Ionut Predoaia

Serverless Data Analytics

Final Master's Project

Directed by Dr. Pedro García-López

University Master's Degree in

Computer Security Engineering and Artificial Intelligence

Universitat Rovira i Virgili

Tarragona

2023

Abstract

Serverless computing has shown vast potential for data analytics applications, especially for embarrassingly parallel workloads. Nevertheless, little consideration has been given in the literature to porting stateful applications requiring shared state to serverless. This work closes this gap by exploring hardware resource disaggregation in serverless computing, with the aim of porting to serverless stateful machine learning algorithms, i.e., k-means clustering and logistic regression. A set of guidelines, challenges and limitations of porting stateful machine learning algorithms to serverless are presented in this work. Running stateful applications on serverless architectures inherently induces overheads, as serverless functions are not directly network-addressable, hence one must rely on a remote storage service for storing the shared state. In this work, the feasibility of solving stateful machine learning algorithms is evaluated, and furthermore, optimization techniques are proposed to enhance their feasibility. The performance, scalability and overheads of the stateful machine learning algorithms running with serverless architectures have been evaluated. The serverless implementation of the k-means algorithm has achieved an 87-fold speedup compared to a sequential implementation of the algorithm. In terms of scalability, the serverless implementation has achieved a scale-up factor of 0.91 with 100 concurrent serverless functions. To raise the feasibility of the serverless implementation, intra-function parallelism has been employed as an optimization technique to parallelize the execution of the serverless functions, achieving up to 68% improved performances.

Keywords: Serverless, Machine Learning, Big Data

List of Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Chapter 1

Introduction

The rise of serverless computing has widely propagated the democratization of massive-scale data parallelism. By leveraging serverless computing, cloud users can today launch thousands of concurrent stateless functions to run big data analytics workloads, without the need of complex cluster management and the burden of managing and provisioning cloud resources. One can seamlessly launch thousands of cores on demand, to enable the parallel execution of machine learning algorithms over data sets in the order of terabytes, that could typically not be stored on a single machine.

A multitude of prior works have been carried out in the context of embarrassingly parallel data analytics workloads. However, a limited body of literature exists today that addresses stateful distributed applications with serverless architectures. This research project aims to close this gap by exploring hardware resource disaggregation in serverless computing with the purpose of porting stateful machine learning algorithms to serverless. Note that stateful machine learning algorithms are by nature profound examples of stateful applications. As such, two stateful machine learning algorithms requiring shared state will be ported to serverless in this research project, k-means clustering and logistic regression. For instance, the k-means clustering algorithm is highly parallelizable, however, it requires regular communication at the end of each iteration to update and retrieve the new centroids.

The research area of this research project lies at the intersection of machine learning and distributed computing, specifically, serverless computing. Its objective is to evaluate the feasibility of solving stateful applications (i.e., iterative machine learning algorithms) with serverless architectures. Although one can enable big data analytics workloads by launching thousands of concurrent serverless functions, there are limitations related to connectivity and regarding the management of shared state between serverless functions. Accessing remote resources will always yield a latency penalty, regardless of the latency improvements that have been and will be achieved.

As serverless functions are not directly network-addressable, they cannot communicate with each other to share the state related to a machine learning algorithm (e.g., the centroids in k-means clustering, the gradients in logistic regression). Consequently, one must rely on a remote storage service for storing the shared state, which will be used by the serverless functions to access and update the shared state, over the network. This inherently induces communication overheads, as multiple serverless functions must access and update the shared state from the remote storage service. In the context of stateful machine learning algorithms, the serverless functions would need to access and update the shared state at every iteration of the algorithm, and consequently, significant overheads can result in the case of executing hundreds of iterations.

In this research project, the Lithops[1] multi-cloud distributed computing framework will be used for porting stateful machine learning algorithms to serverless via AWS Lambda Functions. The remote storage service stores mutable shared state, as it has been opted to store the shared state in disaggregated memory (i.e., the Redis[2] in-memory data structure store) rather than disaggregated storage, for the reason that it ensures low latency and high throughput.

This research project aims to address the following three research questions:

RQ1 What challenges and limitations are encountered when porting stateful machine learning algorithms to serverless?

The first research question aims to address how can stafeful machine learning algorithms be ported to serverless, and what challenges and limitations are encountered during this process. To address this question, stateful machine learning algorithms will be implemented through stateless serverless functions that share state via a remote storage service. The aim is to educate cloud users about the challenges and limitations of porting stateful data analytics workloads to serverless.

RQ2 What are the overheads when running stateful machine learning algorithms with serverless architectures?

The second research question aims to identify and measure the overheads that result from porting stateful machine learning algorithms to serverless. Executing stateful machine learning algorithms on serverless architectures inherently induces communication overheads, as the serverless functions must access and update the shared state from the remote storage service at every iteration. Furthermore, launching serverless functions and retrieving the output of serverless functions yields additional overheads. Moreover, significant overheads can result whenever synchronizing the execution of serverless functions at each iteration of the stateful machine learning algorithms.

RQ3 What optimizations can be made to improve the performance of stateful machine learning algorithms running on serverless architectures?

The third research question aims to propose optimizations that make it feasible from a performance perspective to run stateful machine learning algorithms on serverless architectures. The purpose of the optimizations is to compensate for the overheads that slow down the execution of the algorithms. Optimization techniques will be proposed and evaluated for improving the performance of the stateful algorithms when running on serverless architectures. Cloud providers typically provide one virtual CPU (vCPU) per serverless function, however, a few cloud providers have upgraded their offering by providing multiple vCPU per serverless function (e.g., AWS Lambda and Google Cloud Functions). Thus, engineers can adopt a so-called intra-function parallelism pattern to leverage the multi-core computing resources of a serverless function

[1] `https://lithops-cloud.github.io`

[2] `https://redis.io`

for parallelizing its execution via threads or processes. Therefore, one of the optimization techniques that will be explored will employ intra-function parallelism to achieve an improved performance.

The technical contributions of this research project are the following:

- Sequential (single-machine and serial) implementation of stateful machine learning algorithms in Python
- Serverful (single-machine and parallel) implementation of stateful machine learning algorithms with the Python Multiprocessing API and with Lithops
- Serverless implementation of stateful machine learning algorithms with Lithops, with two versions being implemented for each algorithm
- Validation scripts that verify the correctness of the implemented stateful machine learning algorithms by comparing their results with the *scikit-learn*[3] machine learning library

The research contributions of this research project are the following:

- Guidelines, challenges and limitations when porting stateful machine learning algorithms to serverless
- Experiments for identifying and measuring the overheads when running stateful machine learning algorithms with serverless architectures
- Experiments for evaluating the feasibility from a performance perspective of the serverless implementations by comparing them to the sequential and serverful implementations
- Proposing and evaluating optimization techniques that can improve the performance of the algorithms

[3] `https://scikit-learn.org`

Chapter 2

Background

2.1 Cloud and Serverless Computing

In recent years, cloud adoption has widely expanded in the IT industry, with the purpose of reducing cost, increasing collaboration, mitigating risks via disaster recovery, and achieving scalability and data availability.

Before the rise of cloud computing, IT organizations deployed software applications and systems on their privately owned physical infrastructures. For carrying out the deployment, system administrators prepared the hardware, installed the operating system and the drivers for each physical machine, prepared the network, and deployed the software. The described approach is referred to as on-premises, as software is deployed to physical hardware that is located on the premises of the organization. Alternatively, this is referred to as private cloud, as the hardware is privately owned and is not made available to the general public. One main shortcoming of this approach is that large capital investments are required at the beginning of software projects for purchasing hardware. Additionally, another drawback is that the hardware is being provisioned at a fixed capacity. As resources are being provisioned at a fixed capacity, this is problematic in the case of fluctuating workloads. Therefore, private clouds are inefficient, as one may have more computing resources than required for handling the traffic of a service, or may have less than needed to be able to support unforeseen increases in traffic.

Cloud computing has become a buzzword in the IT industry, and it can be considered a vague term, as it has a variety of definitions. Amazon Web Services (AWS) has defined cloud computing as the on-demand delivery of IT resources over the Internet with pay-as-you-go pricing, without the need of buying, owning, and maintaining physical data centers and servers [1]. In [2], cloud computing has been defined as a computing technique in which IT services are offered by massive low-cost computing units that are connected by IP networks. Furthermore, it can be defined as a set of network-enabled services that provide scalable and inexpensive computing infrastructures on demand, that are easily accessible [3]. As stated by IBM Cloud [4], cloud computing can be defined as:

> Cloud computing is on-demand access, via the internet, to computing resources—applications, servers (physical servers and virtual servers), data storage, development tools, networking capabilities, and more—hosted at a remote data center managed by a cloud services provider (or CSP). The CSP makes these resources available for a monthly subscription fee or bills them according to usage.

Cloud computing encompasses the applications provided over the Internet as services and the hardware and systems software providing the services. The term Software as a Service (SaaS) is used to refer to the services, whereas the hardware and systems software from the data centers are called a *cloud* [5]. A cloud that is made publicly available in a pay-as-you-go manner is called a public cloud, where the service being sold is utility computing (e.g., AWS, Microsoft Azure). Cloud computing represents the sum of SaaS and utility computing, and typically it does not include private clouds.

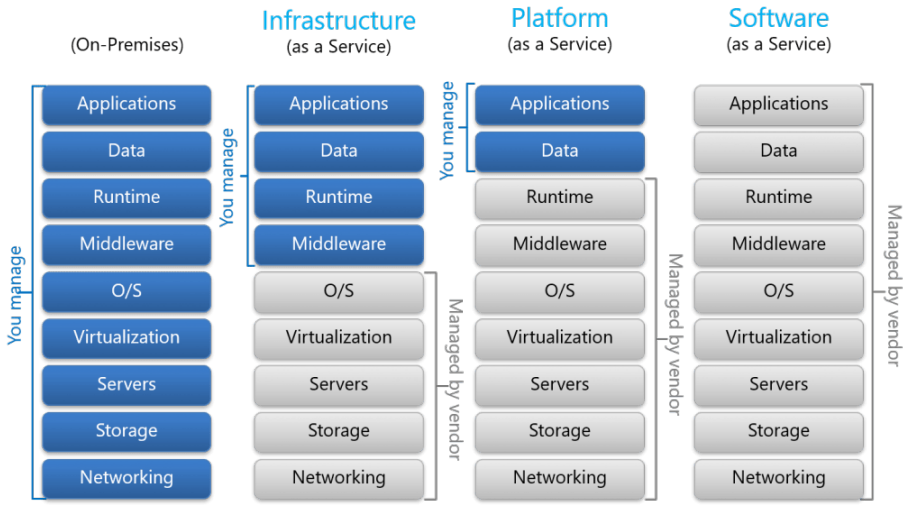

Figure 2.1: Cloud computing service models

There are three principal (or basic) cloud computing service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). The functions implemented by the cloud service provider for the principal cloud computing service models are outlined in Figure 2.1. For traditional IT, i.e., on-premises, the entire cloud stack is managed by the customer. Note that each cloud service model provides less management responsibility for the customer, than the previous cloud service model. Accordingly, the customer has less management responsibility with PaaS than with IaaS, and furthermore, no management responsibility with SaaS, as the entire cloud stack is managed by the cloud service provider.

IaaS provides essential servers, compute, storage and networking resources, and the customer is able to access on demand the underlying cloud infrastructure. The infrastructure resources delivered by cloud service providers are shared among multiple customers, which are alternatively called tenants. Virtualization is used to prevent conflicts and collisions related to software applications of different tenants deployed on the same physical machine. Customers can commonly self-provision infrastructures via a Graphical User Interface (GUI), or via an Application Programming Interface (API). Amazon Elastic Compute Cloud[1] (Amazon EC2) and Google Compute Engine[2] (GCE) are examples of IaaS.

PaaS provides a cloud platform that hosts a complete suite of infrastructures and software systems, i.e., networks, servers, storage, operating system software, databases,

[1] `https://aws.amazon.com/ec2`
[2] `https://cloud.google.com/compute`

development tools, and runtime environments. Commonly referred to as application Platform as a Service (aPaaS), it offers a scalable cloud platform for developing, deploying, running and managing applications. Thus, PaaS can be considered an operating system in the cloud [6]. Heroku[3], AWS Elastic Beanstalk[4] and Google App Engine[5] are examples of PaaS.

SaaS provides software services in the cloud that can be accessed by customers at any time and place. With SaaS, customers are no longer required to install and run software applications on physical or virtual machines (VMs). Furthermore, it removes the customer's burden for software installations, maintenance, upgrades, and patches, as these operations are outsourced to the cloud provider. Customers rent the use of a software service in a pay-as-you-go manner, that can be accessed over the Internet, commonly within a web browser. Google Gmail, Microsoft 365 and Slack[6] are examples of SaaS.

In the following, two emerging cloud computing service models will be presented, Backend as a Service (BaaS) and Function as a Service (FaaS), which represent the core serverless offerings of public cloud providers. Before introducing BaaS and FaaS, one must first be acquainted with serverless computing.

Serverless computing, or more simply Serverless, is an emerging paradigm for the deployment of software applications in the cloud. The term *serverless* can be confusing, as one may think that there are no servers involved. However, servers do exist, but their provisioning and management is handled by the cloud provider. Therefore, serverless can be indeed described as server worry-less. With serverless computing, customers simply upload their code to a cloud platform, and then, the platform executes the code on the customer's behalf as needed at any scale [7]. Note that customers only pay for the compute resources used for the duration in which their code is invoked. As stated in [8], $serverless\ computing = FaaS + BaaS$.

FaaS is a serverless computing model in which the unit of computation is a function that is executed in response to triggers, e.g., events and HTTP requests [9]. Serverless functions have the ability to scale to zero and have almost infinite on-demand scalability. The functions are run in stateless compute containers, that are created and destroyed by the FaaS provider depending on the runtime need. Although FaaS functions have storage available, they are considered naturally stateless, as the internally stored state of a function is not guaranteed to be persisted across multiple invocations [10]. Therefore, if persistent state is required for a function, one should persist it to an external storage (e.g., Amazon S3). The initialization of a function has a start-up latency and it can either be a warm start or a cold start. A cold start describes the phenomenon that a function that has not been executed yet or for a long time takes longer to start up. In contrast, a warm start describes the case in which a function has been recently executed and the instance and host container are reused from the previous invocation. Note that the execution duration of a function is limited to a short amount of time, and after the timeout, the function is terminated (e.g., the maximum timeout for AWS Lambda is 15 minutes). Examples of FaaS include AWS Lambda,

[3] `https://www.heroku.com`
[4] `https://aws.amazon.com/elasticbeanstalk`
[5] `https://cloud.google.com/appengine`
[6] `https://slack.com`

Google Cloud Function, IBM Cloud Function, Microsoft Azure Functions and Alibaba Cloud Function Compute.

BaaS is a serverless computing model in which engineers outsource the behind-the-scenes aspects of web and mobile applications. Most commonly, BaaS providers offer services for database management, cloud storage, user authentication and push notifications. The development of application backends can be significantly facilitated by leveraging BaaS services. Examples of cloud storage BaaS include Amazon S3[7] and IBM COS[8].

2.2 Hardware Resource Disaggregation

Serverless computing has brought the shift towards a disaggregated data center, where each resource type (e.g., compute, memory, storage) is built as a standalone resource *blade*, and a network fabric interconnects the resource blades [11]. Hardware resource disaggregation is a technique for decomposing general-purpose monolithic servers into segregated, network-attached resource pools, each of which can be built, managed, and scaled independently [12]. The network is the crucial element in a disaggregated data center, as it controls the overall performance of the data center. The architectural differences between a traditional data center and a disaggregated data center are captured in Figure 2.2.

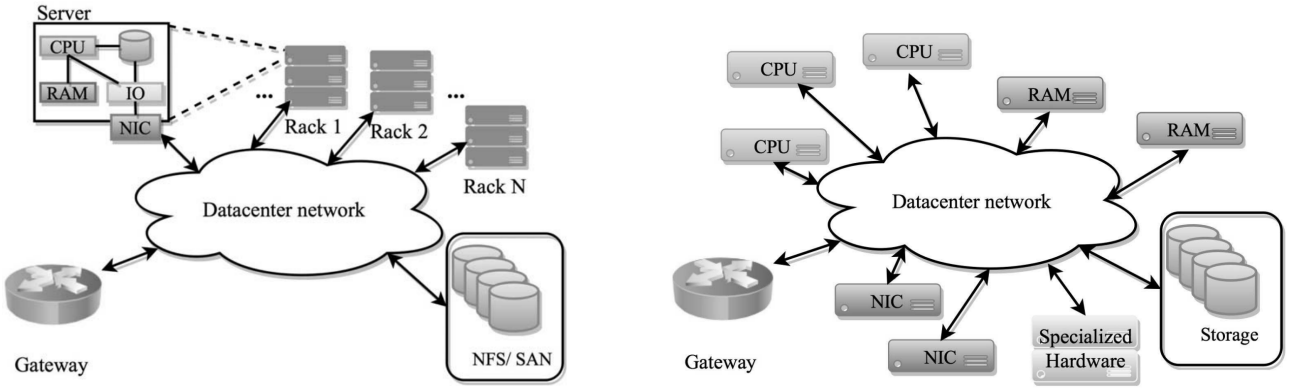

Figure 2.2: Comparison of data center architectures [13]
(a) Traditional data center architecture, (b) Disaggregated data center architecture

The failure of one type of resource (e.g., CPU) in a monolithic server may force the entire server to fail, as other resources in the server may become unusable. In contrast, a disaggregated architecture improves reliability, as different resource types may fail in a more independent way due to the decoupling of resources [14]. By disaggregating

[7] `https://aws.amazon.com/s3`
[8] `https://www.ibm.com/cloud/object-storage`

hardware resources, the resource utilization efficiency and energy consumption of data centers are improved [13]. Additionally, disaggregation makes data centers more modular and flexible, by providing fine-grained control over resources [13]. Disaggregated hardware resources are not without drawback, as accessing remote resources always yields a latency penalty, no matter how fast networks become [15].

2.2.1 Monolithic Server

Software components and applications are deployed to a monolithic server in the Serverful hosting pattern. With the Serverful hosting pattern, cloud customers maintain control over both, deployment stack and scaling configuration management [16]. Throughout this research project, a bare metal cloud offering (i.e., Amazon EC2) is used to set up a monolithic server to which serverful implementations of machine learning algorithms will be deployed.

Amazon Elastic Compute Cloud (Amazon EC2) provides access to reliable, scalable infrastructure on demand. Using Amazon EC2 eliminates the need of investing in hardware up front, as one can launch as many VMs as needed. The cloud hosted VMs are called instances.

