, it is made aware of the general structure of the workload. This includes the unique IDs of tasks (i.e.,

task keys), the dependents and dependencies of each task, etc.

Dependency Management in Wukong 3.0

AWS Lambda Executors transfer intermediate data to one another using the P2P communication protocol described in 5.3. Since there is no longer a centralized, external storage, the dependency management system of WUKONG needed to be redesigned. In order for Lambdas to determine when downstream tasks are ready to execute (i.e., when all the data dependencies of the downstream task are computed and available), WUKONG 3.0 uses the Coordinator service along with the P2P fan-in protocol described in 5.3.3.

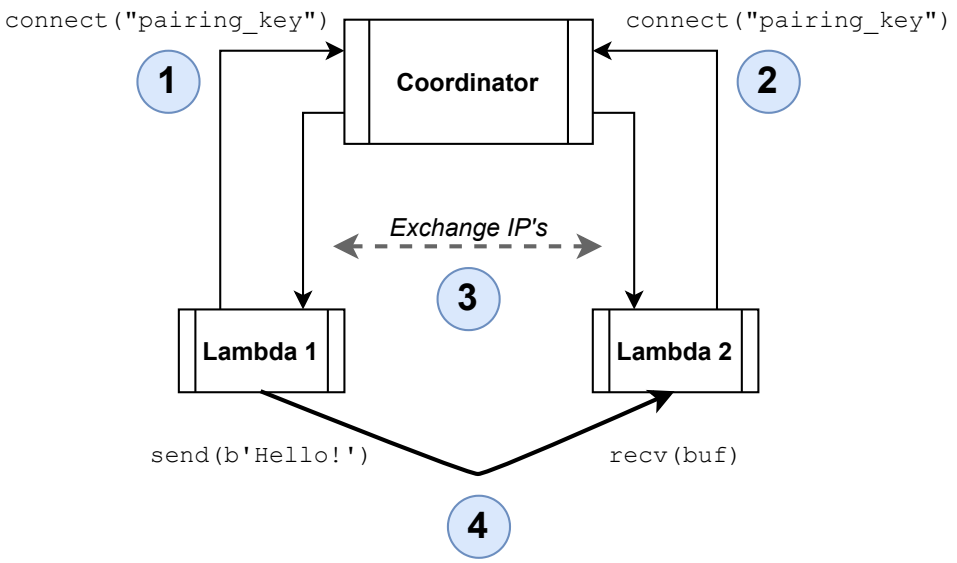


Figure 5.2: The P2P communication protocol used in WUKONG 3.0. Lambda functions issue pairing requests to the Coordinator, including a case-sensitive pairing key with the request. The Coordinator exchanges IP information between Lambdas based on these pairing keys.

5.3 Peer-to-Peer (P2P) Communication in Wukong 3.0

5.3.1 P2P Overview

WUKONG 3.0 uses UDT [64], a reliable UDP based application-level data transport protocol, and a technique known as NAT hole punching, as implemented by the ServerlessNetworkingClients library [63] for peer-to-peer (P2P) networking in serverless applications. This P2P protocol allows WUKONG 3.0's AWS Lambda Executors to directly communicate with one another. The use of P2P communication, along with a redesign of how Executors cooperate during dynamic scheduling, eliminates the need for the Fargate storage cluster in WUKONG 3.0.

WUKONG 3.0's P2P communication requires the Lambda Executors to use a long-running Coordinator service. The Coordinator service is used as a name server by the Executors, i.e., the Coordinator service tells each Executor what its IP address is so that the Executors can connect to one another via P2P.

5.3.2 Standard Connection Establishment

Connection establishment between AWS Lambda functions occurs in four steps. Fig. 5.2 displays a diagram of this process. During steps one and two, each Lambda function will issue a *pairing request* to the Coordinator service. The Lambdas will pass several arguments to the Coordinator in this request, but the most important argument is the *pairing key*. The Coordinator receives pairing requests from Lambda functions, and matches Lambdas together according to their pairing key. When the Coordinator receives a pairing request with a pairing key it has not yet seen, the Coordinator does not respond. Instead, the Coordinator waits until it receives a second pairing request with the same pairing key. When this occurs, the Coordinator exchanges the IP addresses of the two Lambdas that issued pairing requests with the same pairing key. This is step three. Once the Lambdas receive each other's IP addresses, they perform the forth and final step by establishing a direct connection with each other. At this point, they can communicate by sending and receiving messages.

If more than two Lambdas issue pairing requests with the same pairing key, then the Coordinator simply pairs the first two Lambdas it receives requests from. If the Coordinator receives another two pairing requests for that same key, then it will just pair the Lambdas that issued those requests. If the Coordinator receives only a single pairing request for a particular pairing key, then the Lambda that issued that request will never be paired with another Lambda (based on that pairing key, at least).

5.3.3 Connection Establishment During Fan-Ins

During a fan-in operation, one Lambda Executor is responsible for executing the downstream task. This Executor is referred to as the *fan-in Lambda* or *fan-in Executor*. The other Executors that have input data that the downstream task depends on must transfer their data to the fan-in Executor. This manifests as a many-to-one connection topology in which several upstream Lambdas are all trying to connect to the same downstream Lambda. In order to facilitate this network behavior, the Coordinator adopts a modified protocol.

When an Executor successfully completes a task, it iterates over the dependents of that task. Ideally, the Executor will be able to become the Executor for one of the downstream tasks and thus

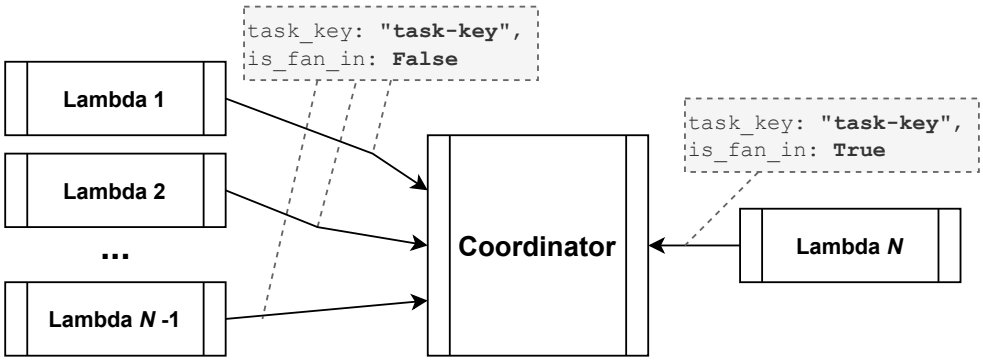


Figure 5.3: A fan-in operation in Wukong 3.0.

avoid transferring its data over the network. The Executor will transmit its intermediate data to the Lambdas responsible for the other downstream tasks. In order to "become" a downstream task, the Executor contacts the Coordinator, specifying the task key of the downstream task and a flag indicating that the Executor would like to become responsible for executing the downstream task. If no other Executors have tried to claim responsibility for executing this task, the Coordinator will inform the Executor that it can execute the downstream task.

At this point, one of two things will happen. If the downstream task has a single dependency, then the Executor can immediately begin executing the task. However, if the downstream task has *more than one* dependency, the Executor must perform a fan-in operation. Note that, because the Coordinator knows the overall structure of the workload, the Coordinator will already know whether or not a fan-in operation must take place. In order to initiate the fan-in operation, the fan-in Executor will contact the Coordinator again, this time indicating that it is the designated fan-in Lambda. The Coordinator will incrementally exchange the IP addresses between the paired fan-in Executor and upstream Executors as data dependencies for the fan-in task are completed. The Executors establish connections and begin transmitting data as soon as their IPs are exchanged, and the fan-in Executor will maintain its connection to the Coordinator until all IP addresses have been exchanged. This process is illustrated in Fig. 5.3.

The fan-in Executor will continue receiving data directly from its peer Lambdas until all data dependencies required to execute the fan-in task have been gathered. Once the final IP address between upstream Executors and the fan-in Executor has been exchanged, the fan-in Executor

will terminate its connection to the Coordinator. Once this occurs and all required data has been successfully transferred, the fan-in Executor will execute the fan-in task.