Amazon EC2 provides a varied range of types of instances that are optimized to fit different use cases [17]. An instance type is characterized by its configuration, which is defined by the number of vCPU, the amount of memory measured in gibibyte (GiB), the storage used (e.g., Amazon EBS[9]), and the storage and network bandwidth measured in gigabits per second (Gbps). The essential types of instances are grouped as follows:

- **General Purpose** - provide a balance of compute, memory and networking resources, and can be used for a variety of diverse workloads
- **Compute Optimized** - are ideal for compute-bound applications that can benefit from high-performance processors
- **Memory Optimized** - are designed to deliver fast performance for workloads that process large data sets in memory
- **Storage Optimized** - are best suited for workloads that require high, sequential read and write access to very large data sets on local storage

2.2.2 Disaggregated Compute

In this research project, disaggregated compute resource services are used to realize the implementation of serverless machine learning algorithms. By leveraging disaggregated compute resource services like AWS Lambda, one can benefit from hundreds

[9] `https://aws.amazon.com/ebs`

Memory (MB)	vCPU
128 - 1769	1
1770 - 3538	2
3539 - 5307	3
5308 - 7076	4
7077 - 8845	5
8846 - 10240	6

Table 2.1: Amount of vCPU per memory

of compute units (i.e., CPUs) to parallelize the implementation of machine learning algorithms, where one CPU typically equates to one serverless function. Note that serverless functions are not directly network-addressable, hence concurrent serverless functions cannot communicate with each other to share state. Stateful applications are parallelized by leveraging stateless functions that either share state through disaggregated storage (e.g. Amazon S3) or disaggregated memory to overcome the throughput and speed bottlenecks of slow disk-based storage services [18].

AWS Lambda[10] is a FaaS, event-driven and disaggregated compute service that enables engineers to run code that automatically scales in response to events. Customers are not required to provision and manage servers, and they only pay for the duration for which functions are running. The timeout, i.e., the maximum amount of time that a Lambda function can run is 15 minutes. When the timeout has passed, the function is terminated.

One can deploy code written in several programming languages to AWS Lambda, however, Python is the only programming language used throughout this research project. Code can be deployed to a Lambda function either by uploading a zip archive that contains application code and its dependencies, or by uploading a container image that packages the code and its dependencies [19].

A Lambda function can be configured to run in a virtual private cloud (VPC) via Amazon VPC[11]. With Amazon VPC, one can launch AWS resources in logically isolated virtual networks. A virtual network is similar to a traditional network that would be operated in one's own data center, with the benefits of using the scalable infrastructure of AWS [20].

Memory is the core lever available for controlling the performance of a Lambda function. One can allocate between 128 MB to 10240 MB of memory to a Lambda function. The amount of memory determines the amount of vCPU available to a function. An increase in memory is correlated with a proportional increase in the amount of vCPU, thus increasing the overall computational power available to a function. If a function is memory-bound, CPU-bound, or network-bound, then increasing the amount of memory can significantly improve the function's performance. Note that a function with 1769 MB of memory has the equivalent of one full vCPU [21]. Table 2.1 displays the amount of vCPU that is allocated to a function depending on its memory. One

[10] `https://aws.amazon.com/lambda`

[11] `https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html`

can leverage the multiple vCPU to parallelize the execution of a Lambda function. For parallelizing the execution of a Lambda function based on Python code, there is the limitation that one can only use processes and pipes from the Python Multiprocessing API, therefore queues and pools cannot be used [22].

2.2.3 Disaggregated Storage

The data sets used by the machine learning algorithms that have been implemented in this research project are stored in a disaggregated storage resource service.

Amazon Simple Storage Service (Amazon S3) is a BaaS, disaggregated object storage service that provides scalability, data availability, security, and performance. Data is stored in Amazon S3 as objects within buckets, where an object is a file and any metadata that describes the file, and a bucket is a container for objects [23]. When creating a bucket, one must choose the AWS Region in which the bucket will reside. Note that an AWS Region represents a separate geographic area (e.g. Europe Central, US West).

An object is uniquely identified within a bucket by a key and optionally by a version ID if versioning is enabled. Object versioning allows for restoring objects that are accidentally deleted or overwritten, as one can preserve multiple versions of an object from a bucket. By default, buckets and objects are private and can be made accessible by explicitly granting access permissions.

2.2.4 Disaggregated Memory

A disaggregated memory resource service (i.e., Redis) has been used in this research project for storing the shared state of the implemented serverless machine learning algorithms.

Remote Dictionary Server (Redis) is an open source in-memory data structure store that is used as a distributed in-memory key-value database, cache and message broker. Redis is single-threaded and it has high performance. It typically holds the database entirely in memory, and the disk can optionally be used for persistence.

Redis is installed on a machine, which will be called the Redis server, or alternatively the Redis node. To execute operations on the Redis server, one must run Redis commands. A Redis client must first establish a connection with the Redis server for launching commands. Different Redis commands are used depending on the type of data structure that is operated on. A wide range of data types is supported in Redis, including strings, lists, sets, sorted sets, hashes, bitmaps, and streams. Table 2.2 describes the primary commands for the LIST type [24].

Command	Description
LRANGE	retrieves a range of elements from a list
LSET	sets the value of an element in a list by its index
LPUSH	inserts elements at the head of a list
RPUSH	inserts elements at the tail of a list
LPOP	retrieves and removes the element from the head of a list
BLPOP	retrieves and removes the element from the head of a list, or blocks until an element is available

Table 2.2: Redis commands for lists

2.3 Parallel and Distributed Computing and Programming APIs

Section 2.3.1 introduces the essential terminology and notions related to parallel and distributed computing that will be used throughout this research project. Next, Section 2.3.2 presents the Python Multiprocessing module, which is used in this research project for delivering process-based parallelism. Furthermore, Section 2.3.3 presents the Lithops distributed computing framework, which is leveraged to facilitate the implementation of algorithms with a serverless architecture. Finally, Section 2.3.4 outlines a set of abstractions for managing the shared state of parallel and distributed algorithms.

2.3.1 Terminology

The fundamental difference between parallel and distributed computing is that in parallel computing, one computer with multiple processors is required to enable parallel processing, whereas in the case of distributed computing, several autonomous computer systems, often located in geographically separate regions, are required.

A workload is called embarrassingly parallel when it can be parallelized into multiple parallel tasks that require little or no communication between each other. An embarrassingly parallel workload is alternatively called embarrassingly parallelizable or perfectly parallel. Examples of embarrassingly parallel workloads are: Monte Carlo simulations for computational physics, hyperparameter tuning for machine learning, and featurization for data science [25].

Data parallelism and task parallelism are two widely used styles of parallelism. In the case of data parallelism, a data set is distributed into partitions across several processors or nodes, each carrying out the same operation or instruction, repeatedly and simultaneously over their assigned partition [26]. Moreover, task parallelism represents

a form of parallelization of a set of sequential tasks into several concurrent tasks across multiple processors or nodes. Note that task and data parallelism are complementary rather than competing programming models, and many problems can benefit from a mixture of the two approaches [27].

To achieve data parallelism, large volumes of data may be required to be transferred to several nodes. As this may bring a significant overhead, it is often preferred to move the computation close to where the data is located rather than to move the data to the computation node [28]. The process of moving the computation to the node in which the data resides is called data locality. The benefit of data locality is that it reduces network congestion and it improves the overall performance of the system [28].

Fork-join parallelism is a parallel computing model which forks (i.e., divides) a sequential control flow into multiple parallel branches of execution, which join (i.e., merge) at a subsequent point and resume the execution sequentially [29].

In the context of serverless computing, a serverless function (alternatively called a cloud function) represents a worker. FaaS offerings such as AWS Lambda and Google Cloud Function provide serverless functions with multi-core computing resources. As a serverless function can have multiple virtual CPU (vCPU), engineers can take advantage of them to parallelize a function's execution via threads or processes. The procedure of parallelizing a function's execution will be called intra-function parallelism [30]. Additionally, each thread or process will be referred to as an inner worker, i.e., a worker that is nested in another worker.

MapReduce is a parallel programming model that is designed for processing and generating large volumes of data via massively parallel computations that utilize thousands of processors at a time [31]. With MapReduce, a data set is split into several chunks (i.e., partitions) which are processed in parallel (*map* phase) on a distributed system based on several nodes called workers (i.e., commodity machines), and finally, the output of each node is collected and passed to a *reduce* procedure (*reduce* phase) which performs a summary operation over the aggregated output (e.g., counting name frequencies). MapReduce has procedures in place to ensure that it tolerates machine failures and stragglers. Note that a straggler is a worker that takes an unusually long time to complete its *map* or *reduce* task [32].

Scalability is a desirable attribute of a distributed system, which is defined as a system's ability to accommodate an increasing number of elements, to process a growing workload gracefully, and to be susceptible to enlargement [33].

2.3.2 Process - Based Parallelism

The Python interpreter makes use of a Global Interpreter Lock (GIL) to ensure that only one thread executes Python bytecode at a time. This has the benefit of making the object model implicitly safe against concurrent access. Note that an interpreter using GIL will always execute exactly one thread at a time, even when being run on multi-core processors.

In Python, the Threading and Multiprocessing modules are used to enable concurrency in programs. However, the Threading module provides limited parallelism

due to the GIL, as only one thread can run at a time. One can achieve parallelism with the Threading module only for I/O-bound tasks, i.e., waiting for the input/output from a user, file, database, or network. To achieve complete parallelism in Python, one must use the Multiprocessing module, which provides process-based parallelism. Via the Multiprocessing module, one can parallelize the execution of CPU-bound tasks, i.e., tasks that involve performing computation and are not I/O bound.

Class	Method	Description
PROCESS	`run()`	represents the process's activity
	`start()`	starts the process's activity
	`join([timeout])`	blocks until the process terminates, or blocks at most *timeout* seconds
CONNECTION	`send(obj)`	sends an object to the other end of the pipe
	`recv()`	retrieves an object sent from the other end of the pipe
POOL	`apply(func[, args[, kwds]])`	calls *func* with arguments *args* and keyword arguments *kwds*
	`map(func, iterable[, chunksize])`	chops the *iterable* into a number of chunks which it submits to the process pool as separate *func* tasks
LOCK	`acquire(block=True, timeout=None)`	acquires the lock
	`release()`	releases the lock
BARRIER	`wait(timeout=None)`	pauses the execution of the process until all processes of the barrier have called this method

Table 2.3: Python Multiprocessing APIs

The Multiprocessing module provides the following essential classes: *Process*, *Pool*, *Lock*, *Barrier*, *Pipe*, and *Queue*. The *Process* class is used to instantiate individual processes, whereas the *Pool* class is used for instantiating pools of worker processes. Note that a *Process* object represents an activity that is run in a separate process. A lock is a mutual exclusion synchronization primitive for processes. A barrier is a synchronization primitive that allows multiple processes to pause their execution until a predefined number of processes reach a certain execution point, and then, all processes are notified and released to continue their execution. Pipes and Queues are used to enable message passing for communication between processes and avoid the need of using synchronization primitives. When a *Pipe* object is instantiated, it returns a pair of *Connection* objects that represent the two ends of the pipe. Note that a pipe allows a connection between two processes, whereas a queue allows multiple producer and consumer processes. Table 2.3 describes the methods from the Python Multiprocessing API [34] that have been used in this research project.

2.3.3 Lithops : A Distributed Computing Framework

The trend of serverless data analytics has been initiated by PyWren[12] [25], a distributed computing framework that enables untrained users to execute single-machine code at massive scale in the cloud. The pioneering work from PyWren has been extended in IBM-PyWren for the purpose of enabling the execution of broader MapReduce jobs [35]. As IBM-PyWren only supported IBM Cloud as a cloud provider, Lithops has continued the work from IBM-PyWren to avoid vendor lock-in.

Lightweight Optimized Serverless Processing (Lithops) is a multi-cloud distributed computing framework that enables engineers to run unmodified single-machine Python code at scale in the main serverless computing platforms, such as AWS, Google Cloud and IBM Cloud. Code is transparently deployed into serverless functions, without requiring knowledge of how the deployment is carried out. This has the benefit of lowering the entry barrier for novice cloud users, who may often be puzzled by the wide array of choices (e.g., VM instance type and cluster size) that can be made regarding the execution of a simple application in parallel [36]. The framework is especially useful for building embarrassingly parallel and MapReduce-like jobs. Furthermore, the framework avoids vendor lock-in by providing portability across multiple cloud providers via its multi-cloud agnostic architecture.

Lithops makes use of a *compute backend* to run parallel jobs and a *storage backend* to store the input and output data of the jobs. One can run Lithops in a serverless context, in which the compute backend is a FaaS platform (e.g., AWS Lambda) and the storage backend is a BaaS service (e.g., Amazon S3), such that the compute and storage resources can scale independently of each other. Alternatively, one can run Lithops on a local machine (i.e., localhost mode), in which the compute backend is represented by local parallel processes and the storage backend is represented by the local storage of the machine.

Figure 2.3 captures the high-level architecture of Lithops. In the following, the main components of Lithops [36] are presented, and additionally, to facilitate the understanding of how they work, the code from Listing 2.1 is used as an example.

```
1 import lithops
2
3 def my_function(x):
4     return x + 7
5
6 if __name__ == "__main__":
7     data = [3, 6, 9]
8     fexec = lithops.FunctionExecutor()
9     fexec.map(my_function, data)
```

Listing 2.1: Lithops example

[12] https://pywren.io

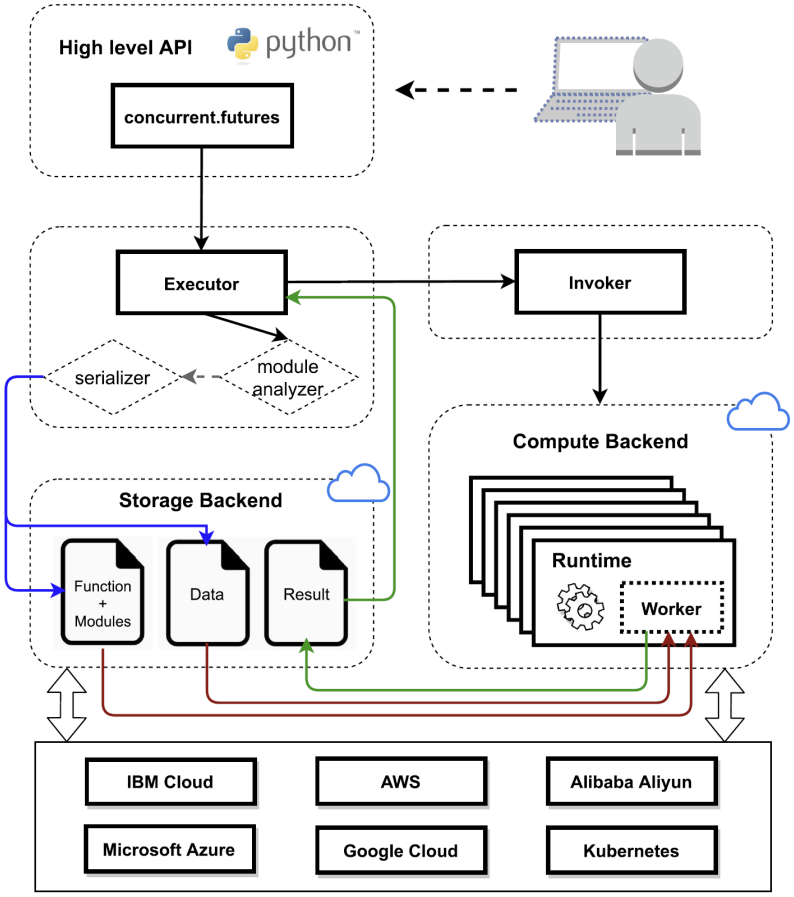

Figure 2.3: High-level architecture of Lithops [36]

The *Executor*, upon a Lithops API call (e.g., the call to the `map` method from line 9), serializes and uploads the single-machine code (e.g., `my_function` from lines 3-4) and the input data (e.g., the `data` variable defined at line 7) from the local machine to the storage backend. Note that the single-machine code is a user-defined function (UDF). At this point, the *Invoker* has the duty of invoking a serverless function for each partition of the input data against the compute backend. Considering the example from Listing 2.1, the *Invoker* performs three invocations of serverless functions against the compute backend, i.e., one for each item from the `data` list defined at line 7. The *Worker* runs in a serverless function on the compute backend, and it fetches the code and input data from the storage backend and it executes the code, and finally, the output is persisted to the storage backend. The *Executor* monitors the storage backend for the output data and when it becomes available, it transfers it to the local machine.

Lithops facilitates the invocation of MapReduce-like jobs via a built-in data partitioner. The partitioner splits a data set into several partitions, where each partition is passed as input to a serverless function. When the storage backend is a BaaS object storage service like Amazon S3, the data set must either be defined as a list of object keys or a list of buckets. When a list of buckets is supplied, Lithops carries out the data discovery of all objects in the buckets, and then automatically partitions them via the partitioner. One can configure a chunk size parameter to specify in how many partitions a data set should be split into. However, if the chunk size parameter is not

Method	Parameters	Description
`call_async()`	• UDF code • data	asynchronously executes a UDF as a serverless function
`map()`	• map UDF code • map data	asynchronously executes a UDF as a parallel serverless function for each partition of the input data
`map_reduce()`	• map and reduce UDF code • map data	asynchronously executes a UDF as a parallel serverless function for each partition of the input data (map phase), and then, executes another UDF as one or more reducers (reduce phase)
`wait()`	• list of futures • when to release	provides synchronization points in the client code by blocking the execution until the condition to release the execution is met (e.g., resume execution only after all results are available in the storage backend)
`get_result()`	• list of futures	fetches the results of the serverless functions that are stored in the storage backend

Table 2.4: Lithops API specification

defined, then each data object from object storage will be considered a partition. Note that the partitioner has the additional benefit of speeding up the process of reading the data set, as the serverless functions read their assigned partition in parallel.

Lithops provides an API that mimics the *concurrent.futures* [37] API from Python. The *concurrent.futures* API delivers a high-level interface for asynchronously executing callables [37]. Note that a *Future* is a non thread-safe, awaitable object and it represents an eventual result of an asynchronous operation [38]. The Lithops API contains five main methods that have been outlined in Table 2.4. There are three methods for running code in serverless functions: `call_async()`, `map()` and `map_reduce()`. Furthermore, there is one method that keeps track of a function's execution, i.e., `wait()`, and another function that downloads the final results of a function from the storage backend, i.e., `get_result()`.

2.3.4 Shared State Abstractions

In Python, each process has its own isolated memory space. The Python Multiprocessing API provides abstractions to enable concurrent processes to share state between each other. Similarly, Lithops contains a Multiprocessing module that provides abstractions to enable concurrent serverless functions to share state with each other.