If the fan-in Executor has tasks other than the fan-in task that are ready-to-execute, then the Executor can execute these tasks while the fan-in operation occurs in the background. In order to overlap the transmission of data dependencies, connection establishment, and task execution, the Wukong 2.0 Executors were completely rewritten to be fully asynchronous. This allows for almost all operations performed by the Executor to be performed asynchronously (i.e., in an overlapping/interleaved manner). The Executor is able to make progress on computation tasks while connection establishment and data transfer (either sending or receiving data) occurs transparently in the background.

5.3.4 Connection Establishment During Fan-Outs

As described in the Chapters 3 and 4, a fan-out operation occurs when a task has multiple downstream dependents. In this scenario, the upstream Lambda Executor invokes downstream Lambdas to execute the dependent tasks. The connection topology here can be thought of as one-to-many. This scenario is handled in the standard way. The upstream Lambda performs a series of P2P connections for each of the newly-invoked downstream Lambdas. The upstream Lambda transmits its data to each of the downstream Lambdas, after which the downstream Lambdas begin executing their assigned tasks.

Large fan-out operations may incur significant overhead due to repeatedly establishing peer-to-peer connections and transmitting the same data over and over. In order to mitigate the performance impact of large fan-outs, Wukong 3.0 can be configured with a threshold parameter corresponding to the number of Lambdas involved in the fan-out. If the number of Lambdas is greater than or equal to this threshold, the upstream Lambda can opt to store its data in AWS S3. The newly-invoked downstream Lambdas will receive a notification of this in their invocation payload, thereby completely circumventing the peer-to-peer connection establishment protocol. Instead, the downstream Lambdas will simply retrieve the intermediate data and necessary workload metadata from AWS S3.

5.4 Deployment and Management of Wukong 3.0

The deployment and management of serverless Wukong 3.0 is much simpler than that of the serverless-oriented Wukong 2.0. This is because all user-managed servers in Wukong 2.0 were either removed completely or replaced with serverless versions in Wukong 3.0.

5.4.1 Deployment

Wukong 3.0 deployment requires users to deploy two AWS Lambda functions. These functions can be deployed manually or by using the AWS SAM template that is also used in Wukong 2.0. Wukong 3.0 users can execute the front-end client on their personal computer, meaning they are no longer required to deploy and manage an EC2 virtual machine to run the client. Users can also elect to deploy and run the front-end client on a user-deployed EC2 machine as was done in Wukong 2.0 (for performance reasons, for example).

The Coordinator service used by Wukong 3.0 can be automatically deployed by Wukong 3.0's front-end client on the user's personal computer. No deployment effort is required from the user in this case. Alternately, users can deploy the Coordinator manually on an AWS EC2 VM, which may provide better performance than what their home network and computer can provide. In this case, the user may run both the Coordinator and the front-end client on the same virtual machine or on two different virtual machines.

The process of deploying the Coordinator manually on EC2 is relatively straight-forward. Users must create an EC2 machine and start the Coordinator process on that machine. The public IPv4 address of the EC2 VM must then be specified in Wukong 3.0's configuration file. We note that cloud providers could eventually provide a serverless Coordinator service. Also, the Coordinator does not depend upon any AWS-specific API or infrastructure. In theory, users could deploy the Coordinator service on any cloud platform, though it has only been tested on AWS EC2.

The final deployment step of Wukong 3.0 involves creating an S3 bucket and then specifying the S3 bucket's name in Wukong 3.0's configuration file. This can be done manually via the AWS S3 web interface, the AWS CLI, or by using the script provided by Wukong 3.0.

5.4.2 Management

If Wukong 3.0's front-end client and Coordinator service are deployed on the user's personal computer, the Coordinator service will automatically be managed by Wukong 3.0's front-end client.

No management effort is required by the user in this case. Alternately, users can deploy and manage the front-end client and the Coordinator manually on one or two EC2 VMs. In this case, the one or two EC2 VMs used must be turned on and off by the user. S3 is a fully-managed storage service provided by AWS. As a result, there is no management required on the part of the user.

As soon as the Coordinator, the front-end client, and the two Lambda functions have been deployed, a workload can be executed on Wukong 3.0 using a simple command. For example, executing SVD1 (singular value decomposition of a tall-and-skinny matrix) for a $128,000 \times 1000$ matrix with $10,000 \times 100$ partitions can be accomplished by running the following command:

```
python3 driver.py --function svd --array_dimension 128000 --chunk_dimension 10000
```

The job will begin automatically, and the result will be delivered back to the client. Wukong 3.0 will also automatically generate several interesting graphs and plots based on the execution of the workload, all the while transparently collecting a vast amount of workload metadata. All of this information will be automatically returned to the user. The amount of data collected and the graphs/plots generated by Wukong 3.0 can be controlled via arguments passed to the framework.

5.5 Wukong 3.0 Evaluation

5.5.1 Implementation

Wukong 3.0 was implemented in roughly 8,531 lines of Python and Golang code (5,100 for the front-end client and 3,431 for the AWS Lambda Executor). The Dask [44] library is once again used to generate DAGs, which are used as input for Wukong 3.0.

In order to improve Wukong 3.0's performance, we optimized the Lambda Executors. First, the Lambda Executors were rewritten to use asynchronous I/O, which enables data transfer and data processing to be performed concurrently. Second, Lambda Executors were rewritten to be multi-threaded, in order to take advantage of the fact that Lambda Executors can be configured to run on up to six cores. Executors executing on multiple cores can focus on task execution and other operations (e.g., managing task data, tracking and updating task dependencies, etc.), while expensive I/O operations are offloaded to the remaining cores. These optimizations were implemented very early in the overall development of Wukong 3.0. As a result, there does exist a stable version of Wukong 3.0 without these optimizations, meaning it is not possible to compare the current version of Wukong 3.0 against a pre-optimized version.

5.5.2 Experimental Goals and Methology

This evaluation was performed on AWS connected to by a desktop computer. This desktop had an i7 3770k CPU and 32GB of DDR3 Corsair Vengeance 1866 MHz RAM. The operating system was Windows 10 Pro 10.0.19041 N/A Build 19041. The AWS Lambda executors were configured with a timeout of two minutes and 3GB of main memory. In all experiments, the Coordinator and front-end client were executed on the local desktop computer. Note that the performance results of Wukong 2.0, Dask, and Numpywren are reused from Section 4.5.

The goal of this evaluation is to determine the potential performance of a completely serverless DAG execution engine. To that end, Wukong 3.0 is compared directly against Wukong 2.0 to determine the performance difference between a serverless-oriented DAG execution (i.e., Wukong 2.0) and a completely serverless execution framework. Wukong 3.0 is also compared against Dask as both frameworks use the same input DAGs and perform identical computations. Finally, Wukong 3.0 is compared against Numpywren as it is arguably the most closely-related framework and because

Numpywren was compared against Wukong 2.0.

The differences that existed between Wukong 2.0 and Numpywren also exist between Wukong 3.0 and Numpywren. That is, Wukong 3.0 is uses different input DAGs than Numpywren. While Wukong 3.0 uses Dask DAGs that explicitly encode each task and its dependencies, numpywren uses an implicit DAG representation for its programs, all of which are implemented using the LAmbdaPACK language for linear algebra [9]. Each node in a Numpywren DAG is generated on-demand at runtime. Additionally, numpywren uses AWS S3 for all of its intermediate storage, whereas Wukong 3.0 only uses S3 for large fan-out operations. Using the default configuration, 80-90% or more of the data transfer in Wukong 3.0 is carried out using the peer-to-peer protocol for the workloads tested.

Only a subset of the workloads used to evaluate Wukong 2.0 are also used to evaluate Wukong 3.0. This is primary due to the fact that Wukong 3.0 suffers from several significant performance bottlenecks, as will be shown in the next section. The overall performance of Wukong 3.0 is sufficiently evaluated using the two selected workloads. The first workload is **Singular Value Decomposition** (SVD), specifically the SVD1 variant from [3]. SVD1 computes the SVD of a tall-and-skinny matrix. The second workload os **Tall-and-Skinny QR Reduction** (TSQR). This workload performs a QR reduction of a tall-and-skinny matrix. Together, these two workloads illustrate the general performance trends of Wukong 3.0 and highlight the performance bottlenecks.