Note that the Lithops Multiprocessing API mimics the Python Multiprocessing API.

The shared state abstractions from the Python Multiprocessing API store their state values in local memory, whereas the shared state abstractions from the Lithops Multiprocessing API store their state values in a remote memory data store (i.e., Redis). For the implementation of the machine learning algorithms from this research project, the Python Multiprocessing API is used for managing shared state in the serverful implementations, whereas the Lithops Multiprocessing API is used for managing shared state in the serverless implementations. The abstractions that enable shared state between processes and serverless functions are of three types: message passing primitives, shared memory primitives and synchronization primitives. The design decisions for the implementation of the various types of primitives from Lithops [39] will be presented in the following.

Message passing: Pipes are used to enable message passing between a pair of processes. When a *Pipe* object is instantiated, two *Connection* objects are returned, each corresponding to a Redis LIST. The `send()` method of a *Connection* object executes a Redis LPUSH command to insert a data item to the tail of the list, whereas the `recv()` method of a *Connection* object executes a Redis BLPOP command to retrieve and remove the data item from the head of the list or block until there is a data item available. By following this approach, a list associated with a pipe *Connection* object is treated as a FIFO queue. Queues are implemented the same as Pipes, with the difference that more than two processes can push and pop data items to the queue.

Shared memory: Data can be stored in a shared memory map using the *Value* and *Array* classes. *Value* and *Array* objects can store only basic C type values (e.g., `c_bool`, `c_char`, `c_int`, `c_float`, `c_double`). In Lithops, these primitives have been implemented using the LIST type from Redis. Processes can read and write to an *Array* at specific indexes or slices. Furthermore, a *Value* object is treated as an *Array* of size 1. Another shared memory primitive is the *Manager*. Via a *Manager* object, data can be shared between concurrent processes and can be shared over a network between processes that are running on different machines. In the Python Multiprocessing API, a *Manager* object acts as a server process that manages shared objects that can be accessed by other processes via proxies. For storing shared objects, *Managers* support basic Python data types such as lists and dictionaries. In Lithops, the *Manager* implementation of lists and dictionaries is trivial, as Redis provides the LIST and HASH types natively.

Synchronization: In the Lithops Multiprocessing API, locks are implemented using the LIST type from Redis. When a *Lock* object is instantiated, one token is added to the list. The list either contains one or zero tokens. The lock is acquired (i.e., locked) when the list is empty, otherwise, the lock is unlocked when the list contains one token. Each time the `acquire()` method of a lock is executed, the Redis BLPOP command is triggered, which removes the existing token from the list. The `release()` method inserts one token into the list via the Redis LPUSH command. If the list is empty when `acquire()` is executed, the Redis BLPOP command will block until the process that has acquired the lock calls `release()`. The *Barrier* primitive is implemented by using a set of notification lists, which is initially empty. The processes that have called the `wait()` method of a *Barrier* object are blocked until a condition event is satisfied.

The condition event is that all processes arrive at the `wait()` method of the *Barrier* object. When a process calls the `wait()` method and the condition event is not satisfied, the process adds a new empty list to the set containing notification lists. The process then calls the Redis BLPOP command to the new empty list, and the execution of the process is blocked until an element is available in the list. Subsequently, another process calls the `wait()` method and the condition event is satisfied, i.e., all processes have arrived at the `wait()` method. The process that has satisfied the condition adds an element to every notification list from the set. Consequently, the blocked processes by the Redis BLPOP command will resume their execution, as an element is available in each notification list.

2.4 Stateful Machine Learning Algorithms

This section presents the machine learning algorithms that have been chosen to be implemented with serverless architectures in this research project.

Shared state is required for the implementation of iterative machine learning algorithms based on distributed architectures. In contrast, non-iterative algorithms [40] such as Naïve Bayes and K-Nearest Neighbors do not require shared state when being implemented with distributed architectures, as they are embarrassingly parallel. Consequently, two iterative machine learning algorithms will be introduced, k-means clustering, an unsupervised learning algorithm, and logistic regression, a supervised learning algorithm for classification.

2.4.1 K-Means Clustering

K-means is an unsupervised learning algorithm that is one of the most widely used clustering algorithms, due to the simplicity of its implementation and interpretation of its results [41]. There are multiple variants of k-means algorithms, however, throughout this research project the focus is on Lloyd's algorithm [42], which is alternatively called the standard k-means algorithm [43].

K-means is a partitional clustering algorithm, as given data sets are partitioned into a specified number of k disjoint clusters. The k-means algorithm begins by randomly selecting k data points from the data set, and setting them as the initial centroids. A centroid c_i represents the center of the cluster C_i, as it is found by averaging the values of the N_i data points assigned to the cluster, as shown in Eq. 2.1.

$$c_i = \frac{1}{N_i} \sum_{j=1}^{N_i} x_j, \ \forall i \in \{1, 2, ..., k\} \tag{2.1}$$

The algorithm is iterative and it has two phases, the assignment phase and the

update phase [44]. The aim of the assignment phase is to find the cluster membership by partitioning the data set into k clusters. During the assignment phase, for each data point, the distance (i.e., the Euclidean distance metric) from the data point to all centroids is evaluated, and then, the data point is assigned to the cluster of the closest centroid [45]. Eq. 2.2 highlights this operation, as y_i represents the label or the cluster index of the centroid that is the closest to the data point x_i. Next, during the update phase, new centroids are calculated with the new cluster membership, by averaging the data points that have been assigned to each cluster. The assignment phase and the update phase are repeated until convergence, i.e., until there is stability in the cluster membership.

$$y_i = \underset{1 \leq k \leq K}{\operatorname{argmin}} ||x_i - c_k||^2 \tag{2.2}$$

Algorithm 1 contains the pseudocode of the standard k-means clustering. The input of the algorithm is the number of clusters k and the data set $X\{x_1,x_2,...,x_n\}$, where x_i represents a data point. The algorithm partitions the data set into k clusters $(C_1, C_2, ..., C_k)$, and it outputs the centroids $C\{c_1,c_2,...,c_k\}$, where c_i represents the centroid of cluster C_i. Additionally, it outputs the labels $Y\{y_1,y_2,...,y_n\}$, i.e., the cluster index of each data point, where x_i is assigned to the cluster with the index y_i. First, the centroids are randomly initialized at line 1. Lines 3-7 represent the assignment phase, furthermore lines 4-5 are based on Eq. 2.2. Finally, the update phase is carried out at line 8 by applying Eq. 2.1.

Algorithm 1: Standard K-Means Clustering

Input: k, $X\{x_1,x_2,...,x_n\}$
Output: $C\{c_1,c_2,...,c_k\}$, $Y\{y_1,y_2,...,y_n\}$

```
1 C{c1,c2,...,ck} ← random k data points from X{x1,x2,...,xn}
2 repeat
3 |   foreach xi in X{x1,x2,...,xn} do
4 |   |   D{d1,d2,...,dk} ← compute distances from xi to C{c1,c2,...,ck}
5 |   |   yi ← argmin (D{d1,d2,...,dk})
6 |   |   assign xi to C_yi
7 |   end
8 |   C{c1,c2,...,ck} ← data points average of each cluster (C1,C2,...,Ck)
9 until cluster membership convergence;
```

2.4.2 Logistic Regression

One of the most used supervised learning algorithms for classification is logistic regression. Although there are multiple variants of the logistic regression algorithm, this section presents logistic regression with gradient descent [46], as it is the variant used throughout this research project.

Logistic regression is a probabilistic classifier that models a binary output variable based on one or more variables called predictors [47]. The classifier is trained to make a binary decision regarding the class of a new given observation. Note that logistic regression is a discriminative model, i.e., it models the decision boundary between the classes by learning to distinguish and discriminate between them.

The classifier takes as input an observation x, which is represented as a vector of features [x_1,x_2,...,x_n], where n is the number of features (i.e., predictor variables). The output $\hat{y}$ of the classifier can either be 1, which means that the observation is predicted to belong to the class, or it can be 0, which means that the observation is not predicted to be a member of the class. During the training phase of the classifier with a training set, the classifier learns a vector of weights and a bias term. Note that the training set contains observations and the true labels of the observations, i.e., $y^{(i)}$ is the true class label of observation $x^{(i)}$.

The weights are real numbers that represent the importance of the features to the classification decision, i.e., the weight w_i represents the importance of the input feature x_i to the classification decision. The weighted sum of the inputs is calculated, and then, to the result is added the bias term, as shown in Eq. 2.3.

$$z = \left(\sum_{i=1}^{n} w_i x_i \right) + b = \mathbf{w} \cdot \mathbf{x} + b \tag{2.3}$$

As the weights $\mathbf{w}$ and the bias b represent the parameters of the model, the conventional machine learning notation θ is used to refer to the parameters, where $\theta = [b, \mathbf{w}] = [b, w_1, w_2, ..., w_n]$. To facilitate the discussion from this section, the term *weights* will alternatively be referring to the term θ, meaning that it also includes the bias term. Eq. 2.3 can be simplified into $z = \theta \cdot \mathbf{x}$, with the constraint that an observation x must be represented as a vector of the form [1, x_1,x_2,...,x_n].

The resulting number z represents the weighted sum of the evidence for the membership to the class, however, it is not a probability. As the weights are real numbers, the resulting number z may not have a value between 0 and 1. To obtain a probability, the value of z is passed through the sigmoid function from Eq. 2.4. The sigmoid function is also called the logistic function, and it ensures that a probability is obtained, i.e., a value between 0 and 1. If the obtained probability $\boldsymbol{\sigma}(z)$ is higher than 0.5, then the classification decision $\hat{y}$ is 1, otherwise $\hat{y}$ is 0.

$$\boldsymbol{\sigma}(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + exp\ (-z)} \tag{2.4}$$

The training phase aims to learn the weights that make the predicted label $\hat{y}$ of each observation x as close as possible to the true label y. This is realized by minimizing the distance between the predicted labels and the true labels. For this purpose, a loss function $L(\hat{y}, y)$ is used to measure the distance or dissimilarity between the predicted label and the true label. Logistic regression is based on the cross-entropy loss function, which represents the negative log-likelihood, as shown in Eq. 2.5. The cross-entropy loss function is convex, therefore it has at most one minimum, which is global.

$$L(\hat{y}, y) = -[y \log \boldsymbol{\sigma}(\mathbf{w} \cdot \mathbf{x} + b) + (1 - y) \log(1 - \boldsymbol{\sigma}(\mathbf{w} \cdot \mathbf{x} + b))] \tag{2.5}$$

An optimization algorithm is used during the training phase, to iteratively update the weights with the purpose of minimizing the loss function. The optimization algorithm that has been used in this research project is batch gradient descent, also known as standard gradient descent [48].

Gradient descent is guaranteed to find the global minimum of the cross-entropy loss functions, as there are no local minima [46]. At every epoch, gradient descent searches in the space of the parameters θ for the direction in which the loss function's slope rises the most steeply. The direction represents the gradient, which is a vector that points to the direction of the highest increase in the function. The aim is to move in the opposite direction specified by the gradient, and the magnitude of the movement is specified by a learning rate μ. The gradient is the derivative of the loss function from Eq. 2.5, as shown in Eq. 2.6.

$$\frac{\partial L(\hat{y}, y)}{\partial w_j} = (\hat{y} - y)\, x_j \tag{2.6}$$

Algorithm 2 represents the training phase of the logistic regression algorithm with (batch) gradient descent. The algorithm takes as input the observations $\mathbf{x}$, the true labels $\mathbf{y}$, the learning rate μ, and the number of epochs e. Note that $\mathbf{x}$ is represented as a matrix, in which the rows represent the observations $x^{(i)}$. Additionally, $\mathbf{y}$ is represented as a vector, where $y^{(i)}$ is the true label of observation $x^{(i)}$. The algorithm outputs the optimal weights θ that minimize the loss function. At line 1, the weights are initialized to 0. With batch gradient descent, the gradient is computed for the entire training set at each epoch. Note that with Stochastic Gradient Descent (SGD), the gradient is computed only for a single randomly chosen observation from the training set. The predicted labels are computed at line 3. Then, at line 4, the gradient is computed with the transposed of matrix $\mathbf{x}$ by following Eq. 2.6. Finally, the weights are updated with the negative value of the gradient that is weighted by the learning rate. Note that for reporting purposes, one could optionally compute the loss at each epoch [46].

Algorithm 2: Logistic Regression with (Batch) Gradient Descent

Input: e, μ, $\mathbf{x}$, $\mathbf{y}$
Output: θ

1 $\theta \leftarrow 0$
2 **for** $i \leftarrow 0;\ i < e;\ i \leftarrow i + 1$ **do**
3 $\quad \hat{\mathbf{y}} \leftarrow \boldsymbol{\sigma}(\theta \cdot \mathbf{x})$
4 $\quad g \leftarrow (\hat{\mathbf{y}} - \mathbf{y}) \cdot \mathbf{x}^{\mathrm{T}}$
5 $\quad \theta \leftarrow \theta - \mu g$
6 **end**

2.5 Distributed Machine Learning

The rapid technological innovations in recent years have brought massive growth in data collection. Training machine learning models can take an extremely long amount of time, as the data sets can be in the order of terabytes. A serial solution, i.e., a centralized solution of machine learning algorithms would not be feasible in the case that a data set is too large to be stored on a single machine. Thus, machine learning models are often trained on distributed systems to increase the parallelization and total amount of I/O bandwidth. One must enable parallel computation, data distribution and resilience to failures for the implementation of distributed machine learning algorithms.

Machine learning workloads can be accelerated by applying two different complementary methods: vertical scaling or scaling up (i.e., adding more resources to a single machine) and horizontal scaling or scaling out (i.e., adding more machine nodes to the distributed system) [49]. For vertical scaling solutions, the most common method is to leverage programmable GPUs, as many research works have demonstrated the benefits of leveraging GPUs [50, 51, 52, 53]. Machine learning algorithms can be scaled up even by using general-purpose CPUs, as they have increased the availability and width of vector instructions in the recent generations of products [54]. However, there are several reasons to prefer scaling out rather than scaling up. First, the cost of horizontally scaling a solution is often lower than vertical scaling. The second reason is that a distributed system with multiple nodes increases resilience against failures, as the system can continue operating when a single processor fails. Lastly, having several machines increases the aggregate I/O bandwidth compared to a single machine.

There are two fundamental methods for partitioning a machine learning algorithm across several machines: data parallelism and model parallelism. Note that the two

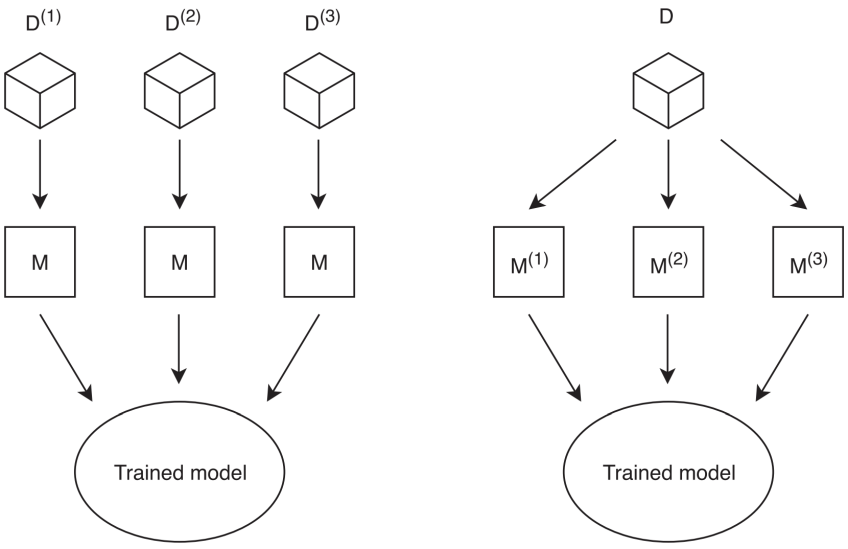

Figure 2.4: Parallelism methods in distributed machine learning [54]

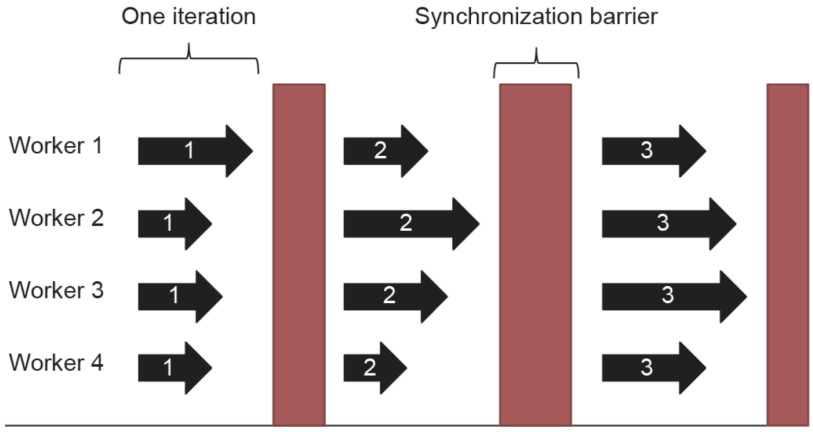

Figure 2.5: Bulk synchronous parallel (BSP) [49]

methods are complementary [49] and both methods can be simultaneously applied [54]. In the Data-Parallel method, the data set is divided into a number of partitions equal to the number of worker nodes or machines in the system, and then the partitions are distributed to the workers. All workers apply the same algorithm to their assigned partition, and they have access to the same model [54]. In the Model-Parallel method, the model is partitioned into several partial models [49] which are distributed to the worker nodes. Each worker node processes the entire data set and updates his assigned partial model. The resulting model is finally obtained by aggregating all model parts. The two methods are outlined in Figure 2.4.

There are several synchronization protocols for bridging the computation and communication between nodes. These protocols will be presented in the following.

Bulk Synchronous Parallel (BSP) is the simplest and strictest synchronization protocol. Algorithms that follow BSP alternate between a computation phase and a communication phase or synchronization barrier [49]. The results of a computation phase are visible to worker nodes only after the next synchronization barrier has been completed. Figure 2.5 shows that each worker node executes one iteration of computation and then waits for all other nodes to complete their iteration, and exchange information about model parameters before continuing the next iteration. The advantage of algorithms based on BSP is that a correct solution is guaranteed. The drawback of this protocol is that it induces overheads when certain workers are progressing slower than others [54].