5.5.3 End-to-End Performance Comparison

SVD1. Fig. 5.4 shows the end-to-end performance results for SVD1. This figure includes the results for Wukong 2.0, labeled *Wukong (Multi-Redis)*, the two versions of Dask used in Section 4.5, and Wukong 3.0, labeled *Wukong (P2P)*. Wukong 3.0 executes all problem sizes of the workload significantly slower than the other frameworks, including both versions of Dask. While the performance gap between the 1,000-Worker Dask version and Wukong 3.0 decreases as the problem size increases, the 1,000-Worker Dask version still executes the workload roughly 2× faster than Wukong 3.0 for the largest problem size. Additionally, Wukong 2.0 completes the largest problem size approximately 6.96× faster (i.e., 85.64% faster) than Wukong 3.0.

The reason for the large disparity in performance between Wukong 2.0 and Wukong 3.0 is due

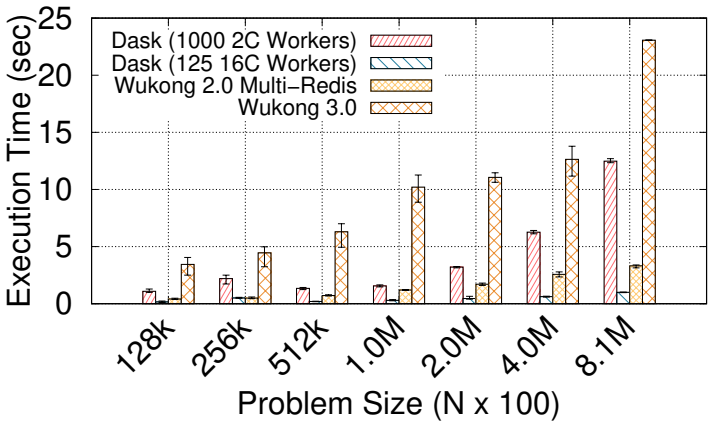


Figure 5.4: End-to-end times for SVD1.

to the overhead of establishing P2P connections. It is not uncommon for functions to take four to five seconds to successfully establish a connection. This issue was investigated during development, and the Coordinator was not found to be a source of the bottleneck. Running the Coordinator (which is implemented in Golang using goroutines and is therefore highly parallel) on a more powerful machine (i.e., more CPU cores, higher network bandwidth, etc.) does *not* have an impact on end-to-end performance. In addition to the large overhead for establishing peer-to-peer connections, the relatively low network bandwidth of the peer-to-peer communication also negatively impacts performance. During testing, the network bandwidth of the P2P connections was found to be around 50-60Mbps, which is significantly less than non-P2P communication. (Note that the network bandwidth of a Lambda function generally depends upon the type of EC2 VM on which the function is executing and how many other Lambda functions are executing on the same VM, sharing the network resources.)

The overhead of P2P being the primary reason for the poor performance of Wukong 3.0 is reinforced by the general trend of the data. As the problem size is doubled, the end-to-end time of the workload does not increase substantially (and in-fact typically stays within the upper bound of the runtimes of the previous workload size). As we reported in Chapter 4, Wukong 2.0 is also often limited by network performance. These results show that the network performance limitation is more severe for Wukong 3.0.

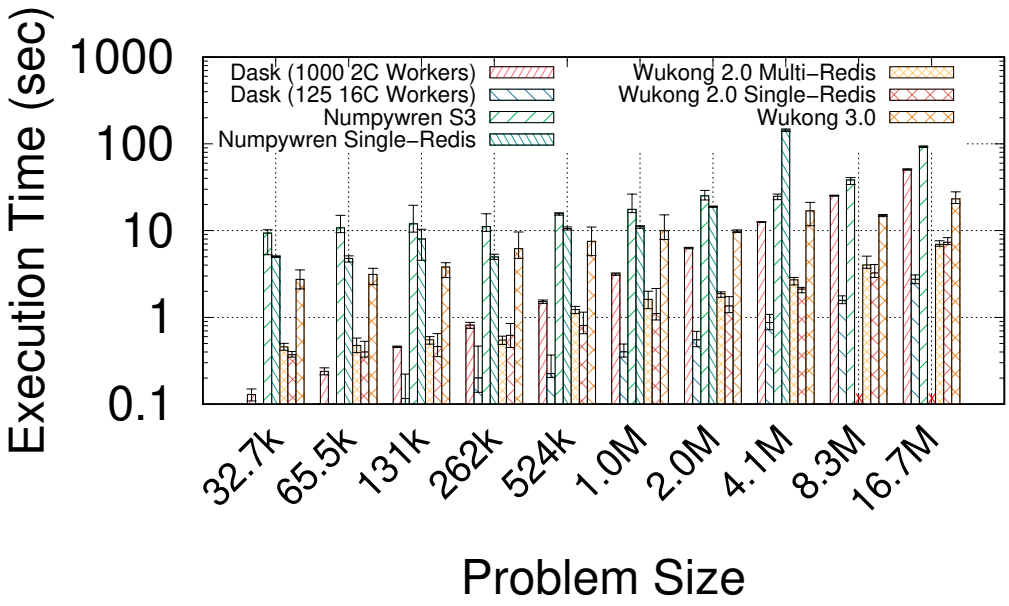


Figure 5.5: End-to-end times for TSQR.

TSQR. Fig. 5.5 shows the end-to-end performance results of the TSQR workload for several different frameworks. All of the frameworks used in the evaluation of Wukong 2.0 are included along with Wukong 3.0, which is labeled *Wukong P2P*. For the TSQR workload, Wukong 3.0 actually outperforms both variants of numpywren for all problem sizes except for $262k$, in which the Single-Redis version of numpywren slightly out-performed Wukong 3.0. The disparity between the performance of Wukong 3.0 and numpywren generally increased as the problem size increased.

The largest gap between the Single-Redis variant of numpywren and Wukong 3.0 occurred during the $4.1M$ workload (which was the last problem size that the Single-Redis variant of numpywren completed successfully). For that workload, Wukong 3.0 completed the workload roughly 8.5× faster (i.e., 88.29% faster) on average than Single-Redis numpywren. As for numpywren S3, the largest performance gap occurred during the $16.7M$ workload. For that problem size, Wukong 3.0 completed the workload approximately 3.98× faster (i.e., 74.93% faster) on average than numpywren S3.

The 125-worker Dask cluster out-performed Wukong 3.0 (and every other framework) for every

problem size. However, WUKONG 3.0 out-performed the 1000-Worker Dask cluster for the $8.3M$ and $16.7M$ problem sizes, completing the workload $1.68\times$ (40.52%) and $2.16\times$ (53.81%) faster on average, respectively. This is likely due to the substantial scheduling and communication overhead incurred by the 1000-worker Dask cluster.

As with the SVD1 workloads, WUKONG 3.0 failed to perform as well as WUKONG 2.0 for all problem sizes, though the performance gap actually decreased significantly by the $16.7M$ workload. If the trend were to continue for increasingly large problem sizes, then WUKONG 3.0 would have eventually started to out-perform both versions of WUKONG 2.0.

In fact, the end-to-end runtime for WUKONG 3.0 did not increase between the $4.1M$, $8.3M$, and $16.7M$ problem sizes. The runtime also remained the same between the $1.0M$ and $2.0M$ problem sizes. Once again, this is likely due to the overhead of establishing P2P connections. This overhead does not increase substantially with problem size, as the time to establish a connection is mostly static (and communication with the Coordinator is not a bottleneck). While the low network bandwidth is an issue, the overall runtime is still dominated by P2P connection establishment. The increase in computation and data transfer that occurs when the problem size increases does not contribute significantly to the overall end-to-end performance.

Our initial performance test showed that WUKONG 3.0 could not match the performance of WUKONG 2.0. This is due in part to the overhead associated with establishing P2P connections and the fact WUKONG 3.0 transfers data more slowly using P2P than WUKONG 2.0 does using Redis. The slow rate of data transfer is partially due to the low network bandwidth achieved using the ServerlessNetworkingClients API. Recently a new framework for P2P communication between serverless functions, called Boxer, was published [65]. Boxer achives considerably higher bandwidth and lower latency. In the future, WUKONG 3.0 can be modified to use Boxer instead of the ServerlessNetworkingClients API.

5.5.4 Runtime Analysis of Wukong 3.0

In this section, we examine the various activities performed by the AWS Lambda Executors during the execution of a typical workload on WUKONG 3.0. Specifically, we analyze the runtime activities performed during the execution of SVD1 $8.1M$ and TSQR $8.3M$ on WUKONG 3.0. This analysis will help to clarify the performance issues of WUKONG 3.0 as well as provide an explanation for the

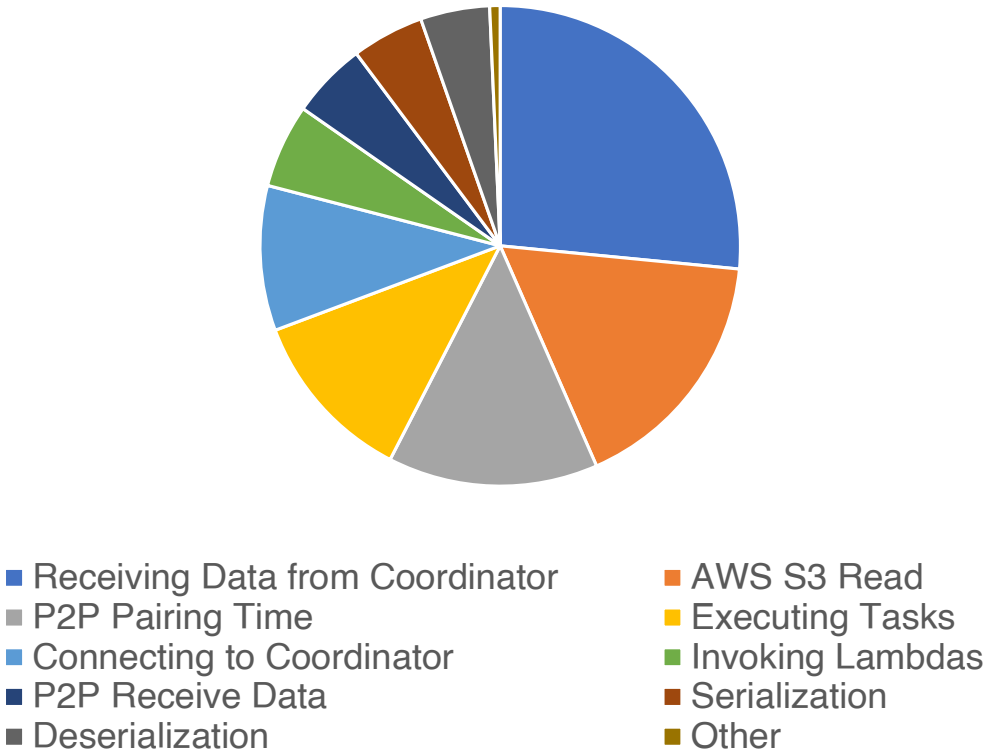


Figure 5.6: Breakdown of aggregate time spent across all AWS Lambda Executors during the execution of SVD 8.1M on Wukong 3.0.

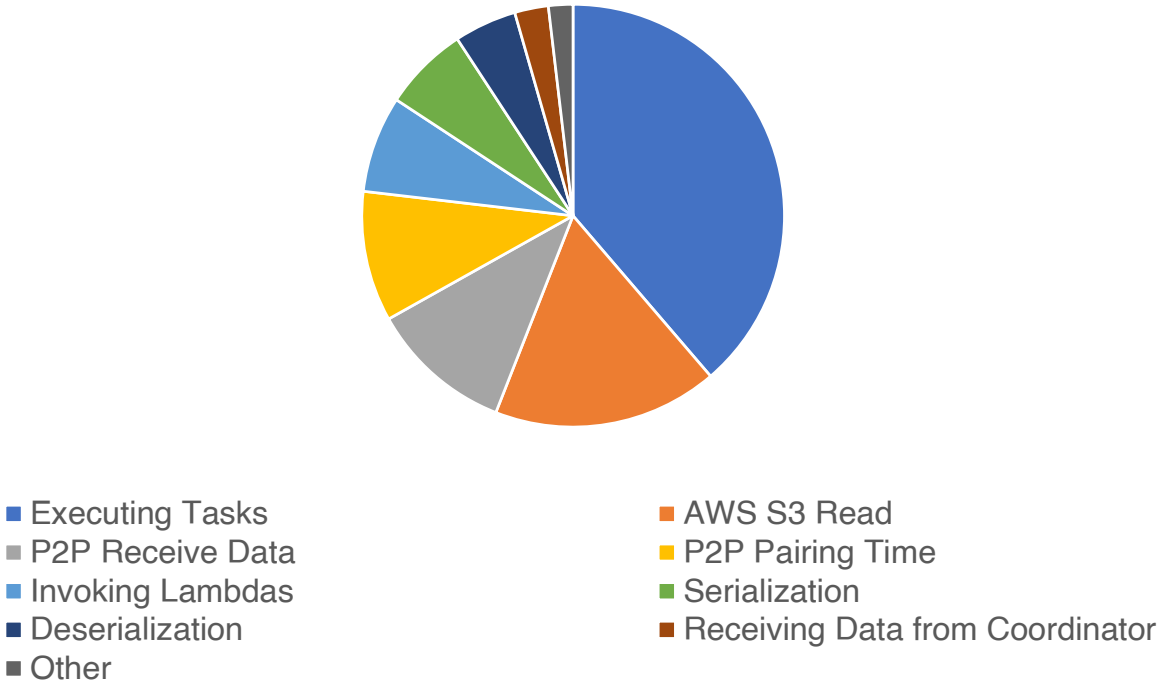


Figure 5.7: Breakdown of aggregate time spent across all AWS Lambda Executors during the execution of TSQR 8.3M on Wukong 3.0.

scaling behavior of Wukong 3.0 as problem sizes increase.

SVD1 Runtime Analysis

Figure 5.6 shows a breakdown of the aggregate time spent on various activities during SVD1 8.1 by Wukong Executors. The format of this chart is the same as Fig. 5.7. There are some notable differences between the SVD1 and TSQR workloads. In particular, the `Receiving Data from Coordinator` activity was by-far the most time-consuming activity in the SVD1 workload. This suggested that the Coordinator may be a potential bottleneck for the framework, and this was something we investigated.

The Coordinator was originally written in Python but was later rewritten in Go for performance reasons. Specifically, the Coordinator was rewritten to be fully multi-threaded using Goroutines, meaning it is capable of servicing many connections at once. Additionally, we noted that each individual connection serviced by the Coordinator requires only a fraction of a second to process. The Coordinator only needs to manage a few hash map data structures, and no intensive computations are required by the Coordinator to establish connections between paired Lambdas. This made it

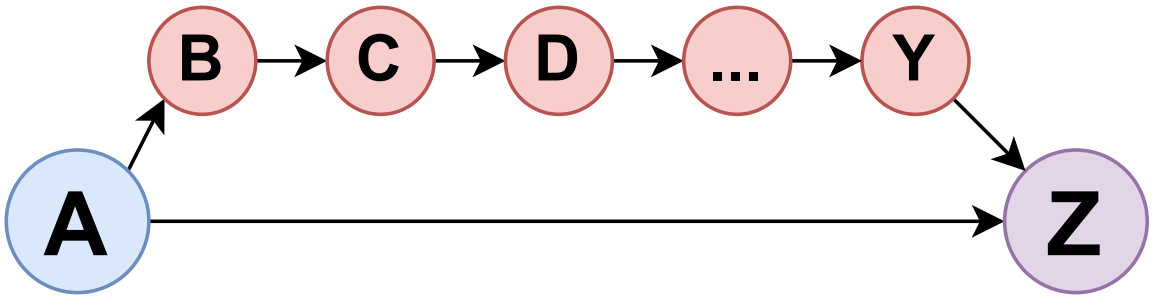


Figure 5.8: A common DAG topology in SVD workloads (present in both SVD1 and SVD2).

seem unlikely that the Coordinator was a bottleneck.