Stale Synchronous Parallel (SSP) relaxes the synchronization overhead from BSP, by allowing the faster workers to continue their execution for a certain number of iterations that is defined via a staleness threshold [49]. Once a worker has passed the staleness threshold it is stopped by SSP. A worker operates over cached data and commits changes to the data at the end of a task cycle, and this can cause other workers to operate over stale data [54]. SSP ensures strong convergence guarantees, however, when the staleness becomes too high, the convergence rates decline [54].

Approximate Synchronous Parallel (ASP) specifies the limit of how inaccurate a parameter can be. Whenever an insignificant aggregated update is carried out, the synchronization can be delayed indefinitely by the server. However, it is not straightforward which parameter to choose for defining which updates are insignificant [54].

Barrierless Asynchronous Parallel (BAP), also called Total Asynchronous Parallel (TAP), tolerates workers to communicate in parallel without waiting for each other. Therefore, BAP is the fastest synchronization protocol, as synchronization overheads are avoided. A drawback of BAP is that the model can develop incorrectly and can converge slowly [54].

Chapter 3

Porting Stateful Machine Learning Algorithms to Serverless

This chapter presents a technique for porting stateful machine learning algorithms to serverless, and the challenges and limitations that are encountered during this process. Furthermore, optimization techniques are proposed for improving the performance of the stateful machine learning algorithms running on serverless architectures.

Lithops has been used for porting k-means and logistic regression to serverless. The serverless implementation of k-means is presented in Section 3.1, whereas the serverless implementation of logistic regression is presented in Section 3.2. Note that the serverful implementations based on the Python Multiprocessing API are not presented, as they are essentially identical to the serverless implementations, with the difference that they are based on processes rather than serverless functions.

In this research project, the serverless functions are running via AWS Lambda, and the data sets are stored in Amazon S3. The implemented stateful machine learning algorithms follow the BSP synchronization protocol, as it ensures the correctness of the results, however, the protocol has the limitation that it brings synchronization overheads that slow down the execution of the algorithms. The execution design of the algorithms is illustrated in Figure 3.1. In the proposed implementation, the inter-function parallelism pattern is adopted for employing data parallelism on stateful machine learning algorithms across a set of concurrent serverless functions called workers. A client machine executes client code that launches several serverless functions that share mutable state through disaggregated memory. Each worker operates on a partition of the data set and at each iteration exchanges partial results with the other workers via the shared state, i.e., the Redis node.

When porting stateful algorithms to serverless, the first challenge is to identify what data can be computed independently and concurrently by the workers. The workers perform parallel computations regarding data of interest (e.g., centroids) and then the partial results obtained by all workers are aggregated to attain global results. As the stateful algorithms are in this case iterative machine learning algorithms, these steps are repeated at each iteration of the algorithms. The partial results computed by the workers must be stored in the shared state, as one of the workers must fetch the partial results of all other workers, and then aggregate them. Furthermore, the global results must be stored in the shared state, for the reason that they must be accessed by all workers at each iteration. In the case of the k-means algorithm, the partial results are represented by the cluster totals and cluster counters, whereas the global results are the centroids that are computed by dividing the cluster totals by the cluster counters. Moreover, in the case of the logistic regression algorithm, the partial results are represented by the partial gradients (i.e., sub-gradients), whereas the global

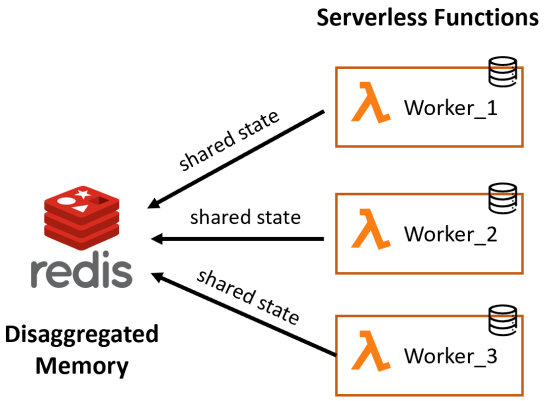

Figure 3.1: Inter-function parallelism algorithm design based on workers

results are represented by the weights and the global gradients.

In the implementation of the algorithms, the challenge is to access the Redis node as little as possible to minimize communication overheads. Therefore, the workers should operate over data stored in local memory as much as viable and then access and update the shared state only when necessary. Furthermore, bulk operations for accessing and updating the shared state should always be preferred, as they minimize the number of communication requests to the Redis node. At each iteration, the workers fetch the global shared state from the Redis node, then run a computation phase that yields partial results that are stored in the shared state before continuing to the next iteration. At the end of every iteration, one of the workers fetches the partial results from the global shared state, and then performs an aggregation of the partial results of all workers, and finally stores the aggregated results in the shared state. The limitation of this approach is that fetching the partial results of all workers, aggregating them, and updating the global shared state with the aggregated results is time-consuming and slows down the execution of the algorithms. In the proposed implementation, the worker who performs the described aggregation is always the first worker. These steps ensure that at the beginning of every iteration, all workers fetch from the shared state the aggregated results of the last iteration.

Two versions have been implemented for both algorithms, where each version uses a different design for storing the shared state. In the first version of the implemented algorithms, there is one single copy for each data object being stored globally in the shared state. Therefore, locks are used to prevent concurrent serverless functions from simultaneously accessing the same data object. Locks have the limitation of carrying overheads and slowing down the execution of the algorithms, therefore the second version applies an optimization technique that aims to mitigate overheads by proposing a lock-free design. In contrast to the first version, the second version stores in the shared state one copy of each data object for every worker. Therefore, whenever one worker must update the shared state with his partial results, the worker will access the data objects that are associated with him. By following this approach, it can be said that every worker has his own memory space in the global shared state, as presented in Figure 3.2. Consequently, no access conflicts can occur when the workers update the shared state with their partial results. Furthermore, at the end of every iteration, one of the workers (i.e., the first worker) fetches all data objects of all workers from the

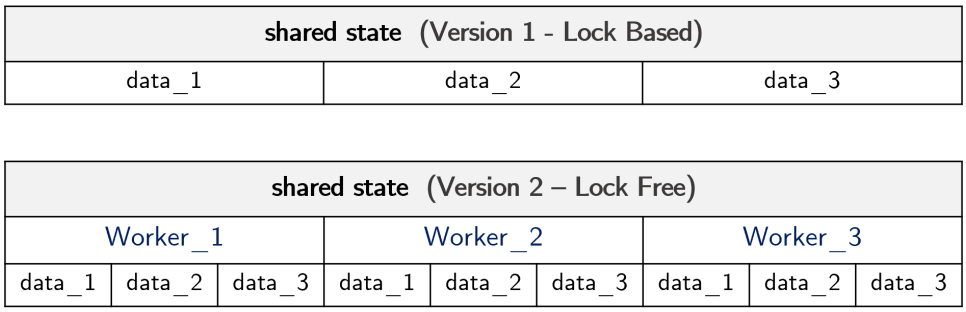

Figure 3.2: Shared State Design

shared state, and aggregates them, and then stores the aggregated results in the shared state before continuing to the next iteration. However, the limitation of the second version of the algorithms is that the aggregation phase lasts longer, considering that more data objects from the shared state must be aggregated, as there is one copy of each data object for every worker.

An optimization technique for improving the performance of the algorithms is to additionally employ the intra-function parallelism pattern in conjunction with the inter-function parallelism pattern. This modified execution design of the algorithms is illustrated in Figure 3.3. Similarly to the previous case, each worker operates on a partition of the data set and at each iteration exchanges partial results with the other workers via the shared state. At each iteration, the workers fetch the global shared state from the Redis node, and the computation results are stored in the shared state before continuing to the next iteration. However, the computation phase is not executed by the workers, but by the inner workers. Each worker launches at every iteration a number of inner workers, where the number is dictated by the number of vCPU of the serverless function. Each worker splits his data set partition into smaller partition chunks that are sent to the inner workers. The inner workers operate over the smaller partition chunks and run the computation phase of the iteration in parallel. When the inner workers have finished running the computation, they send the results back to the (parent) workers, and then each (parent) worker aggregates the results received from the inner workers. The disadvantage of this approach is that sending the results from the inner workers to the (parent) workers and aggregating the received results is time-consuming, as these operations take place at every iteration. However, these overheads are expected to be compensated by the fact that the inner workers execute the computation phase in parallel, which should significantly speed up the execution of the algorithms. The inner workers are implemented as processes that are launched and terminated at every iteration, therefore this brings additional overheads for each iteration. Each worker adopts fork-join parallelism, as it forks its sequential control flow into multiple parallel branches of execution, i.e., the inner worker processes, and join when the inner workers have finished running the computation phase, and then the sequential execution of the worker is continued.

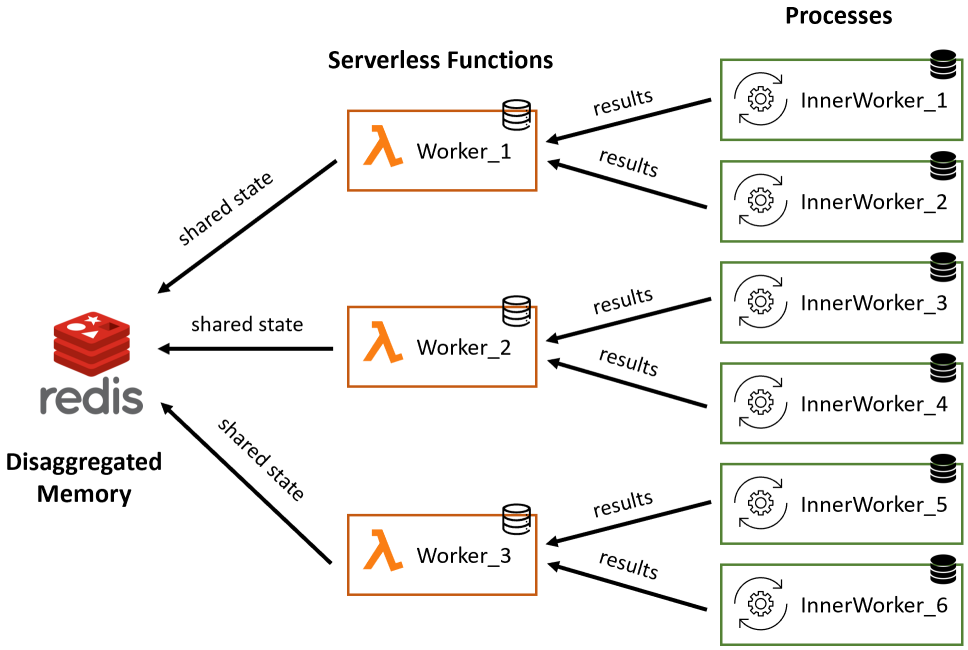

Figure 3.3: Inter-function and intra-function parallelism algorithm design based on workers and inner workers

3.1 Serverless K-Means Clustering

In this section, the two serverless implementation versions of the k-means clustering algorithm are detailed. Both versions follow the same logic, however, the first version requires locks for updating the shared state, whereas the second version proposes an optimization by using a lock-free design. Listing 3.1 contains the client code that is executed on the client machine for invoking concurrent serverless functions (via the `map` method of *FunctionExecutor*) that run the k-means algorithm, where each function reads its assigned data set partition from disaggregated object storage (in the method `launch_worker`).

Worker ID	Data Point ID	Data Point Value	Cluster ID
1	1	[1.1 ; 1.5]	2
	2	[2.4 ; 3.8]	3
2	3	[0.2 ; 2]	2
	4	[3.6 ; 4]	2

clusters_counters		
0	3	1

clusters_totals		
0	[4.9 ; 7.5]	[2.4 ; 3.8]

clusters_centers		
0	[1.63 ; 2.5]	[2.4 ; 3.8]

Figure 3.4: Lock-Based K-Means - detailed shared state variables (clusters $k = 3$)

Lock-Free K-Means – shared state (clusters $k = 3$)			
Worker_1		Worker_2	
clusters_counters	clusters_totals	clusters_counters	clusters_totals
[0 ; 0 ; 0]	[0 ; 0 ; 0]	[0 ; 0 ; 0]	[0 ; 0 ; 0]

Figure 3.5: Lock-Free K-Means - initial shared state for each worker

```
1  def launch_worker(obj):
2    worker_id = obj.part - 1
3    dataset = get_dataset(obj.data_stream.read().decode("utf-8"))
4    workers[worker_id].set_dataset(dataset)
5    return workers[worker_id].run()
6
7  def fit(X):
8    ...
9    initialize_workers()
10   fexec = FunctionExecutor()
11   fexec.map(launch_worker, X, obj_chunk_number=n_workers)
12   workers_results = fexec.get_result()
```

Listing 3.1: K-Means - client code for invoking serverless functions

The k-means clustering algorithm computes k centroids at each iteration. The centroids are the principal pillar of the algorithm, as the assignment of the data points to clusters depends entirely on the centroid values. Listing 3.2 defines all the variables that represent the shared state in the first version of the implemented algorithm, i.e., the lock-based implementation. The shared state is primarily represented by data related to the centroids. The centroid of a cluster represents the average of all data points that are assigned to the cluster. To calculate the average, one would need the sum of all data points assigned to the cluster, and the number of data points belonging to the cluster. As such, the core data objects in the shared state are the sum of all data points for each cluster, and the number of data points from each cluster, as both data objects can be computed independently and concurrently by the workers. Each of the two data objects is represented as an array, in which the index i is associated with the cluster i, as illustrated in Figure 3.4. The array containing the cluster totals (i.e., the sum of all data points for each cluster) has been defined at line 1 in Listing 3.2, and the array containing the cluster counters (i.e., the number of data points from each cluster) has been defined at line 2 in Listing 3.2. The workers iterate over their data set partition, and for each data point, they update the values of the arrays from the shared state, depending on which cluster the data point is assigned. In the lock-free implementation of the algorithm, the difference in the shared state is that the variables `clusters_totals` and `clusters_counters` are defined as a list of arrays as in Listing 3.3, where each item of a list represents the memory space of one worker, as illustrated in Figure 3.5.

```
clusters_totals = Array('d', [0 for _ in range(n_clusters)])
clusters_counters = Array('i', [0 for _ in range(n_clusters)])
locks_clusters = [Lock() for _ in range(n_clusters)]
barrier = Barrier(n_workers)
has_converged = RawValue('i', 0)

if centroids_initialized == False:
  clusters_centers = Array('d', [0 for _ in range(n_clusters)])
else:
  clusters_centers = Array('d', init)
```

Listing 3.2: Lock-Based K-Means - shared state variables

```
clusters_totals =
  [Array('d', [0 for _ in range(n_clusters)]) for idx in range(n_workers)]

clusters_counters =
  [Array('i', [0 for _ in range(n_clusters)]) for idx in range(n_workers)]
```

Listing 3.3: Lock-Free K-Means - shared state variables

Figure 3.4 presents how the principal data objects from the shared state are computed, where the table from the left represents the data set, and each table from the right represents a shared state data object (i.e., a shared state variable). In the example from Figure 3.4, the data set is bi-dimensional, and it is split into two partitions, one for each worker. Each worker iterates the data points, assigns them to clusters and updates the shared state variables accordingly. To exemplify, the first worker assigns the first data point to the second cluster, therefore it increments by 1 the value of the second index of the `clusters_counters` array, and additionally, the first data point is summed to the value of the second index of the `clusters_totals` array. Three data points from the data set are assigned to the second cluster, therefore the value at index 2 of `clusters_counters` will become 3 by the end of the iteration. Furthermore, the value at index 2 of `clusters_totals` will be the sum of all data points assigned to the second cluster by the end of the iteration. One data point has been assigned to the third cluster, therefore the value at index 3 of `clusters_counters` will become 1 by the end of the iteration. As only one data point has been assigned to the third cluster, the value at index 3 of `clusters_totals` will become the value of the only data point assigned to the third cluster. When both workers have finished assigning the data points to clusters, and have finished updating the shared state variables called `clusters_counters` and `clusters_totals`, one of the workers (i.e., the first worker) will compute the values of the centroids. The first worker will fetch the final values of the iteration of `clusters_counters` and `clusters_totals`, and then compute the centroids by dividing the totals by the counters. Finally, the first worker will update the shared state variable `clusters_centers` by the computed

centroids. At the beginning of the next iteration, all workers will retrieve the current centroids by fetching the value of `clusters_centers`.

The synchronization of the workers is realized at each iteration via a barrier object that is stored in the shared state, which has been defined at line 4 in Listing 3.2. In the lock-based version of the algorithm, locks are required for synchronization when the workers update the values of the arrays containing the cluster totals and cluster counters. The limitation of the described implementation is that using barrier and lock objects brings additional overheads that slow down the execution of the algorithm. Considering the example from Figure 3.4, the two workers should not be able to write simultaneously at the same index of the arrays `clusters_totals` and `clusters_counters`. Therefore, locks have been used to synchronize access to a specific index of the two arrays. One lock has been used for every index (i.e., for every cluster), as an entire array should not be locked at a certain point by a worker, but rather only the access to the value of a specific cluster index should be locked by a worker. The locks have been defined at line 3 in Listing 3.2.

```
1 if worker_id == 0 and centroids_initialized == False:
2   cluster_centers_idx = np.random.choice(len(X), n_clusters, replace=False)
3   cluster_centers[:] = [X[idx] for idx in cluster_centers_idx]
```

Listing 3.4: K-Means - centroids initialization

The centroids are stored in the shared state, and they are accessed by all workers at the beginning of each iteration, and they are updated by the first worker at the end of every iteration. The centroids have been defined at lines 7-10 in Listing 3.2. In the standard k-means clustering algorithm, the centroids are initially chosen by selecting k random data points from the data set. As the data set is only accessed by the workers in parallel by reading their assigned partitions from object storage, selecting k random data points from the entire data set would unnecessarily complicate the implementation and slow down the execution of the algorithm. To facilitate the implementation and speed up the execution of the algorithm, it has been decided to simply select k random data points from the partition of one of the workers (i.e., the first worker), as presented in Listing 3.4. Additionally, one can predefine a set of initial centroids in a file on disk. The predefined initial centroids must be stored in a file because one may have hundreds of clusters, therefore the centroids must be stored on disk, as it is not feasible to specify hundreds of centroids as command line arguments when running the program containing the algorithm. Having the option of predefining a set of centroids is important for evaluation purposes. When comparing performance aspects such as the execution time of two different versions of the k-means algorithm, one must initialize the algorithms with the same centroids, as different values of initial centroids can yield different results and convergence may be achieved in a different number of iterations.