To verify that the Coordinator was not a bottleneck, we added profiling information to the Coordinator and monitored its performance during SVD1 workload execution on Wukong 3.0. Based on the profiling results, the Coordinator did not appear to be a bottleneck, as it serviced requests rapidly even during large workloads. Instead, the performance issues are a consequence of the DAG structure of the SVD1 workload: there are several large fan-ins that occur during the SVD1 workload that require a substantial amount of overhead in establishing direct connections between Lambdas.

Additionally, there is another pattern common to the DAGs of the workloads in which tasks that appear relatively early in the workload have dependents that are not executed until much later in the workload. Certain Lambdas will `become` these late-stage tasks early in the workload's execution. These Lambdas contact the Coordinator to be paired with upstream Lambdas, but the upstream Lambdas do not attempt to pair until much later in the workload (i.e., when the workload has caught up to these late-stage tasks). As a result, the upstream Lambdas end up waiting a long time for results from the Coordinator, but this is not because the Coordinator is a bottleneck. It is simply because there is no data to send to the upstream Lambdas until the workload has progressed further. Fig. 5.8 shows a sample DAG that contains a simplified version of this DAG topology.

TSQR Runtime Analysis

Figure 5.7 shows a breakdown of the aggregate time spent by Wukong Executors on various activities during TSQR 8.3M. That is, the size of each section corresponds to the sum total of the time spent performing that activity by all Lambda Executors involved in the workload's execution. Also

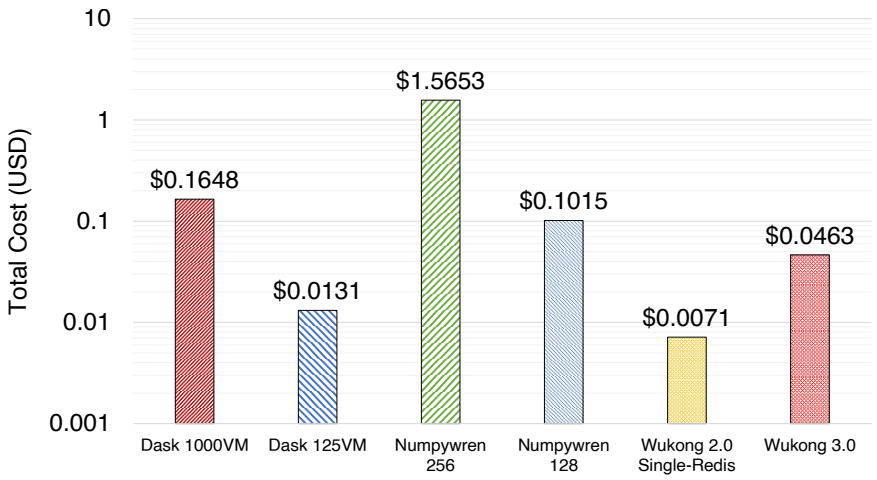


Figure 5.9: Log-scale TSQR 16.7M cost.

note that the key/legend of the figure is sorted top-down and left-to-right.

Executors spent the most time executing task code. Task execution was performed for more than twice as much time (in aggregate) compared to the next highest-performed activity, which was reading data from AWS S3. The data read from S3 includes Python dependencies (which are only read by an Executor invoked in a cold container) and possibly dependency data and workload metadata in the case of large fan-outs.

The next two highest activities are receiving data via peer-to-peer and peer-to-peer pairing time, which refers to the time spent establishing direct connections with other Executors. Executors spent a combined 51.6 seconds performing these two activities, which is about as much time as they spent reading data from AWS S3. The reason that these activities contribute significantly to the overall runtime is that a majority of the time occurs on just a few Lambdas, particularly for the `P2P Receive Data` activity. In these workloads, the intermediate results are incrementally aggregated by a few Lambdas at the end of the workload. As many upstream Lambdas attempt to connect and transmit their data, they end up getting stuck in line as the upstream Lambdas slowly establish connections. Although the Lambdas perform network operations asynchronously, performing many simultaneous network operations is still very slow.

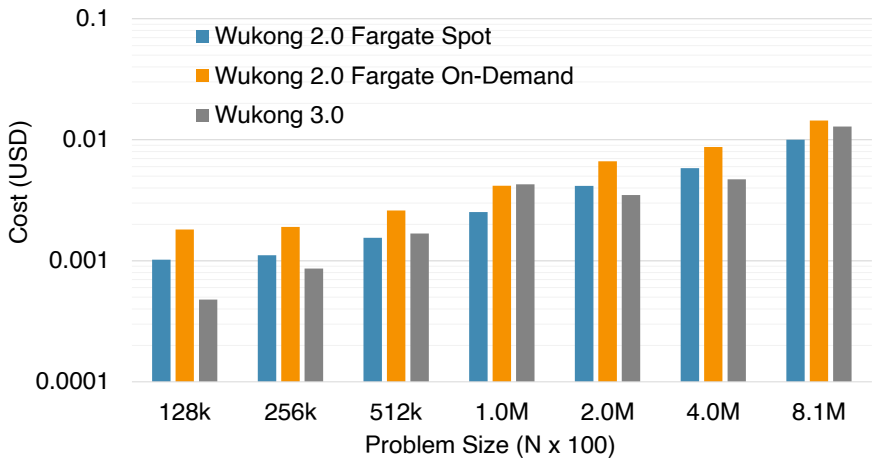


Figure 5.10: Log-scale SVD1 cost.

5.5.5 A Brief Cost Analysis of Wukong 3.0

In this section, we examine the monetary cost of running a workload on Wukong compared with the other frameworks. Fig. 5.9 displays the cost in USD of running TSQR $16.7M$ on the various frameworks. Just as before, Wukong 2.0 is the cheapest of the various frameworks by a considerable amount. The 125-Worker Dask is also fairly inexpensive. This is largely due to the extremely short running time of this workload on that Dask configuration. Wukong 3.0 is less expensive than both versions of numpywren and the 1000-Worker Dask cluster. This is because Wukong executes the workload relatively fast and without using too many compute resources to do so. That is, there are no long-running servers or anything incurring cost when using Wukong. The only component contributing to the cost of Wukong is AWS Lambda. (AWS S3 may contribute to the cost, but it is so inexpensive that the contribution would be negligible). Compared to the 256-worker numpywren configuration, Wukong 3.0 is 33.80× (i.e., 97.04%) less expensive. Compared to the 128-worker numpywren configuration, Wukong 3.0 is 2.19× (i.e., 54.38%) less expensive.

Fig. 5.10 shows the cost of running SVD1 (i.e., SVD of a tall-and-skinny matrix) on Wukong 2.0 and Wukong 3.0 for all problem sizes tested. Wukong 3.0 is cheaper for both of the pricing models (spot and on-demand) used by Wukong 2.0 for the first two problem sizes. Wukong 3.0 is cheaper than Wukong 2.0 using on-demand pricing for all problem sizes except for $1.0M$. Wukong 2.0 is the cheapest framework for the $512k$, $1.0M$, and $8.1M$ problem sizes. This is largely due to the fact that Wukong 3.0 spends a considerable amount of time establishing P2P connections, during

which the Lambdas involved incur additional cost while they are waiting for a connection to be established. These results show that the current version of Wukong 3.0 can provide competitive cost-effectiveness compared to Wukong 2.0 for certain jobs and problem sizes. As the P2P protocol used by Wukong 3.0 is optimized and improved, the overall cost of running workloads on Wukong 3.0 will decrease.

5.6 Wukong 3.0 Chapter Summary

In this chapter, we presented Wukong 3.0, a completely serverless version of Wukong. The dependencies on user-deployed and user-managed servers have been removed in Wukong 3.0. Executors use a combination of AWS S3 and a peer-to-peer protocol to transfer intermediate data rather than accessing intermediate data on Redis servers. Because Wukong 3.0 no longer requires any user-managed servers, it is the easiest version of Wukong to deploy and use.

In its current form, Wukong 3.0 is unable to consistently achieve the same performance and cost-effectiveness as Wukong 2.0. This is mainly due to bottlenecks in the P2P communication system. Despite this, Wukong 3.0 is still able to consistently provide performance and cost that is competitive with a 1000-Worker Dask cluster. Additionally, Wukong 3.0 can out-perform numpywren on TSQR workloads and is capable of delivering competitive cost-effectiveness compared to Wukong 2.0 for certain job and input sizes.

Future work on Wukong 3.0 will focus on improving the performance of the serverless communication system, including integrating proj 3.0 with the TCP/IP-based Boxer communication system for Lambda-to-Lambda communication. The end-to-end performance results of SVD1 indicate that Wukong 3.0 has the potential to out-perform Wukong 2.0 for very large problem sizes in its current form. If we are able to integrate Boxer and develop additional optimizations, Wukong may eventually become the fastest, cheapest, and easiest to use version of Wukong.

Chapter 6: Conclusion and Future Work

6.1 Conclusion

The goal of this thesis was to realize a serverless-oriented, parallel computing framework that could support fast and efficient, DAG-based, parallel-computation workflows that are easy to deploy and manage. Such a framework has ultimately been realized in Wukong 2.0 and Wukong 3.0, with the two versions offering a trade off between cost, performance, and ease-of-use. With Wukong 1.0, we successfully developed a prototypical serverless-oriented DAG engine that offered competitive cost and performance compared to a closely-related, serveful counterpart, namely Dask `distributed`.

We built and improved upon the Wukong 1.0 prototype, creating Wukong 2.0. This version of Wukong delivered excellent performance while being highly cost-effective relative to closely-related serverful and serverless frameworks. The evaluation of Wukong 2.0 demonstrated the efficacy of Wukong's novel decentralized scheduling technique. Furthermore, our results illustrated the benefit of the other important optimizations designed into Wukong 2.0, namely task collapsing and delayed I/O. Specifically, we showed that Wukong 2.0 can execute parallel computation jobs up to 68.17× faster and reduce network I/O by several orders of magnitude compared to numpywren. In terms of cost, Wukong achieves 92.26% and 95.67% tenant-side cost savings compared to numpywren and Dask, respectively. Lastly, in addition to being both cost-effective and highly performant, Wukong 2.0 is also easy to use. We provide Python scripts and an AWS SAM template to automate the deployment process of the framework.

As a final step, we developed Wukong 3.0. Our goal was to investigate the potential performance of a completely-serverless DAG execution engine and to provide a framework that is as easy to use as possible. To that end, Wukong 3.0 requires no management of external servers/virtual machines. The framework can be transparently executed from the user's personal computer or in the cloud (i.e., on AWS EC2). While the performance of Wukong 3.0 is much slower than Wukong 2.0, the final iteration of Wukong is still able to scale effectively and deliver competitive performance

compared to numpywren and Dask `distributed`. What's more, WUKONG 3.0 is still very cost-effective compared to numpywren and Dask `distributed`.

Altogether, the three versions of WUKONG fully satisfy the criteria specified by the thesis statement. WUKONG achieves high performance, near-ideal scalability, and data locality, while being cost-effective and easy-to-use. WUKONG's evaluation demonstrates the effectiveness of decentralized scheduling, task clustering, and delayed I/O.

6.2 Future Directions

The focus of this thesis has been on developing a fast, efficient, easy-to-use, and cost-effective serverless DAG engine. During the design and implementation of this framework, we have created a number of unique techniques and optimizations that have application beyond the scope of WUKONG. In addition, we have identified several future directions for WUKONG itself. We discuss several of these potential future directions below.

6.2.1 Improving Peer-to-Peer Performance

There are several directions in which this body of work can be extended. First, the performance of WUKONG 3.0 can likely be improved substantially. There are a number of optimizations that have been left to the future, due to time constraints. Many of these optimizations involve modifying the way tasks are assigned to workers and minimizing data transfer wherever possible. Beyond this, WUKONG 3.0 can be modified to utilize Boxer [65], a TCP/IP-based Lambda-to-Lambda communication system. The authors of Boxer have reported significantly higher bandwidth and lower latency compared to WUKONG 3.0's current P2P protocol. Consequently, using Boxer for Lambda-to-Lambda communication may yield significant performance improvements.

6.2.2 Taking Advantage of Increased Resource Limitations

AWS Lambda recently increased the resource limitations for AWS Lambda from 2 vCPUs and 3GB of main memory to 6 vCPUs and 10GB of main memory [66]. WUKONG could take advantage of the increased memory to support larger problem sizes, as each individual serverless Executor could hold larger pieces of data in-memory. This would be particularly beneficial for workloads such as GEMM

and SVD2, which typically perform computations on relatively large partitions of the input data. Additionally, the increased vCPUs could enable each individual executor to perform computations on several tasks at once.

6.2.3 New Applications of Decentralized Scheduling

The scheduling techniques developed in this thesis (namely decentralized scheduling) can also be extended to contexts beyond that of Wukong. I have recently started work on another project that leverages decentralized scheduling in order to provide an elastic metadata management system for traditional distributed file systems (i.e., HDFS/HopsFS [67, 68], IndexFS [69], BeeGFS [70], etc.). Specifically, we are implementing the metadata management component of a distributed file system within serverless functions in order to take advantage of the performance and cost advantages of serverless computing and decentralized scheduling. We can also perform metadata caching in these functions by borrowing techniques developed in the InfiniCache project.

6.2.4 Genericized Programming Model

Beyond simply improving the performance of Wukong, it is possible to generalize Wukong's programming model in order to support a much wider array of applications. Consider the existing DAG representation utilized by Wukong. As discussed, the DAGs used by Wukong encode fan-out and fan-in operations. These can be thought of as a specific instance of more generic `fork` and `join` operations. To that end, Wukong can be extended to support arbitrary `fork/join` applications. This would allow users to execute a significantly larger number of workloads on Wukong as the use of Dask would no longer be required. If this generic programming model were integrated into Wukong 3.0, then the framework would provide users with a cost-effective and easy-to-use solution for parallelizing their `fork/join` workloads.

We will also explore other possibilities in order to find a programming model that is best suited for serverless computing. We will need to verify that `fork/join` is powerful and expressive when implementing modern parallel programs such as streaming, machine learning, and analytics algorithms. The ultimate goal is to give users the freedom to write and design workloads and algorithms to execute on Wukong without the dependency on Dask. Removing the dependency on Dask will

also open the door to many other serverless-oriented optimizations, as Dask was not designed for a serverless environment.

6.2.5 Serverless Benchmark Suite

After implementing a more generic programming model, WUKONG will be able to support a wide variety of applications. We can analyze the execution of many real-world workloads on the WUKONG platform with the purpose of conducting a first-of-its-kind serverless workload analysis. In theory, we could apply machine learning models to the data in order to improve scheduling decisions, reduce data movement, and develop new optimizations for the platform.

6.2.6 A Serverless Supercomputer

If the performance of WUKONG 3.0 were to be improved, and if WUKONG could also be extended to support a generic `fork/join` programming model, then this would allow for an even greater goal to be realized. By combining a fully-serverless, easy-to-use, and generic compute substrate like WUKONG 3.0 with a storage substrate built using serverless functions (e.g., InfiniCache [28]), we could develop a so-called *serverless supercomputer*. This would be a self-contained, fully-serverless execution engine. Intermediate data would be stored in the serverless storage substrate while the compute substrate provides an elastic and cost-effective execution environment.

6.3 Additional Work

InfiniCache. In addition to my work on WUKONG, I have also worked on the InfiniCache project. InfiniCache is a first-of-its-kind in-memory object cache built using serverless functions. InfiniCache uses the memory resources of serverless functions to store data, thereby providing a storage system that is elastic, highly available, and cost-effective. My contributions to this project include creating a MapReduce framework and several MapReduce applications (WordCount, TeraSort, and Grep) for use in evaluating the performance of InfiniCache. I also aided in debugging and made several *small* contributions to the main codebase. InfiniCache is related to WUKONG in that both use serverless functions to implement their core functionality. As we mentioned above, integrating WUKONG and InfiniCache would be a step towards a so-called serverless supercomputer.

6.4 Copyright Notice

Appendix A: Benchmark Code Snippets and DAGs

In this Appendix section, we list the code snippet for each benchmark that we used along with exemplar DAG generated under small problem size (for visibility and illustration purpose).