There is one variable in the shared state which represents the state of convergence of the algorithm, which has been defined at line 5 in Listing 3.2. This variable is not mandatory for the implementation of the algorithm, however, it was used to have only one of the workers (i.e., the first worker) checking for convergence, rather than

all workers. The first worker will verify convergence by comparing the centroids of the last iteration with the centroids of the current iteration, and if they are equal, then the first worker will set the value of the convergence variable to 1. The advantage of this approach is that the other workers will not verify the convergence criteria themselves, but rather they will only verify the value of the convergence variable from the shared state.

```
1  clusters_counters_local = [0 for _ in range(n_clusters)]
2  clusters_totals_local = [0 for _ in range(n_clusters)]
3  clusters_centers_local = clusters_centers[:]
4
5  for x_idx, x_value in enumerate(X):
6    cluster_idx = argmin(computeDistances(clusters_centers_local, x_value))
7    labels[x_idx] = cluster_idx
8    clusters_counters_local[cluster_idx] += 1
9    clusters_totals_local[cluster_idx] += x_value
10
11 for cluster_idx in range(n_clusters):
12   with locks_clusters[cluster_idx]:
13     clusters_counters[cluster_idx] += clusters_counters_local[cluster_idx]
14     clusters_totals[cluster_idx] += clusters_totals_local[cluster_idx]
15
16 barrier.wait()
17
18 if worker_id == 0:
19   cluster_centers_new = computeCentroids(clusters_totals, clusters_counters)
20   clusters_centers[:] = cluster_centers_new
21   clusters_counters[:] = 0
22   clusters_totals[:] = 0
23   if areEqual(cluster_centers_new, clusters_centers_local):
24     has_converged.value = 1
25
26 barrier.wait()
27
28 if has_converged.value == 1:
29   break
```

Listing 3.5: Lock-Based K-Means - worker code of an iteration

Listing 3.5 contains the code for an iteration of the lock-based k-means algorithm that is executed by the workers. The variables `clusters_counters_local` and `clusters_totals_local` represent the local version of the shared state variables `clusters_counters` and `clusters_totals` from Listing 3.2. To minimize communication with the Redis node, changes are made over the local version of shared state variables, and when all required changes have been made locally, the global shared state variables are updated with the values of the local version. At line 3 from Listing 3.5, the cluster centers (i.e., the centroids) are fetched from the shared state. At lines 5-9,

the worker iterates its data set partition (i.e., the `X` variable), and for each data point, it computes the distances between the data point and the centroids to find the label (i.e., the cluster index to which the data point is assigned), and then updates the arrays storing the totals and counters of the clusters. At lines 11-14, the shared state variables are updated using the final values for totals and counters that have been obtained in the previous step by the worker. At line 12, a lock is used to synchronize the access to a specific index of the totals and counters arrays. At line 16, a barrier waits for all workers to have finished updating the shared state. When the barrier passes, at lines 18-19, the first worker computes the new centroids using the final values of totals and counters aggregated from all workers. At lines 20-22, the first worker updates the global centroids with the computed centroids, and resets the global values of the cluster counters and totals to 0 for the next iteration. At lines 23-24, the first worker checks for convergence, and if convergence has been met, the first worker updates the `has_converged` variable from the shared state. A barrier is used at line 26, to wait for the first worker to update the global centroids before continuing the execution of the other workers. Finally, at lines 28-29, all workers check at the end of each iteration whether convergence has occurred, and if it has, the workers break the execution of the iteration phase.

Listing 3.6 contains the primary implementation difference between the lock-based and the lock-free implementation of the algorithm. The code from Listing 3.6 represents the lock-free equivalent of the code at lines 11-14 from Listing 3.5. The variables `clusters_counters` and `clusters_totals` are lists, in which each item is an array. Each worker will only access the lists at the index that is associated with his identifier (i.e., the `worker_id` variable). Therefore, no locks are required, as no access conflicts can occur.

```
1 for idx in range(n_clusters):
2   clusters_counters[worker_id][idx] = clusters_counters_local[idx]
3   clusters_totals[worker_id][idx] = clusters_totals_local[idx]
```

Listing 3.6: Lock-Free K-Means - updating the shared state

```
1  def innerWorker_computeClusters(connection, X, cluster_centers):
2    labels = []
3    clusters_counters = [0 for _ in range(len(cluster_centers))]
4    clusters_totals = [0 for _ in range(len(cluster_centers))]
5    for x in X:
6      cluster_idx = argmin(computeDistances(cluster_centers, x))
7      labels.append(cluster_idx)
8      clusters_counters[cluster_idx] += 1
9      clusters_totals[cluster_idx] += x
10   connection.send([labels, clusters_counters, clusters_totals])
11   connection.close()
```

Listing 3.7: K-Means - inner worker code

```
1  partitioned_X = partition_dataset(X, n_processes)
2  ...
3  for pid in range(n_processes):
4    parent_conn, child_conn = mp.Pipe()
5    parent_connections.append(parent_conn)
6    child_connections.append(child_conn)
7  ...
8  for pid in range(n_processes):
9    process = mp.Process(
10     target=innerWorker_computeClusters,
11     args=(child_connections[pid], partitioned_X[pid], cluster_centers_local,))
12   process.start()
13
14 for parent_connection in parent_connections:
15   results = parent_connection.recv()
16   labels.extend(results[0])
17   for cluster_idx in range(n_clusters):
18     clusters_counters_local[cluster_idx] += results[1][cluster_idx]
19     clusters_totals_local[cluster_idx] += results[2][cluster_idx]
```

Listing 3.8: K-Means - invoking inner workers and aggregating their results

As an optimization technique, the intra-function parallelism pattern will be employed to parallelize the execution of the serverless functions via inner workers. Note that for parallelizing the execution of AWS Lambda functions based on Python code, one can only use processes and pipes from the Python Multiprocessing API. At each iteration, a number of processes (i.e., inner workers) will be launched to parallelize the sequential execution of the serverless function (i.e., worker), where the number is dictated by the number of vCPU of the serverless function. Each inner worker will run the computation phase of the algorithm, i.e., computing the cluster totals, counters and labels, as in Listing 3.7. The code for invoking inner workers and aggregating their results is presented in Listing 3.8. At line 1 in Listing 3.8, the partition assigned to the worker is split into several smaller chunks (i.e., sub-partitions), one for each inner worker. At lines 3-6, a pipe is created for each inner worker, to enable the communication between the inner workers (i.e., child processes) and the parent process of the worker. Each inner worker will run the function from Listing 3.7 called `innerWorker_computeClusters`, which takes as parameters the child process connection object, the chunk of the partition (i.e., a sub-partition) and the centroids. The inner workers perform the computation and when they have finished, the results are received and aggregated at lines 14-19. At this point, the worker has obtained the iteration's final local version of the cluster counters and totals, which will have to be used for updating the global shared state, as in lines 11-14 from Listing 3.5 for the lock-based implementation, or as in Listing 3.6 for the lock-free implementation. The limitation of this optimization technique is that additional overheads result from splitting partitions into sub-partitions, creating pipes for enabling communication with the inner workers, launching and terminating inner workers at every iteration, sending

the output (i.e., the cluster totals, counters and labels) of the inner workers to the (parent) workers and aggregating the received results. However, these overheads are expected to be compensated by the speed-up resulting from the parallel execution of the algorithm via the inner workers.

The output of the k-means algorithm is the centroids and the labels. The centroids are stored in the shared state, i.e., remotely in the Redis node, therefore the serverless functions do not need to output the centroids, as they can be accessed by the client machine directly via the shared state. However, the labels (i.e., the cluster index to which each data point is assigned) represent data that is local to the serverless functions, and it must be returned to the client machine that launched the serverless functions. Therefore, the output of each serverless function is the labels of its assigned partition of the data set. Each serverless function outputs a set of partial labels, which must be concatenated to obtain the complete list of labels for the entire data set. A limitation of the serverless implementations is that fetching the centroids from the shared state, retrieving the partial labels outputted by the serverless functions and aggregating the partial labels yield additional overheads that slow down the execution of the algorithm. Listing 3.9 contains the client code that is executed on the client machine that invokes a number of `n_workers` serverless functions at line 2, receives the partial labels of each serverless function as a result at line 3, and then the complete labels are obtained via concatenation at line 4.

```
1 fexec = FunctionExecutor()
2 fexec.map(launch_worker, X, obj_chunk_number=n_workers)
3 partial_labels = fexec.get_result()
4 labels = np.concatenate(partial_labels, axis=0)
```

Listing 3.9: K-Means - concatenating the partial labels of all workers

Algorithm 3 contains the pseudocode of a generalized serverless implementation of the k-means algorithm. The algorithm takes as input X which represents the data set partition assigned to the worker, *worker_id* which represents the identifier of the worker, *max_iter* which represents the maximum number of iterations, *centroids_initialized* which states whether the centroids have already been initialized, and *cpu_count* which specifies the level of parallelism adopted by the worker (i.e., either serial or parallel via inner workers). The output of the algorithm is the global centroids and the labels. The shared state is represented by all variables prefixed by "`global`", whereas the variables prefixed by "`local`" represent the local version of the variables from the shared state. Initially, the values of the cluster counters and totals from the shared state are initialized to 0. At lines 1-4, if the centroids have not been predefined, then the first worker will randomly initialize the centroids, whereas the other workers wait for the centroids to be initialized. The iteration phase begins at line 5, and at line 6, the global centroids are fetched from the shared state and stored in a local variable. At lines 7-11, the labels and cluster counters and totals are computed serially, whereas at lines 12-18, the labels and cluster counters and totals are computed in parallel by the inner workers, and then aggregated. At lines 19-20, the global shared state variables for counters and totals are aggregated with the partial values computed by each worker. At line 21, all workers synchronize, and by this point, the final values

for the current iteration of the counters and totals have been obtained. At lines 22-23, the first worker computes the new centroids by dividing the final totals and counters from the shared state, and then updates the global centroids with the new centroids. At lines 24-25, the first worker resets the values of the cluster counters and totals from the shared state to 0 for the next iteration. The workers must once again synchronize at line 26, to ensure that they are operating with the new computed centroids. Finally, at the end of every iteration, at lines 27-28, all workers check whether the algorithm has converged and break the execution of the algorithm if necessary.

Algorithm 3: Serverless K-Means Clustering

```
   Input: X, worker_id, max_iter, centroids_initialized, cpu_count
   Output: global_clusters_centers, labels
 1 if centroids_initialized == False then
 2 |   if worker_id == 0 then
 3 |   |   global_clusters_centers ← getRandomCentroids(X)
 4 |   global_barrier.wait()
 5 for iter ← 0; iter < max_iter; iter ← iter + 1 do
 6 |   local_clusters_centers ← global_clusters_centers
 7 |   if cpu_count == 1 then
 8 |   |   clusters_data ← computeClusters(X, local_clusters_centers)
 9 |   |   labels ← clusters_data.getLabels()
10 |   |   local_clusters_counters ← clusters_data.getClustersCounters()
11 |   |   local_clusters_totals ← clusters_data.getClustersTotals()
12 |   else if cpu_count > 1 then
13 |   |   partitions ← partitionDataset(X, cpu_count)
14 |   |   inner_workers ← createInnerWorkers(partitions, local_clusters_centers)
15 |   |   invoke(inner_workers)
16 |   |   labels ← inner_workers.getLabels()
17 |   |   local_clusters_counters ← inner_workers.getClustersCounters()
18 |   |   local_clusters_totals ← inner_workers.getClustersTotals()
19 |   global_clusters_counters ← global_clusters_counters + local_clusters_counters
20 |   global_clusters_totals ← global_clusters_totals + local_clusters_totals
21 |   global_barrier.wait()
22 |   if worker_id == 0 then
23 |   |   global_clusters_centers ← global_clusters_totals / global_clusters_counters
24 |   |   global_clusters_counters ←0
25 |   |   global_clusters_totals ←0
26 |   global_barrier.wait()
27 |   if global_clusters_centers == local_clusters_centers then
28 |   |   break
29 end
```

3.2 Serverless Logistic Regression

The two serverless implementation versions of the logistic regression algorithm based on (batch) gradient descent are described in this section. Note that the training phase (i.e., the fitting) of the algorithm has been implemented, and not the classifier's prediction procedure of a new given observation. As in the implementation of the k-means algorithm, the first version of the algorithm requires locks for updating the shared state, whereas the second version aims to be an optimization of the first version, by providing a lock-free design. The client code from Listing 3.1 can as well be used for invoking concurrent serverless functions (via the `map` method of *FunctionExecutor*) that run the logistic regression algorithm, where each function reads its assigned training set partition from disaggregated object storage (in the method `launch_worker`).

```
1 weights = Array('d', [0 for _ in range(n_features + 1)])
2 gradients = Array('d', [0 for _ in range(n_features + 1)])
3 barrier = Barrier(n_workers)
4 lock = Lock()
```

Listing 3.10: Lock-Based Logistic Regression - shared state variables

```
1 gradients = [Array('d',
2              [0 for _ in range(n_features + 1)]) for idx in range(n_workers)]
```

Listing 3.11: Lock-Free Logistic Regression - shared state variables

At each iteration of the logistic regression algorithm, a set of gradients are computed, that are used for updating the weights. The objective of the training phase of the algorithm is to learn a set of weights that will be used by the classifier to predict the class label of a new given observation. Listing 3.10 defines all the variables that represent the shared state in the first version of the implemented algorithm, i.e., the lock-based implementation. The shared state is primarily represented by the gradients and the weights. Both of these data objects are represented as an array, where the length of the array is dictated by the number of features of the training set. The weights array has been defined at line 1 in Listing 3.10, whereas the gradients array has been defined at line 2. In the lock-free implementation of the algorithm, the difference in the shared state is that the gradients variable has been defined as a list of arrays as in Listing 3.11, where each item of the list represents the memory space of one worker, as illustrated in Figure 3.2. The synchronization of the workers is realized at each iteration via the barrier object that has been defined at line 3 in Listing 3.10. At every iteration, each worker increments the global gradients from the shared state with the partial gradients (i.e., sub-gradients) computed by the worker. Two concurrent workers should not be able to increment the value of the global gradients simultaneously, as conflicts and incorrect results can occur. Therefore, a lock has been defined at line 4

in Listing 3.10 to synchronize the access to the global gradients in the lock-based implementation of the algorithm. The limitation of the described implementation is that using a barrier and a lock leads to additional overheads that slow down the execution of the algorithm. Before the end of every iteration, one of the workers (i.e., the first worker) fetches the final global gradients aggregated from all workers and computes the new values of the weights by using the gradients and a learning rate. The weights represent the output of the logistic regression algorithm, however, the weights do not need to be returned by the serverless functions to the client machine that launched the functions. The weights are stored in the shared state, i.e., remotely in the Redis node, therefore they can be fetched by the client machine directly. A limitation of the described implementation is that the necessity of fetching the weights from the shared state leads to additional communication overheads.

```
1  weights_local = weights[:]
2  z = np.dot(X, weights_local)
3  y_pred = sigmoid(z)
4  errors = y_pred - y
5  gradients_local = np.dot(X.T, errors)
6
7  with lock:
8    gradients[:] += gradients_local
9
10 barrier.wait()
11
12 if worker_id == 0:
13   gradients_total = np.array(gradients[:])
14   weights_local = weights_local - learning_rate * gradients_total
15   weights[:] = weights_local
16   gradients[:] = empty_gradients
17
18 barrier.wait()
```

Listing 3.12: Lock-Based Logistic Regression - worker code of an iteration

At every iteration, each worker computes the gradients associated with his assigned training set partition. Thus, they are considered partial gradients, as they are not computed over the entire training set. However, the global values of the gradients can be obtained by summing the partial gradients of all workers. Therefore, the serverless functions (i.e., the workers) compute the partial gradients in parallel, and each of them increments the global values of the gradients using the partial computed gradients. Listing 3.12 contains the code for an iteration of the lock-based logistic regression algorithm that is executed by the workers. The variables `weights_local` and `gradients_local` represent the local version of the shared state variables `weights` and `gradients` from Listing 3.10. To minimize the number of requests to the Redis node, changes are made to the local version of shared state variables, and when all required changes have been made locally, the global shared state variables are updated with the values of the local version. At the beginning of each iteration, at line 1 in

Listing 3.12, each worker fetches the global weights and stores them in a local variable. At lines 2-5, the partial gradients are computed by following the steps outlined in Algorithm 2. After computing the partial gradients, each worker increments the values of the global gradients by the partial gradients at lines 7-8. At line 7, a lock is used to synchronize the access to the global gradients. At line 10, a barrier waits for all workers to have finished updating the global gradients. When the barrier passes, at lines 12-14, the first worker computes the new weights using the final global values of the gradients aggregated from all workers, by taking into account the learning rate. Then, at lines 15-16, the first worker updates the global weights with the computed weights and then resets the values of the global gradients to 0 for the next iteration. Finally, a barrier is used at line 18, to wait for the first worker to update the global weights before continuing the execution of the other workers.