Code:

```
X = da.random.random((8_192_000, 100),
                     chunks=(10000, 100))
u, s, v = da.linalg.svd_compressed(X, k=5)
res = c.compute(v)
```

DAG for illustration: 10x2, chunks=(5x2)

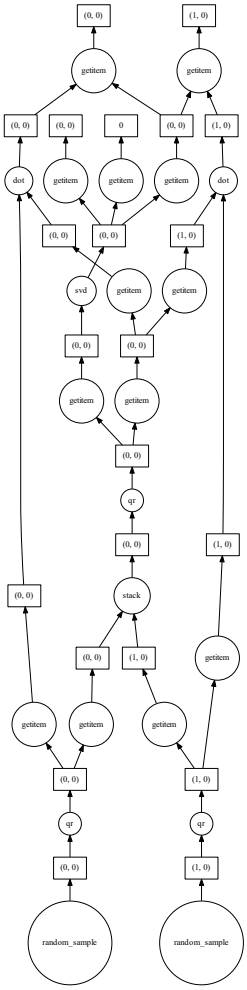


Figure A.1: The DAG of tall-and-skinny SVD (SVD1).

Code:

```
X = da.random.random((256_000, 256_000),
                     chunks=(5000, 5000))
u, s, v = da.linalg.svd_compressed(X, k=5)
res = c.compute(v)
```

DAG for illustration: 200x200, chunks=(50x50)

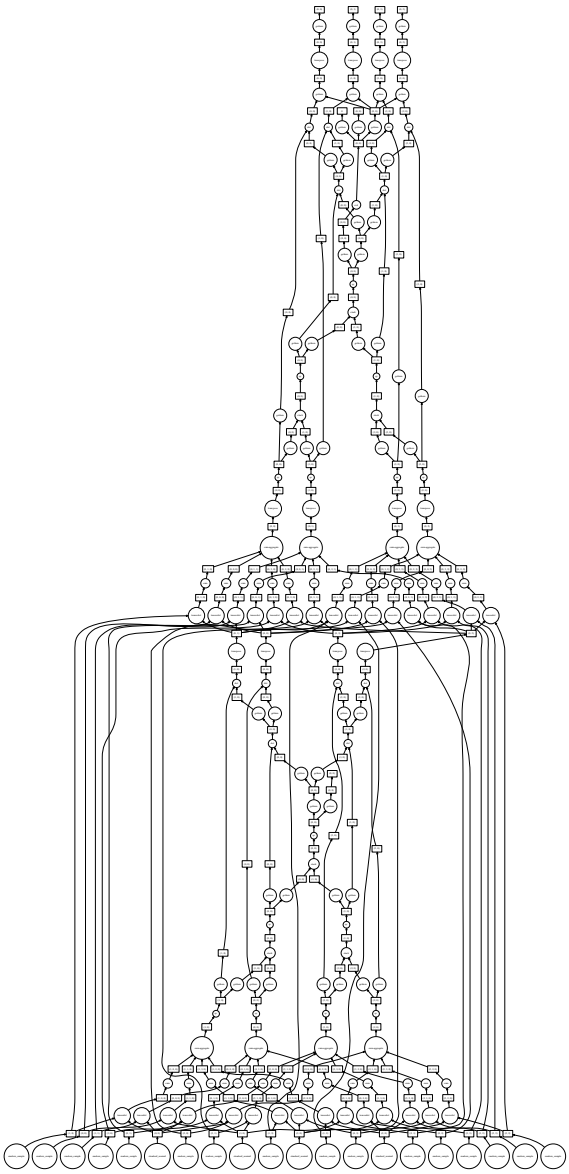


Figure A.2: The DAG of square SVD (SVD2).

Code:

```
X = da.random.random((67_108_864, 128),
                     chunks = (16384, 128))
q, r = da.linalg.tsqr(X)
res = r.compute()
```

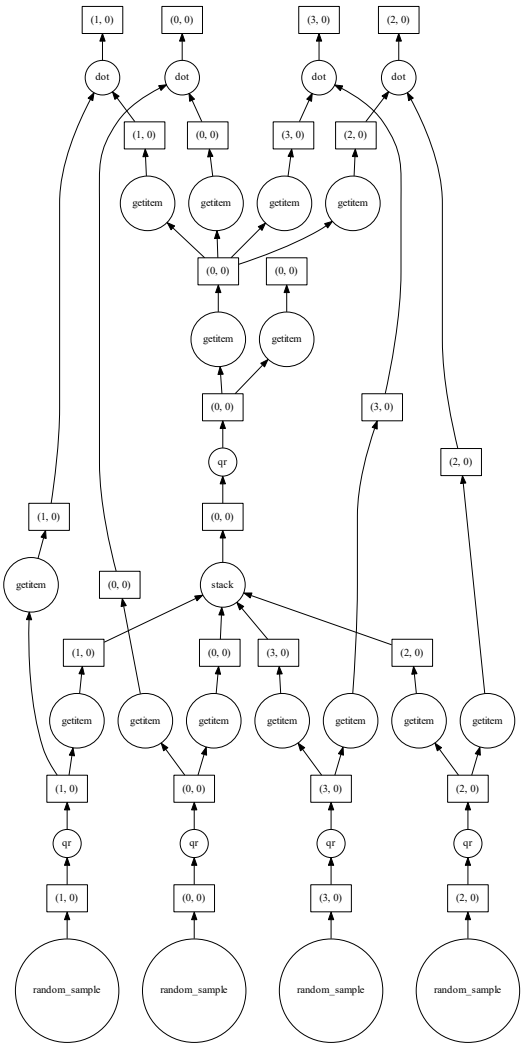


Figure A.3: DAG of tall-and-skinny QR factorization (TSQR). 32,768 x 128, chunks=(8,192 x 128)

Code:

```
X = da.random.random((25_000, 25_000),
                     chunks = (2000, 2000))
Y = da.random.random((25_000, 25_000),
                     chunks = (2000, 2000))
z = da.matmul(X, Y)
z.compute()
```

DAG for illustration: 10x10, chunks=(5x5)

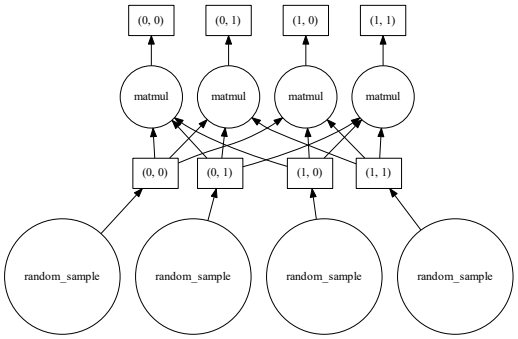
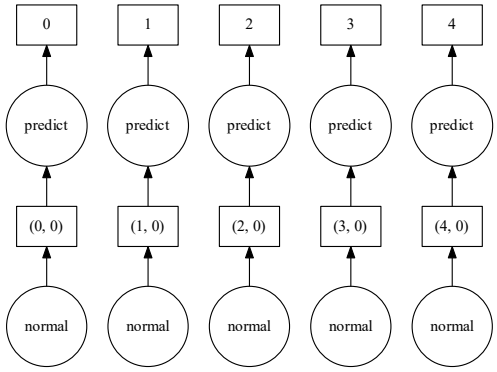


Figure A.4: DAG of GEMM.

Code:

```
X,Y = sklearn.datasets.make_classification(n_samples=1000)
clf = ParallelPostFit(SVC(gamma='scale'))
clf.fit(X, Y)

X,Y = dask_ml.datasets.make_classification(
        n_samples = 32_768_000,
        random_state = 32_768_000,
        chunks = 32_768_000 // 160)
ans = clf.predict(X).compute()
```



DAG for illustration: n=100, chunks=20

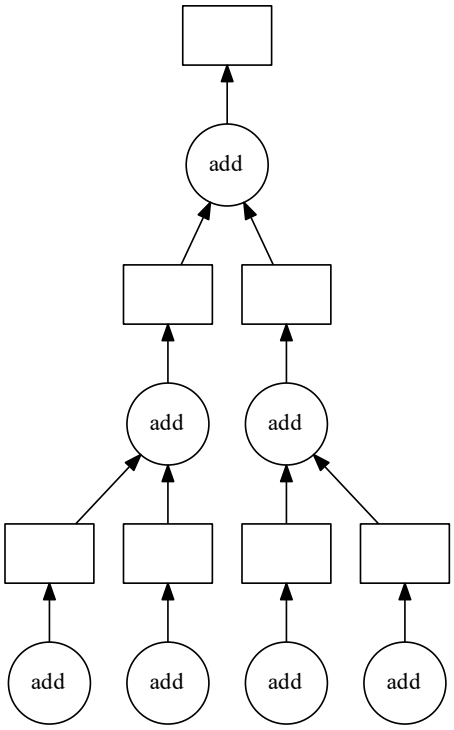
Figure A.5: DAG of SVC.