Listing 3.13 contains the primary difference between the lock-based and the lock-free implementation of the algorithm. The code from Listing 3.13 represents the lock-free equivalent of the code at lines 7-8 from Listing 3.12. The `gradients` variable is a list, in which each item is an array. Each worker will only access the list at the index that is associated with his identifier (i.e., the `worker_id` variable). Thus, locks are not required, as access conflicts cannot occur.

```
1 gradients[worker_id][:] = gradients_local
```

Listing 3.13: Lock-Free Logistic Regression - updating the shared state

```
1 def innerWorker_computeGradients(weights, connection, X, y):
2   z = np.dot(X, weights)
3   y_pred = sigmoid(z)
4   errors = y_pred - y
5   gradients = np.dot(X.T, errors)
6   connection.send(gradients)
7   connection.close()
```

Listing 3.14: Logistic Regression - inner worker code

Similar to the k-means implementation from Section 3.1, as an optimization technique, the intra-function parallelism pattern will be adopted to parallelize the execution of the serverless functions via inner workers. As a reminder, one can only use processes and pipes from the Python Multiprocessing API for parallelizing the execution of AWS Lambda functions based on Python code. At each iteration, a number of processes (i.e., inner workers) will be launched to parallelize the sequential execution of the serverless function, where the number is dictated by the number of vCPU of the serverless function. Each inner worker will run the computation phase of the algorithm, i.e., computing the partial gradients, as in Listing 3.14. The code for invoking inner workers and aggregating the resulting gradients is presented in Listing 3.15. At lines 1-2 from Listing 3.15, the partition (`X`, `y`) of the training set assigned to the worker is split into several smaller chunks (i.e., sub-partitions), one for each inner worker. At lines 4-7, a pipe is created for each inner worker, to enable the communication between the inner

workers (i.e., child processes) and the parent process of the worker. Each inner worker will run the function from Listing 3.14 called `innerWorker_computeGradients`, which takes as parameters the weights, the child process connection object, and the chunk of the partition (i.e., a sub-partition). The inner workers perform the computation and when they have finished, the results are received and aggregated at lines 17-19. At this point, the worker has obtained the iteration's final local version of the gradients, which will have to be used for incrementing the values of the global gradients, as in lines 7-8 from Listing 3.12 for the lock-based implementation, or as in Listing 3.13 for the lock-free implementation. The limitation of the described optimization technique is that additional overheads result from splitting partitions into sub-partitions, creating pipes for enabling communication with the inner workers, launching and terminating inner workers at every iteration, sending the output (i.e., the partial gradients) of the inner workers to the (parent) workers and aggregating the received results to obtain the global gradients. Nevertheless, these overheads are expected to be compensated by the speed-up resulting from the parallel computation of the partial gradients.

```
1  partitioned_X = partition_dataset(X, n_processes)
2  partitioned_y = partition_dataset(y, n_processes)
3  ...
4  for pid in range(n_processes):
5    parent_conn, child_conn = mp.Pipe()
6    parent_connections.append(parent_conn)
7    child_connections.append(child_conn)
8  ...
9  for pid in range(n_processes):
10   process = mp.Process(
11     target=innerWorker_computeGradients,
12     args=(weights_local, child_connections[pid],
13           partitioned_X[pid], partitioned_y[pid],))
14   process.start()
15
16 gradients_local = 0
17 for parent_connection in parent_connections:
18   innerWorker_gradients = parent_connection.recv()
19   gradients_local += innerWorker_gradients
```

Listing 3.15: Logistic Regression - invoking inner workers and aggregating their results

Algorithm 4 contains the pseudocode of a generalized serverless implementation of the logistic regression algorithm. The algorithm takes as input X which represents the observations of the training set partition assigned to the worker, y which represents the true class labels of the observations from the training set partition, $worker\_id$ which represents the identifier of the worker, max_iter which represents the maximum number of iterations, $learning\_rate$ which specifies the step size, and cpu_count which specifies the level of parallelism adopted by the worker (i.e., either serial or parallel via inner workers). The output of the algorithm is the global weights. The shared state is

represented by all variables prefixed by "`global`", whereas the variables prefixed by "`local`" represent the local version of the variables from the shared state. Initially, the values of the weights and gradients from the shared state are initialized to 0. The iteration phase begins at line 1, and at line 2, the global weights are fetched from the shared state and stored in a local variable. At lines 3-4, the partial gradients are computed serially, whereas at lines 5-9, the partial gradients are computed in parallel by the inner workers, and then aggregated. At line 10, the global gradients from the shared state are incremented by the partial gradients computed by each worker. At line 11, all workers synchronize, and by this point, the final values for the current iteration of the gradients have been obtained. At lines 12-13, the first worker computes the new weights using the final global values of the gradients aggregated from all workers and the learning rate, and then updates the global weights with the computed weights. At line 14, the first worker resets the values of the global gradients to 0 for the next iteration. Finally, the barrier from line 15 waits for the first worker to update the global weights before continuing the execution of the other workers.

Algorithm 4: Serverless Logistic Regression

Input: X, y, $worker\_id$, max_iter, $learning\_rate$, cpu_count
Output: $global\_weights$

```
1  for iter ← 0; iter < max_iter; iter ← iter + 1 do
2      local_weights ← global_weights
3      if cpu_count == 1 then
4          local_gradients ← computeGradients(X, y, local_weights)
5      else if cpu_count > 1 then
6          partitions ← partitionDataset(X, y, cpu_count)
7          inner_workers ← createInnerWorkers(partitions, local_weights)
8          invoke(inner_workers)
9          local_gradients ← inner_workers.getGradients()
10     global_gradients ← global_gradients + local_gradients
11     global_barrier.wait()
12     if worker_id == 0 then
13         global_weights ← local_weights − learning_rate · global_gradients
14         global_gradients ← 0
15     global_barrier.wait()
16 end
```

Chapter 4

Correctness Validation of Serverless Implementations

This chapter presents the method in which the correctness of the serverless implementations of the stateful machine learning algorithms has been validated. The aim is to validate that the serverful machine learning algorithms have been correctly ported to serverless. The *scikit-learn* machine learning library has been used as a baseline for comparison.

Scikit-learn is a Python module that comprises a wide range of state-of-the-art machine learning algorithms and it provides an easy-to-use interface that fulfills the needs of non-specialists in statistical data analysis [55]. It features numerous machine learning algorithms for classification, regression and clustering, including k-means, k-nearest neighbors, support vector machines, decision trees, random forests, gradient boosting, principal component analysis, ensemble methods and neural network models. The implementation of *scikit-learn* is largely based on Python and uses *NumPy*[1] extensively for high-performance linear algebra and array operations. Note that *NumPy* is a fundamental Python library for scientific computing that specializes in fast and efficient numerical processing of multi-dimensional arrays [56].

Validation scripts have been used for verifying the correctness of the serverless implementations. To validate the correctness of the serverless implementations of the stateful machine learning algorithms, the results obtained by the serverless implementations have been compared with the results obtained using *scikit-learn*. The results obtained using the serverless implementations and using *scikit-learn* should be identical when using the same configuration of parameters. Note that the correctness of the serverful implementations have been validated in the same way as the serverless implementations, as both implementations follow the same logic, with the difference that the serverful implementations are based on processes rather than serverless functions.

The validation script of the k-means serverless implementation validates that the centroids and labels outputted by the serverless implementation are identical to the centroids and labels outputted by the *scikit-learn* library. The validation of the k-means serverless implementation has been carried out successfully, by using data sets of various sizes, with various number of clusters, various number of workers (i.e., concurrent serverless functions), and various number of inner workers. Specifically, the validation has been carried out using data sets with sizes of [10, 50, 100, 500] MB, with a number of [3, 5, 7, 10] clusters, with a number of [1, 2, 10, 20, 50] workers, and with a number of [2, 3, 4, 5, 6] inner workers.

Listing 4.1 contains an excerpt from the validation script that verifies the correctness of the serverless implementation of the k-means algorithm. The data set

[1] `https://numpy.org`

X_serial is the same as X_serverless, however, X_serial is stored locally on the client machine, whereas X_serverless is stored in disaggregated object storage (i.e., Amazon S3). The k-means algorithm from *scikit-learn* is executed locally on the client machine at lines 7-9, by using the same parameters that are used for the execution of the serverless implementation at lines 11-13. At lines 15-19, the correctness of the serverless implementation is validated, as the results of both algorithms are compared, i.e., the labels and the centroids.

```
1  import argparse
2  import numpy as np
3  from sklearn.cluster import KMeans as KMeansSerial
4  from kmeans_serverless import KMeansServerless
5
6  ...
7  kmeans_serial = KMeansSerial(
8         n_clusters=args.k, init=centroids)
9  kmeans_serial.fit(X_serial)
10
11 kmeans_serverless = KMeansServerless(
12        n_clusters=args.k, init=centroids, n_workers=args.n_workers)
13 kmeans_serverless.fit(X_serverless)
14
15 labels_equal = np.allclose(
16        kmeans_serial.labels_, kmeans_serverless.labels_)
17
18 centroids_equal = np.allclose(
19        kmeans_serial.cluster_centers_, kmeans_serverless.cluster_centers_)
```

Listing 4.1: Serverless K-Means - correctness validation

The validation script of the logistic regression serverless implementation validates that the predictions outputted by the serverless implementation are identical to the predictions outputted by the *scikit-learn* library. The validation of the logistic regression serverless implementation has been carried out successfully, by using data sets of various sizes, with various number of features, various number of workers, and various number of inner workers. Respectively, the validation has been carried out using data sets with sizes of [10, 50, 100, 500] MB, with a number of [3, 5, 7] features, with a number of [1, 2, 10, 20, 50] workers, and with a number of [2, 3, 4, 5, 6] inner workers.

Listing 4.2 contains an excerpt from the validation script that verifies the correctness of the serverless implementation of the logistic regression algorithm. The variable X_serial contains the observations of the training set, and the variable y_serial contains the true class labels of the observations from the training set, where both are stored locally on the client machine. The variable X_y_serverless represents the storage object from Amazon S3 that contains the same observations and true class labels as the variables X_serial and y_serial. The logistic regression algorithm from *scikit-learn* is executed locally on the client machine at lines 7-11, by using equivalent parameters that are used for the execution of the serverless implementation at

lines 13-17. At line 19, the correctness of the serverless implementation is validated, as the results of both algorithms are compared, i.e., their prediction results.

```
1  import argparse
2  import numpy as np
3  from sklearn.linear_model import SGDClassifier
4  from logisticRegression_serverless import LogisticRegressionServerless
5
6  ...
7  logisticRegression_serial = SGDClassifier(
8          max_iter=args.max_iter, eta0=args.learning_rate, alpha=0,
9          learning_rate="constant", loss="log_loss", tol=None)
10 logisticRegression_serial.fit(X_serial, y_serial)
11 predictions_serial = logisticRegression_serial.predict(X_serial)
12
13 logisticRegression_serverless = LogisticRegressionServerless(
14         max_iter=args.max_iter, learning_rate=args.learning_rate,
15         n_features=args.n_features, n_workers=args.n_workers)
16 logisticRegression_serverless.fit(X_y_serverless)
17 predictions_serverless = logisticRegression_serverless.predict(X_serial)
18
19 predictions_equal = np.allclose(predictions_serial, predictions_serverless)
```

Listing 4.2: Serverless Logistic Regression - correctness validation

Chapter 5

Experimental Evaluations

Several experiments have been carried out to assess the performance of the serverless implementations. Table 5.1 outlines the general configuration parameters of the evaluation setup used for carrying out the experiments. All resources have been placed in the *eu-west2* region, including the EC2 virtual machines, the AWS Lambda functions, and the Amazon S3 buckets storing the data sets. For minimizing the network latency, the AWS Lambda functions and the EC2 instances, i.e., the Redis node and the client machine, have been configured to run inside the same VPC.

A memory optimized EC2 instance has been used for the virtual machine that hosts Redis, for storing the shared state. A general purpose EC2 instance with 32GB of memory has been used as the client machine that launches the serverless functions. A client machine with 32GB of memory was used for the reason that most experiments involve the k-means algorithm, which outputs one label for every observation of the data set, therefore in the case of large data sets, a considerable amount of memory is needed for being able to store in memory the labels outputted by the serverless functions. Note that the experiments involving the k-means algorithm have initialized different executions of the algorithm with the same centroids to secure a fair evaluation.

Configuration Parameter	**Value**
Resources Region	eu-west2 (Europe - London)
Redis Node Instance Type	r5.large (memory optimized - 2 vCPU, 16GB RAM)
Client Machine Instance Type	t2.2xlarge (general purpose - 8 vCPU, 32GB RAM)
AWS Lambda Timeout	15 minutes

Table 5.1: Experiments Evaluation Setup

5.1 Serverless Synchronization Overheads

An experiment has been carried out to measure the synchronization overheads incurred by barriers. The k-means algorithm has been executed over serverless functions with 1769MB (1 full vCPU) and with a data set of 2GB. The average time spent waiting on a barrier has been measured with an increasing number of workers.

The chart from Figure 5.1 outlines the results of the experiment. The horizontal axis of the chart represents the number of workers used for executing the k-means algorithm, whereas the vertical axis represents the average time spent waiting on a

barrier. Thus, when running 300 workers, a synchronization overhead of 2.72 seconds is incurred by each barrier. Considering that two barriers are used at each iteration for synchronizing the execution of the k-means algorithm, substantial overheads can result when executing the algorithm over a large number of iterations.

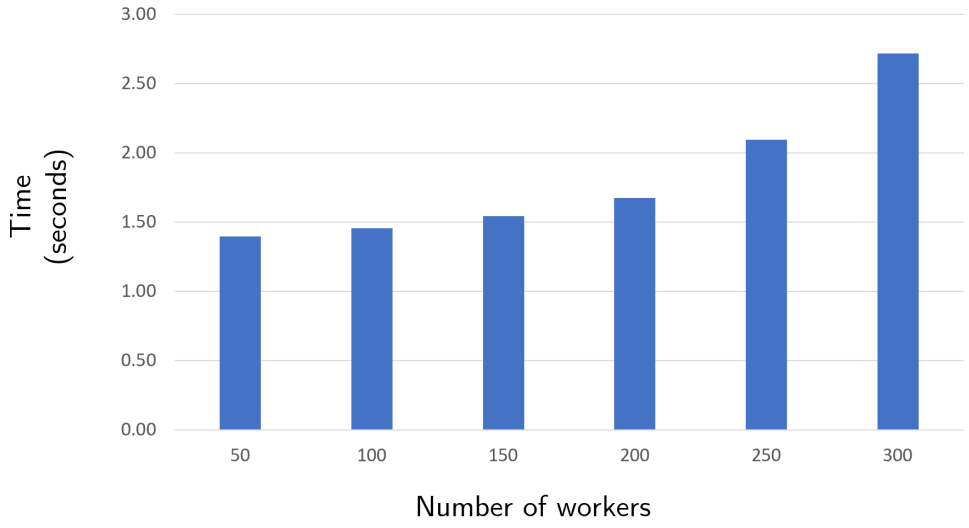

Figure 5.1: Synchronization Overheads - Average Time Spent Waiting on a Barrier

5.2 Serverless Peformance Compared to Sequential

This experiment aims to compare the performance of the serverless (lock-based) implementation of the k-means algorithm with its sequential (i.e., serial) implementation. The sequential implementation of k-means has been executed on a compute optimized EC2 instance, as presented in Table 5.2. Both implementations have used the same 8GB data set. Furthermore, each serverless function has been allocated with 1 full vCPU.

Configuration Parameter	**Value**
AWS Lambda Memory	1769MB (1 full vCPU)
Data Set Size	8GB
EC2 Instance Type	c5.4xlarge (compute optimized - 16 vCPU, 32GB RAM)

Table 5.2: Evaluation Setup - Serverless Performance Compared to Sequential

Figure 5.2 presents the speedup obtained by using a serverless implementation compared to a single-machine sequential implementation. Note that the speedup is

measured as the ratio of the execution time of the sequential implementation, to the execution time of the serverless implementation. Additionally, a speedup factor of n is alternatively called an n-fold speedup. The horizontal axis of the chart represents the number of workers (i.e., concurrent serverless functions) used for executing the k-means algorithm, whereas the vertical axis represents the speedup obtained with a serverless implementation using the specified number of workers. The serverless implementation was launched with a number of concurrent serverless functions that are multiple of 50, up to 400. The peak performance is achieved with 350 workers, obtaining a speedup factor of 87, i.e., an 87-fold speedup. The performance of the serverless implementation improves with the number of workers up to 350 workers, and then the performance begins to decline. The performance decline is to be expected, as the size of the data set is constant throughout the entire experiment.

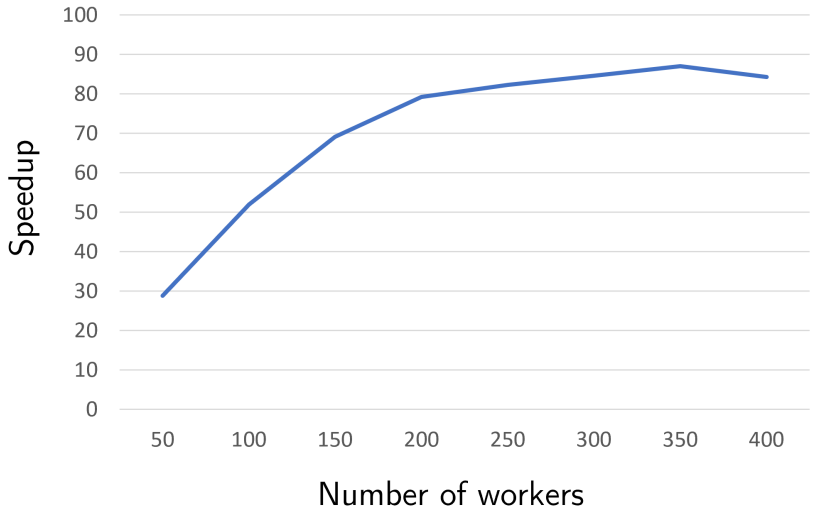

Figure 5.2: Speedup - Serverless Compared to Sequential

5.3 Serverless Peformance Compared to Serverful

This experiment compares the performance of the serverless (lock-based) implementation of the k-means algorithm with its serverful (i.e., parallel and single-machine) implementation. Both implementations have used the same 3GB data set. Moreover, each serverless function has been allocated with 1 full vCPU. The serverful implementation has been realized via Lithops by using *localhost* deployment, and it has been executed on a compute optimized EC2 instance, as presented in Table 5.3. The reason that Lithops was used is that it provides a built-in data partitioner that ensures that the workers (i.e., processes) read their assigned partitions in parallel, thus speeding up the execution of the serverful implementation. The parallelization of the serverful implementation is enabled via processes and a local (i.e., on the same EC2 instance) Redis instance has been set up for storing the shared state of the processes.

Configuration Parameter	Value
AWS Lambda Memory	1769MB (1 full vCPU)
Data Set Size	3GB
EC2 Instance Type	c5.4xlarge (compute optimized - 16 vCPU, 32GB RAM)

Table 5.3: Evaluation Setup - Serverless Performance Compared to Serverful

Figure 5.3 presents the execution time of the serverless implementation compared to the serverful implementation. The horizontal axis of the chart represents the number of workers (i.e., the number of concurrent serverless functions or processes) used for executing the k-means algorithm, whereas the vertical axis represents the execution time. The execution times have been compared when using [8, 12, 16] workers. The chart shows that the serverless implementation is slower when using 8 concurrent serverless functions than the serveful implementation when using 8 processes. Furthermore, the serverless implementation is slightly faster than the serverful implementation when using 12 processes, however, the difference is negligible and could be caused by fluctuating network latencies. Moreover, the serverful implementation is slower than the serverless implementation when using 16 processes. When using all 16 vCPU of the EC2 instance, the performance of the serverful implementation degrades. It can be noted that the performance of the serverful implementation slightly increases with the increase in the level of parallelism, however, the performance declines when all vCPU are used. In contrast, the performance of the serverless implementation improves steadily with the increase in the level of parallelism.