Code:

```
def add(x,y):
  time.sleep(0.5)
  return x + y

L = range(1024)
while len(L) > 1:
  L = list(map(delayed(add), L[0::2], L[1::2]))

L[0].compute()
```



DAG for illustration: len(L)=8

Figure A.6: DAG of tree reduction.

Bibliography

Bibliography

[1] "Tree Reduction benchmark," https://matthewrocklin.com/blog/work/2017/07/03/scaling.

[2] B. Carver, J. Zhang, A. Wang, and Y. Cheng, "In search of a fast and efficient serverless dag engine," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, 2019, pp. 1–10, © 2019 IEEE. Reprinted, with permission, from Benjamin Carver and Jingyuan Zhang and Ao Wang and Yue Cheng, In Search of a Fast and Efficient Serverless DAG Engine, In Search of a Fast and Efficient Serverless DAG Engine, 2019.

[3] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–15. [Online]. Available: https://doi.org/10.1145/3419111.3421286

[4] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," EECS Department, University of California, Berkeley, Tech. Rep., 2019.

[5] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *USENIX ATC 18*, 2018.

[6] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," in *USENIX ATC 18*, 2018.

[7] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "{SAND}: Towards High-Performance Serverless Computing," 2018, pp. 923–935. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/akkus

[8] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *USENIX NSDI 19*, 2019.

[9] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "numpywren: serverless linear algebra," *arXiv preprint arXiv:1810.09679*, 2018.

[10] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *ACM SoCC '17*, 2017.

[11] "HyperFlow: a scientific workflow execution engine," https://github.com/hyperflow-wms/hyperflow.

[12] "AWS Step Functions," https://aws.amazon.com/step-functions/.

[13] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *ACM SoCC '18*, 2018.

[14] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *USENIX NSDI 17*, 2017.

[15] "Fission Workflows," https://github.com/fission/fission-workflows.

[16] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions," *Future Generation Computer Systems*, Nov. 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X1730047X

[17] P. López, M. S. Artigas, G. París, D. B. Pons, Á. R. Ollobarren, and D. A. Pinto, "Comparison of faas orchestration systems," *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 148–153, 2018.

[18] Fission, "Serverless functions for kubernetes," https://fission.io/.

[19] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the faas track: Building stateful distributed applications with serverless architectures," in *Proceedings of the 20th International Middleware Conference*, ser. Middleware '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 41–54. [Online]. Available: https://doi.org/10.1145/3361525.3361535

[20] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: ACM, 2019, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/3357223.3362711

[21] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *USENIX ATC 19*, 2019.

[22] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, April 2018, pp. 257–262.

[23] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "Funcx: A federated function serving fabric for science," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 65–76. [Online]. Available: https://doi.org/10.1145/3369583.3392683

[24] A. Ellis, "Introducing functions as a service," https://blog.alexellis.io/introducing-functions-as-a-service/.

[25] M. Thömmes, "Squeezing the milliseconds: How to make serverless platforms blazing fast!" https://goo.gl/zvqtBP.

[26] "Google Cloud Functions," https://cloud.google.com/functions/.

[27] "Azure Functions," https://azure.microsoft.com/en-us/services/functions/.

[28] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 267–281. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/wang-ao

[29] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 561–577. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/moritz

[30] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *ACM SOSP '13*, 2013.

[31] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 351–364. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465386

[32] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing," *Parallel Comput.*, vol. 38, no. 1-2, pp. 37–51, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.10.003

[33] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive parallel programming in python," in *ACM HPDC '19*, 2019.

[34] "Serverless: Build and run applications without thinking about servers," https://aws.amazon.com/serverless/.

[35] J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," 2018. [Online]. Available: http://arxiv.org/abs/1812.03651

[36] J. Gray, "Why do computers stop and what can be done about it?" 1985.

[37] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *USENIX HotCloud 16*, 2016.

[38] "2018 Serverless Community Survey: huge growth in serverless usage," https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/.

[39] "Alibaba Cluster Trace Program (New 2018 Version," https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md.

[40] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *ACm IWQoS '19*, 2019.

[41] Y. Cheng, Z. Chai, and A. Anwar, "Characterizing co-located datacenter workloads: An alibaba case study," in *ACM APSys '18*, 2018.

[42] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[43] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX NSDI 12*, 2012.

[44] "Dask: Scalable Analytics in Python," https://dask.org/.

[45] N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM Review*, 2009. [Online]. Available: http://arxiv.org/abs/0909.4061

[46] "Dask: Scalable Machine Learning in Python," https://dask-ml.readthedocs.io/en/latest/#.

[47] C. Munns, "Announcing improved VPC networking for AWS Lambda functions." [Online]. Available: https://aws.amazon.com/blogs/compute/announcing-improved-vpc-networking-for-aws-lambda-functions/

[48] "Developing Convex Optimization Algorithms in Dask parallel: math is fun," https://matthewrocklin.com/blog/work/2017/03/22/dask-glm-1.

[49] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, p. 273–297, Sep. 1995. [Online]. Available: https://doi.org/10.1023/A:1022627411411

[50] "Scikit-learn Support Vector Machines," https://scikit-learn.org/stable/modules/svm.html#svm-classification.

[51] K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, H. Cui, and E. Y. Chang, "Parallelizing support vector machines on distributed computers," in *Advances in Neural Information Processing Systems 20*, J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, Eds. Curran Associates, Inc., 2008, pp. 257–264. [Online]. Available: http://papers.nips.cc/paper/3202-parallelizing-support-vector-machines-on-distributed-computers.pdf

[52] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[53] "Apache Hadoop," https://hadoop.apache.org/.

[54] "MPI Forum," https://www.mpi-forum.org/.

[55] "kubernetes: Production-Grade Container Orchestration," https://kubernetes.io/.

[56] "Serverless: Cold Start War," https://mikhail.io/2018/08/serverless-cold-start-war/.

[57] "AWS Fargate: Serverless compute for containers," https://aws.amazon.com/fargate/.

[58] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 427–444. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic

[59] "Dask User Interfaces," https://docs.dask.org/en/latest/user-interfaces.html.

[60] "Dask Task Graphs," https://docs.dask.org/en/latest/graphs.html.

[61] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[62] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, 1999.

[63] "Serverlessnetworkingclients," https://networkingclients.serverlesstech.net/.

[64] "UDT," https://udt.sourceforge.io/.

[65] M. Wawrzoniak, I. Müller, R. Fraga Barcelos Paulus Bruno, and G. Alonso, "Boxer: Data analytics on network-enabled serverless platforms," 2021-01, 11th Annual Conference on Innovative Data Systems Research (CIDR 2021); Conference Location: online; Conference Date: January 11-15, 2021; The conference lecture will be held on January 12, 2021. Due to the Coronavirus (COVID-19) the conference will be conducted virtually.

[66] "New for AWS Lambda – Functions with Up to 10 GB of Memory and 6 vCPUs," https://aws.amazon.com/blogs/aws/new-for-aws-lambda-functions-with-up-to-10-gb-of-memory-and-6-vcpus/.

[67] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[68] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, "Hopsfs: Scaling hierarchical file system metadata using newsql databases," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 89–104. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi

[69] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 237–248.

[70] "BeeGFS," https://www.beegfs.io/c/.

Curriculum Vitae

Ben Carver received his Bachelor of Science in Computer Science from George Mason University in 2020. He enrolled in the Accelerated Master's program and began taking graduate classes in Spring 2019. After graduating in 2020, Ben continued working on his Master's degree.