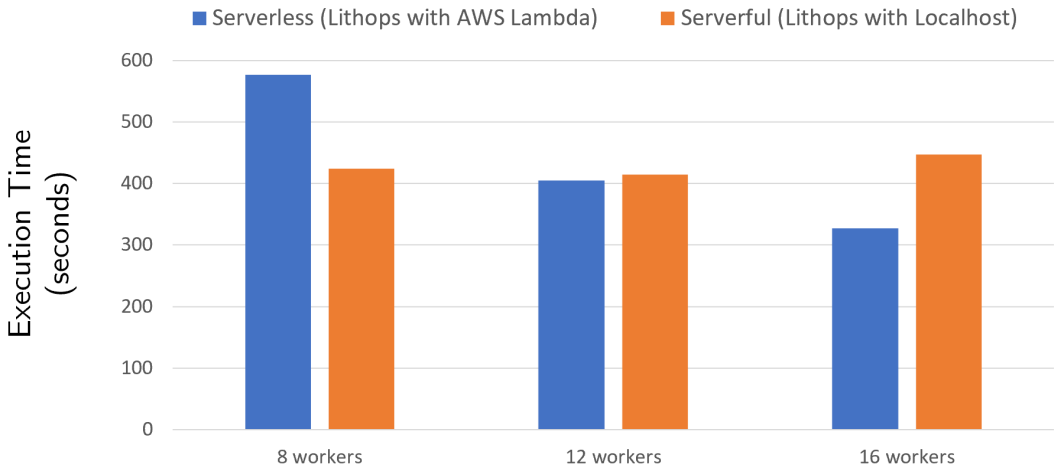

Figure 5.3: Execution Time - Serverless Compared to Serverful

A breakdown of the execution times has been carried out to determine the reason why the performance of the serverless implementation improves with the increase in the number of workers. After analyzing the breakdown of execution times, it can be concluded that the performance of the serverless implementation improves with the increase in the number of workers due to the parallelization of computation and of reading the data set from object storage. Table 5.4 shows that the total time of reading the data set decreases with the increase in the number of workers. Furthermore, the

duration of the computation phase of the k-means algorithm decreases significantly with the increase in the level of parallelism.

No. Workers (Serverless Functions)	Read Dataset	Compute
8	28.83	527.27
12	19.47	341.15
16	14.08	262.12

Table 5.4: Breakdown of Execution Times (seconds) - Serverless Implementation

Similarly, a breakdown of the execution times has been carried out to determine the reason why the performance of the serverful implementation slightly increases with the increase in the number of processes but then declines when a number of processes equal to the number of vCPU are used. Table 5.5 shows that the total time for reading the data set and for performing the computation phase slightly decreases when increasing the number of processes from 8 to 12. However, when using all vCPU of the EC2 instance, the total time of reading the data set and carrying out the computation phase of the k-means algorithm slightly increases.

No. Workers (Processes)	Read Dataset	Compute
8	17.06	378.82
12	11.64	361.79
16	17.37	398.02

Table 5.5: Breakdown of Execution Times (seconds) - Serverful Implementation

An additional serverful implementation of k-means has been realized via the Python Multiprocessing API. The serverful implementation based on Lithops relies on a local Redis node for managing the shared state, however, the implementation based on the Python Multiprocessing API leverages its built-in shared state abstractions. The disadvantage of the serverful implementation based on the Python Multiprocessing API is that it does not provide a built-in data partitioner, therefore the entire data set is read sequentially, rather than in parallel, which consequently slows down the execution of the algorithm. However, one could implement a data partitioner to mitigate this limitation. A breakdown of the execution times of the two serverful implementations using 16 processes is presented in Table 5.6. The breakdown shows that the total synchronization of the k-means algorithm for all iterations via barrier objects has a slightly longer duration in the implementation based on Lithops with *localhost*. Additionally, updating the shared state, i.e., updating the cluster totals and counters, has a slightly longer duration in the implementation based on Lithops with *localhost*. It can be concluded that using the Python Multiprocessing API for the serverful implementation is beneficial as it provides faster access to the shared state compared to accessing a local Redis node for managing the shared state. However, Lithops provides a built-in data partitioner that compensates for these overheads and speeds up the execution of the implementation.

Implementation	Compute	Synchronization	Fetch Shared State	Update Shared State	Aggregate Shared State
Lithops with Localhost	400.021	1.664	0.003	0.020	0.015
Python Multiprocessing API	400.650	1.077	0.005	0.005	0.011

Table 5.6: Breakdown of Execution Times (seconds) - Serverful Implementations

5.4 Serverless Scalability Compared to Serverful

An experiment has been carried out to compare the scalability of the serverless (lock-based) implementation of the k-means algorithm with its serverful implementation. The serverful implementation has been realized via Lithops by using *localhost* deployment, and it has been executed on two compute optimized EC2 instances, with 8 and 16 vCPU, as presented in Table 5.7. The size of the baseline data set is 50MB, i.e., this represents the size of the data set when executing the k-means algorithm with one worker (i.e., one serverless function or one process). Data sets of sizes that are multiples of 50MB are used proportionally to the number of workers. For example, a data set of size 100MB is used when running two workers, and a data set of size 200MB is used when running four workers respectively.

Configuration Parameter	Value
AWS Lambda Memory	256MB
Baseline Data Set Size	50MB
First EC2 Instance Type	c5.2xlarge (compute optimized - 8 vCPU, 16GB RAM)
Second EC2 Instance Type	c5.4xlarge (compute optimized - 16 vCPU, 32GB RAM)

Table 5.7: Evaluation Setup - Serverless Scalability Compared to Serverful

Figure 5.4 presents the scalability of the serverless implementation compared to the serverful implementation. The horizontal axis of the chart represents the number of workers (i.e., the number of concurrent serverless functions or processes) used for executing the k-means algorithm, whereas the vertical axis represents the scale-up. The scalability is measured in terms of *scale-up* $= T_1/T_n$, where T_1 is the execution time using one worker and a data set of 50MB, whereas T_n is the execution time using n workers and a data set of $n\times$50MB. Perfectly linear scalability would be denoted with *scale-up* $= 1$, i.e., the increase in the number of workers can handle the increase in the workload size. The chart shows that the scalability of the serverless implementation is

better than the serverful implementation. The scale-up of the serverless implementation degrades at a slower pace than the serverful implementation, as it can be noted that the scale-up of the serverful implementation quickly degrades. The serverless implementation achieves a scale-up factor of 0.96 with 40 workers, but it lowers down to 0.91 with 100 workers. The serverful implementation running on the EC2 instance with 8 vCPU achieves a scale-up factor of 0.99 with 4 workers, but it lowers down to 0.47 with 8 workers. Correspondingly, the serverful implementation running on the EC2 instance with 16 vCPU achieves a scale-up factor of 0.98 with 8 workers, but it lowers down to 0.65 with 12 workers, and to 0.48 with 16 workers.

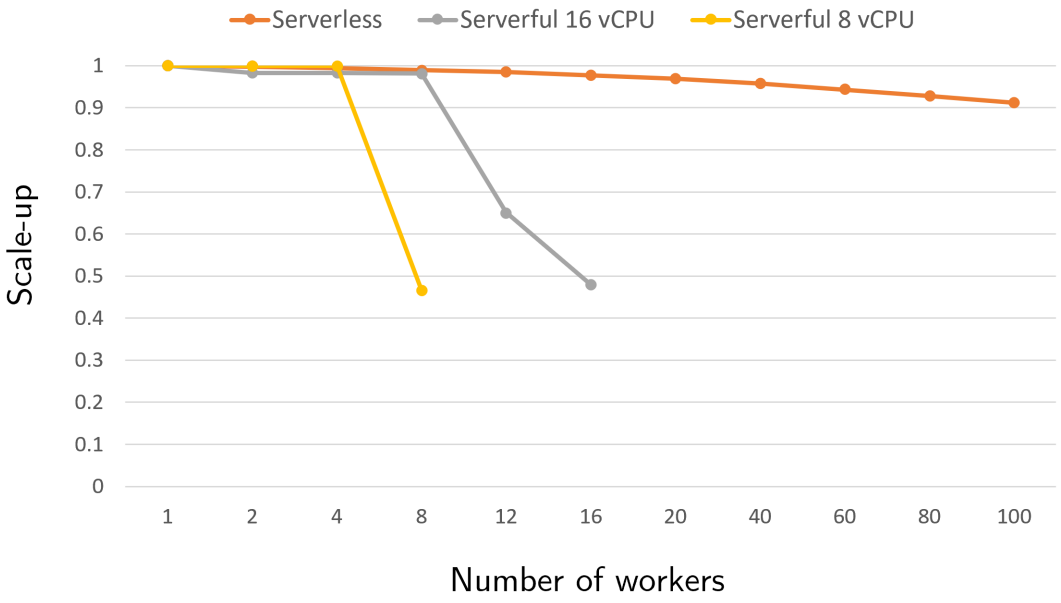

Figure 5.4: Scalability - Serverless Compared to Serverful

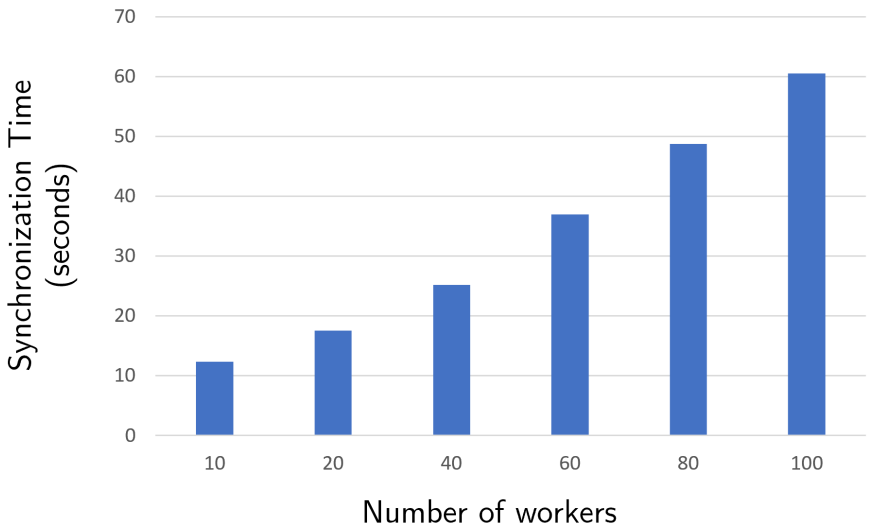

Figure 5.5: Serverless Scalability - Synchronization Time

After analyzing a breakdown of the execution times, it has been determined that the scalability of the serverless implementation slowly degrades due to synchronization

overheads. Figure 5.5 illustrates that the total synchronization times of the serverless implementation increase with the increase in the number of workers. The synchronization times are caused by the total time spent waiting on barrier objects. One of the reasons for this may be that the Redis node cannot handle gracefully requests from multiple connection points, i.e., the concurrent serverless functions.

Similarly, after carrying out a breakdown of the execution times, it has been determined that the scalability of the serverful implementation degrades due to the duration of the computation phase of the algorithm. Table 5.8 presents the execution time of the computation phase when running the algorithm on the two EC2 instances. The results show that as the number of used vCPU increases, the performance of the computation decreases. Note that the duration of the computation phase more than doubles when using all vCPU.

No. Workers	**Compute EC2 c5.4xlarge (16 vCPU)**	**Compute EC2 c5.2xlarge (8 vCPU)**
4	51.22	49.13
8	51.72	49.59
12	70.76	109.26
16	106.80	-

Table 5.8: Serverful Scalability - Computation Time (seconds)

5.5 Serverless Lock-Based and Lock-Free Performance

Experiments have been carried out to evaluate the performance of the two types of serverless implementations, in the context of the k-means algorithm. As a reminder, the first implementation version is lock-based as it requires locks for updating the shared state, whereas the second implementation version is an optimization of the first version, by proposing a lock-free design. Serverless functions with 1GB of memory have been used in the experiments.

The first experiment aims to explore the execution times of the two serverless implementations, with an increasing number of workers. Note that only the execution times of the iteration phase have been measured, as other phases are not relevant for the evaluation (e.g. launching the serverless functions, reading the data sets). The number of clusters used is $k = 3$ and a data set of 1GB has been used. Table 5.9 shows the execution times of the lock-based and lock-free serverless implementations with an increasing number of workers. The results show that the lock-based implementation is slightly slower, by an average of 1.5 seconds. Therefore, the optimization proposed by the lock-free implementation provides minimal added value, as it delivers a negligible performance improvement.

The second experiment compares the execution times of the iteration phase of the two serverless implementations, with an increasing number of clusters. The experiment

No. Workers	Lock-Based	Lock-Free	Delta
10	262.27	260.20	2.07
20	141.65	138.99	2.66
30	97.15	96.36	0.79
40	78.20	76.41	1.79
50	64.25	62.07	2.18
60	55.77	54.23	1.54
70	51.16	49.69	1.47
80	48.38	46.71	1.66
90	44.31	43.07	1.24
100	42.12	41.82	0.29

Table 5.9: Serverless Lock-Based and Lock-Free Execution Time (seconds) by Number of Workers

k clusters	Serverless Implementation	Update Shared State	Aggregate Shared State
100	Lock-Based	11.10	10.57
	Lock-Free	5.58	31.31
200	Lock-Based	14.19	8.20
	Lock-Free	11.04	62.96
300	Lock-Based	31.52	32.27
	Lock-Free	15.17	88.62
400	Lock-Based	45.35	44.15
	Lock-Free	22.17	122.13
500	Lock-Based	57.46	55.27
	Lock-Free	27.00	153.01

Table 5.10: Serverless Lock-Based and Lock-Free Breakdown of Execution Times (seconds) by Number of Clusters

aims to determine the effect of the number of clusters on execution times. Data sets of 100MB have been used in the experiment, where each data set contains data that can be clustered into a different number of k partitions. Note that 10 workers have been used in the experiment. Figure 5.6 shows the performance improvement gained by using the lock-based rather than the lock-free serverless implementation. The performance improvement is measured as a percentage and has been calculated as in Eq. 5.1. The horizontal axis of the chart represents the number of clusters contained in the data sets, whereas the vertical axis represents the performance improvement. The chart shows that with a small number of clusters, the performance improvement has a negative value, whereas the performance improves steadily with the increase in the number of

Figure 5.6: Serverless Lock-Based and Lock-Free Performance Comparison by Number of Clusters

clusters. Therefore, the lock-free implementation is slightly faster than the lock-based implementation when using a number of clusters $k < 50$. However, the performance of the lock-free implementation declines with the increase in the number of clusters, to the point in which it is less performant than the lock-based implementation when using a number of clusters $k > 50$.

$$performanceImprovement = 100\% - \frac{lockBased\_executionTime}{lockFree\_executionTime} \tag{5.1}$$

A breakdown of the execution times has been carried out to determine the reason for the performance degradation of the lock-free implementation with the increase in the number of clusters. Table 5.10 shows that the time spent aggregating the shared state in the lock-free implementation increases significantly with the increase in the number of clusters. The reason for this is that more data objects from the shared state must be aggregated, as in the shared state of the lock-free implementation there is one copy of each data object for every worker. However, the results show that the lock-free implementation does speed up the process of updating the shared state, due to the removal of locks. Nevertheless, the performance improvement for updating the shared state by removing the locks does not compensate for the performance degradation in the aggregation of the shared state. Hence, the performance of the lock-free implementation declines with the increase in the number of clusters.

5.6 Serverless Intra-Function Parallelism Performance

Several experiments have been carried out to evaluate the performance improvement, if any, of enabling intra-function parallelism in serverless functions. Both stateful machine learning algorithms, k-means clustering and logistic regression, have been used in the experiments.

In the first experiment, the k-means algorithm has been executed with a data set of 8GB and with a growing number of concurrent serverless functions (i.e., workers), from 50 up to 400, where each serverless function has 3 vCPU allocated. In the first (baseline) instance, the k-means algorithm was executed without employing intra-function parallelism, i.e., without leveraging the multi-core computing resources of each serverless function. In the second instance, the k-means algorithm was executed by employing intra-function parallelism, by using 2 vCPU of each serverless function, i.e., making use of 2 inner workers for each worker. In the third instance, the k-means algorithm was executed by employing intra-function parallelism, by using all 3 vCPU of each serverless function, i.e., making use of 3 inner workers for each worker. The performance of the algorithm when leveraging intra-function parallelism, i.e., when using 2 and 3 inner workers, has been compared against the baseline, i.e., when not employing intra-function parallelism.

In the context of the described experiment, Figure 5.7 presents the performance improvement gained by leveraging intra-function parallelism. The horizontal axis of the chart represents the number of workers used in the execution of the k-means algorithm, whereas the vertical axis represents the performance improvement obtained by using 2 inner workers and 3 inner workers compared to the case in which intra-function parallelism is not employed. For instance, the chart shows that when running the algorithm with 50 workers and 2 inner workers, a performance improvement of 28% has been obtained compared to when using 50 workers without any inner workers, i.e., without employing intra-function parallelism. Additionally, the chart shows that when running the algorithm with 50 workers, each containing 3 inner workers, a performance improvement of 49% has been obtained compared to when using 50 workers without any inner workers, i.e., without leveraging intra-function parallelism. The results show that one can achieve significantly improved performances when leveraging intra-function parallelism. Note that the performance improvement is measured as a percentage and has been calculated as in Eq. 5.2.

$$performanceImprovement = 100\% - \frac{withInnerWorkers\_executionTime}{withoutInnerWorkers\_executionTime} \quad (5.2)$$

Another experiment has been carried out, that is similar to the previous experiment. The k-means algorithm has been executed with a data set of 8GB and with an increasing number of concurrent serverless functions, from 50 up to 400, however in this experiment, each serverless function has 6 vCPU allocated. Figure 5.8 presents the performance improvement gained by leveraging intra-function parallelism in this

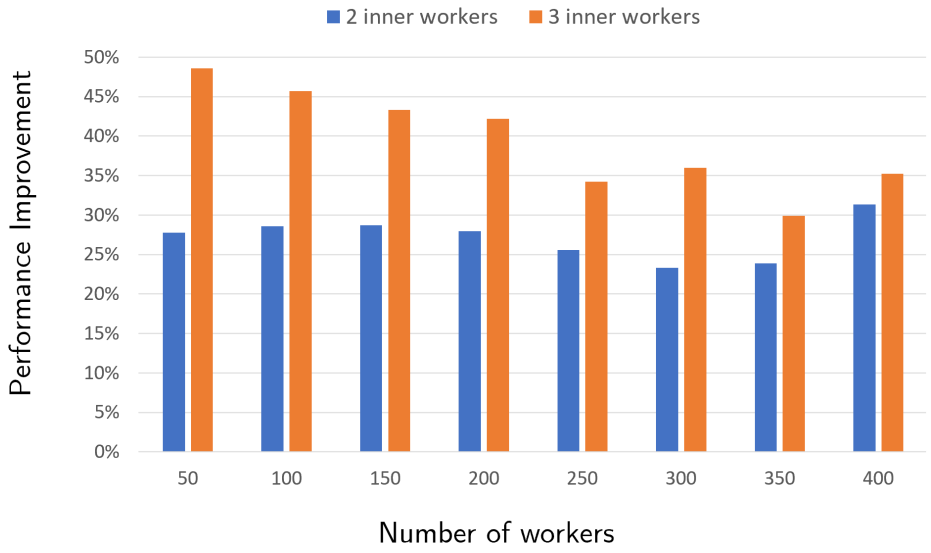

Figure 5.7: Serverless Intra-Function Parallelism Performance using 2 and 3 Inner Workers

experiment. For instance, the chart shows that when running the algorithm with 400 workers and 2 inner workers, a performance improvement of 20% has been obtained compared to when using 400 workers without any inner workers, i.e., without employing intra-function parallelism. Additionally, the chart shows that when running the algorithm with 400 workers, each containing 6 inner workers, a performance improvement of 30% has been obtained compared to when using 400 workers without any inner workers, i.e., without leveraging intra-function parallelism. The results show that one can achieve better performances by up to 68% when leveraging intra-function parallelism. It can be noticed that as the number of workers increases, the performance gained by employing inner workers slowly decreases. Therefore, employing inner workers when the number of workers is very high may provide minimal performance improvements.

One may argue that the previous two experiments do not carry out a fair evaluation. For example, executing 50 workers with 3 inner workers each does not yield the same level of parallelism as 50 workers without inner workers. In the case of executing 50 workers, each with 3 inner workers, 150 vCPU are used, whereas in the case of using 50 workers without inner workers, only 50 vCPU are used. Furthermore, one may argue that instead of using 50 workers with 3 inner workers each, one could instead simply use 150 workers without any inner workers. To validate this proposal, experiments have been carried out the evaluate the performance improvement obtained when leveraging intra-function parallelism, whilst maintaining the same level of parallelism. The aim is to determine whether it is better to use multiple workers, each with one vCPU, without inner workers, or a lesser number of workers with multiple vCPU, each employing multiple inner workers. For example, it is to be determined whether a better performance can be obtained when invoking 50 serverless functions, each with 6 vCPU, that leverage intra-function parallelism, compared to when invoking 300 serverless functions, each with only one vCPU. One may expect fewer synchronization overheads in the case of using 50 serverless functions, each with 6 vCPU, due to the fewer connection points to the Redis node. Furthermore, in the case of using 50 serverless functions, data locality

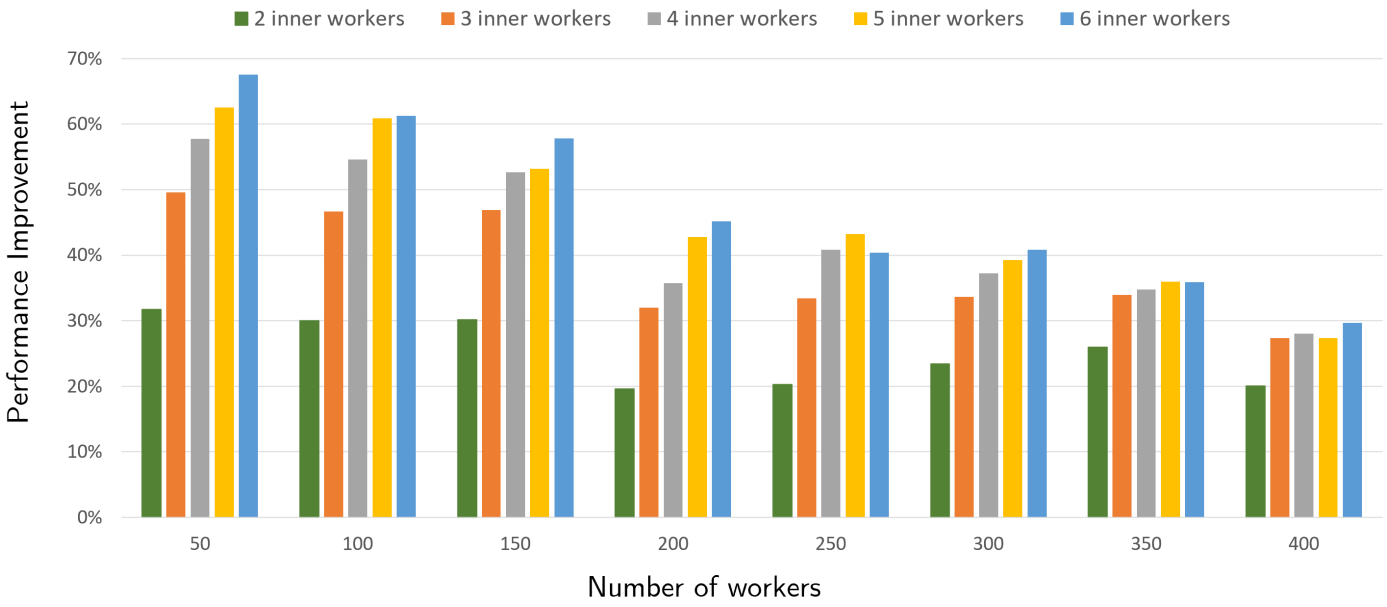

Figure 5.8: Serverless Intra-Function Parallelism Performance using 2 to 6 Inner Workers

may improve the performance of the algorithm. In both cases, a level of parallelism of 300 would be achieved, as a total number of 300 vCPU are used in the execution of the algorithm.

An experiment has been carried out, in which the k-means algorithm has been executed with a data set of 500MB and with various numbers of workers and inner workers, whilst maintaining an equivalent level of parallelism. Table 5.11 presents the results of the experiment. The table shows that the algorithm has realized a level of parallelism of 300 by employing 300 workers without intra-function parallelism, 150 workers with 2 inner workers each, 100 workers with 3 inner workers each, 75 workers with 4 inner workers each, 60 workers with 5 inner workers each, and 50 workers with 6 inner workers each. The results show that a better performance is achieved with a lesser number of workers that have more inner workers. Having many workers causes synchronization overheads due to having many connection points to the Redis node. It can be concluded that leveraging intra-function parallelism is beneficial for improving performance.

The previous experiment has been repeated with the logistic regression algorithm, running for 90 iterations, with a large data set of 120GB. Table 5.12 presents the results of the experiment. In this experiment, a much higher improvement in execution time has been obtained. The reason for the improvement is the large synchronization overheads that are caused by having many workers, i.e., connection points to the Redis node. The reason for the large synchronization overheads is that the logistic regression algorithm is generally executed over a large number of iterations, as the weights of the logistic regression algorithm are updated at each iteration with a learning rate. For instance, having a synchronization overhead of 5 seconds at each iteration would result in an overhead of 500 seconds when running the algorithm over 100 iterations. The results outline that inner workers are especially beneficial when executing stateful

vCPU	AWS Lambda Memory (MB)	Workers	Inner Workers	Execution Time (seconds)	Synchronization (seconds)
1	1500	300	-	61.10	53.79
2	3000	150	2	33.21	23.94
3	4500	100	3	22.92	14.34
4	6000	75	4	19.46	8.59
5	7500	60	5	17.71	8.78
6	9000	50	6	15.66	6.48

Table 5.11: Serverless K-Means - Intra-Function Parallelism Execution Time with Equivalent Level of Parallelism

vCPU	AWS Lambda Memory (MB)	Workers	Inner Workers	Execution Time (seconds)	Synchronization (seconds)
1	1500	300	-	893.81	832.55
2	3000	150	2	474.24	397.94
3	4500	100	3	385.87	284.44
4	6000	75	4	311.60	182.38
5	7500	60	5	288.59	128.74
6	9000	50	6	298.53	117.57

Table 5.12: Serverless Logistic Regression - Intra-Function Parallelism Execution Time with Equivalent Level of Parallelism

machine learning algorithms over many iterations. Algorithms like k-means can converge quickly with a small number of iterations, however, algorithms that make use of a learning rate generally require a large number of iterations. A potential benefit of leveraging inner workers is that the achieved reduction in execution time can enable running more iterations in a stateful machine learning algorithm, considering that serverless functions have a timeout of 15 minutes.

Table 5.13 presents a breakdown of the execution times of this experiment. The durations of various operations have been measured, i.e., for reading the data set, updating the shared state, launching the workers, launching the inner workers, and sending the results of the inner workers to the (parent) workers. As the number of workers decreases, the time spent reading the data set increases. This limitation could have been mitigated by implementing a data partitioner that leverages the inner workers to read the data set in parallel. As the number of workers decreases, less time is required for updating the shared state. Furthermore, launching a lesser number of serverless functions, but with more vCPU, increases the launch duration of the serverless functions by a few seconds. Moreover, as the inner workers are launched and terminated at every iteration, higher overheads are incurred when launching a higher number of inner workers.

The benefit of leveraging inner workers is that it decreases the synchronization overheads, by reducing the number of serverless functions (i.e., workers) that communi-

Workers	Inner Workers	Read Dataset	Update Shared State	Launch Workers	Launch Inner Workers	Inner Workers Send Results
300	-	29.76	7.94	0.92	-	-
150	2	46.09	3.04	1.85	0.66	0.51
100	3	67.05	2.52	1.77	0.85	0.40
75	4	90.21	2.43	1.74	1.92	0.99
60	5	114.74	2.31	1.77	2.27	1.02
50	6	133.68	1.96	1.73	2.79	1.47

Table 5.13: Serverless Logistic Regression - Intra-Function Parallelism Breakdown of Execution Times (seconds)

Workers	Inner Workers	Execution Time	Synchronization	Inner Workers Send Results
300	-	164.38	50.65	-
150	2	180.52	38.92	25.52
100	3	181.39	34.07	26.24
75	4	180.48	24.82	28.34
60	5	188.76	28.28	28.75
50	6	193.92	25.91	30.27

Table 5.14: Serverless K-Means - Intra-Function Parallelism Execution Time (seconds) with Equivalent Level of Parallelism and with Large Output of Inner Workers

cate with the Redis node. However, there may be cases in which reducing synchronization overheads may not be enough for improving the performance of the algorithm. For instance, there may be cases in which the decreased synchronization overheads may be canceled out by high overheads of operations such as the ones presented in Table 5.13. Specifically, if the output of the inner workers is very large, then large overheads may be induced, as sending a large result from the inner workers to the (parent) workers will slow down the execution of the algorithm.

The experiment from Table 5.11 has been repeated but with a large data set of 15GB, containing 299439300 data points. As the output of the inner workers is a set of labels, one for each data point, it follows that the output of the inner workers is large. Therefore, a significant transfer overhead is incurred when sending the resulting labels of the inner workers to the (parent) workers at each iteration. Table 5.14 presents the results of this experiment. The results show that the decreased synchronization times caused by employing inner workers are canceled out by the overheads of sending the results from the inner workers to the (parent) workers. Thus, inner workers are not beneficial when the output of the inner workers is very large. In this particular case, this is a limitation of AWS Lambda running Python code, as state between processes can only be shared via pipes that inherently incur transfer overheads.

Chapter 6

Related Work

Several research works have shown the benefits of leveraging serverless computing for data analytics applications. However, the majority of these efforts have been carried out in the context of embarrassingly parallel workloads such as hyperparameter tuning [36], Monte Carlo simulations [57], analytics queries [58, 59, 60], video encoding [61], tone analysis of Airbnb[1] reviews [35], and other simple MapReduce jobs [25, 62, 63]. Nevertheless, additional research efforts have been carried out for tackling stateful data analytics applications by leveraging serverless architectures [64, 65, 66, 67].

The most significant works that are related to this research project have been carried out in [64, 67]. Crucial[2] is a framework that facilitates the development of stateful distributed applications with serverless architectures. The framework is based on the Java programming language, hence parallelism is achieved via threads rather than concurrent processes, as one thread is mapped to an invocation of a serverless function. From the point of view of Crucial, serverless functions are seen as a set of cloud threads that communicate through shared state. Crucial organizes the shared state in a layer of distributed shared objects (DSO). The DSO layer has been implemented using Infinispan[3], a low-latency in-memory data store. The advantage of Crucial is that one can implement complex computations as object methods in the DSO layer, that can be called by serverless functions to perform aggregations. Hence, stateful applications implemented using Crucial can achieve improved performance when executing complex and aggregation operations over the shared state. As Crucial benefits from disjoint-access parallelism, it achieves better performances than the solutions from this research project, considering that Redis is single-threaded.

The work from [64, 67] leverages Crucial to implement two serverless machine learning algorithms (i.e., k-means clustering and logistic regression) that require shared state. The scalability of the serverless implementation of the k-means clustering algorithm has been compared against a serverful implementation of the algorithm. A scale-up factor of 94% has been achieved with 160 cloud threads, whereas a scale-up factor of 90% has been achieved with 320 cloud threads, due to the overhead of thread creation. Note that these results are better than the results obtained in this research project. Moreover, the iteration's phase running time of the Crucial implementation of the k-means clustering and logistic regressing algorithms have been compared against Apache Spark [68]. The results show that logistic regression is 18% faster in Crucial than Apache Spark, whereas k-means clustering is 40% faster in Crucial than Apache Spark with $k = 25$ clusters, however, the time gap decreases with the increase in the number of clusters.

[1] Airbnb is a service for property owners to rent out their property to travelers

[2] `https://github.com/crucial-project`

[3] `https://infinispan.org`

Several works opt to store the shared state to highly-scalable but slow disaggregated object stores [25, 35, 69]. Other works rely on fast in-memory stores for storing the shared state [67] and on a mixture of storage mediums [59]. Moreover, another storage medium for shared state can be shared logs, as presented in the Boki serverless runtime [70].

Function composition is an orchestration technique that can be leveraged to provide share state and coordination among serverless functions. The main public cloud providers offer cloud services that enable function composition: AWS Step Functions[4], Google Cloud Composer[5], Azure Durable Functions[6] and the IBM Composer[7]. Nevertheless, the enumerated cloud services have poor performance when running highly parallel compositions [71, 72]. PyWren has addressed this limitation, as it was the first system to provide coordination by employing the BSP synchronization protocol [25]. Although function composition is a technique that can provide shared state, it is not explored throughout this research project.

Other works have demonstrated the benefits of using serverless architectures for the implementation of machine learning algorithms that require shared state [65, 66]. SIREN is proposed in [65], an asynchronous distributed machine learning framework based on serverless architectures. A serverless implementation of logistic regression with SGD based on SIREN has been compared against its equivalent serverful implementation. The evaluation results have shown that the serverless implementation needed 22.9% less training time to reach the same loss. Furthermore, a comprehensive study has been carried out in [66] to outline the tradeoffs between training stateful machine learning models on serverless and serverful architectures.

CIRRUS [73] is a machine learning framework that leverages serverless infrastructures for the purpose of enabling robust and efficient iterative machine learning training. Distributed machine learning training is carried out via concurrent workers that compute model parameters and share intermediate results on a distributed data store. CIRRUS has been evaluated against PyWren on AWS Lambda, with logistic regression. The term *big lambda* is defined in [73] as an AWS Lambda function with a large amount of memory. The evaluation results of CIRRUS show that bigger lambdas achieve higher throughput, however, small lambdas are more efficient with regard to costs.

Faaslets are a lightweight isolation abstraction that delivers efficient shared memory access to stateful serverless functions [74]. The memory of serverless functions is isolated via the software-fault isolation (SFI) facilities of WebAssembly, enabling memory regions to be shared among functions located in the same address space. A two-tier state architecture is used by Faaslets: a local tier that supports in-memory data sharing for co-located functions, and a global tier implemented as a distributed Redis instance that enables distributed state access across the entire system. Faaslets are executed by FAASM[8], a distributed serverless runtime that provides an API for fine-grained state management to Faaslets.

The work from [75] presents SAND, a serverless computing system that ensures

[4] `https://aws.amazon.com/step-functions`

[5] `https://cloud.google.com/composer`

[6] `https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview`

[7] `https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-pkg_composer`

[8] `https://github.com/faasm/faasm`

low latency and a high level of resource efficiency. SAND relaxes isolation at the application level, and thus serverless functions belonging to the same application are able to share memory and communicate via a hierarchical message bus.

Cloudburst [76] is a stateful FaaS platform that aims to increase the feasibility of stateful serverless functions. This is carried out by reducing to a reasonable level the overheads of managing the shared state. To this end, Cloudburst achieves local disaggregation and physical colocation of computation and state. State sharing across serverless functions is realized by leveraging Anna [77], an autoscaling key-value store. Similarly to this research project, the focus is on making stateful serverless functions practical, however, optimization techniques are not of interest in [76].

In [78], complex stateful applications are ported to the Apache OpenWhisk serverless platform. The work aims to migrate existing stateful applications to serverless, with minimal code changes whilst achieving comparable performance. For storing the shared state, an external storage service is used. If the application to be ported already uses an external storage service for other purposes, that external storage service is leveraged to store the shared state, to minimize code changes. However, if the application to be ported does not already use an external storage service, the application is modified such that it will use a remote storage service (e.g., Amazon S3) for storing the shared state. Similarly to this research project, a set of observations and guidelines for the portage to serverless are reported, but the goal is not to migrate machine learning algorithms, but rather complex stateful applications, such as, Overleaf[9], Social Network [79], LAB Insurance Sales Portal [80] and Robot Shop [81].

In the context of serverless computing, the majority of the research works have adopted a so-called inter-function parallelism pattern for parallelizing applications across a set of individual function instances [82]. Due to the increased computational capabilities of serverless functions offered by cloud providers, one can now achieve intra-function parallelism within a serverless function [30, 82, 83]. Thus, engineers can take advantage of the multi-core computing resources of a single serverless function to parallelize its execution via threads or processes. Significant cost savings have been achieved in [83] by leveraging intra-function parallelism: 81% cost savings with AWS Lambda and 49% with Google Cloud Functions. It can be concluded that the works from the literature leverage inter-function parallelism for the implementation of serverless machine learning algorithms that require shared state. However, no efforts have been carried out for tackling stateful machine learning algorithms by leveraging intra-function parallelism.

[9] `https://www.overleaf.com`

Chapter 7

Conclusions

In this work, a technique has been presented for porting to serverless two stateful machine learning algorithms, k-means clustering and logistic regression. There are several limitations and challenges that are encountered when porting stateful machine learning algorithms to serverless. The focal limitation is that serverless functions are not directly network-addressable, hence one must rely on a remote storage service for storing the shared state. However, using a remote storage service for storing the shared state induces communication overheads, as the serverless functions are required to access and update the shared state at each iteration of the algorithms. The overheads are amplified by the number of executed iterations, as having hundreds of concurrent serverless functions accessing a remote storage service for hundreds of iterations can result in significant overheads. At the implementation level, the challenge is to access the Redis node as little as possible, for minimizing communication overheads. Hence, the workers should operate over data that is stored in local memory as much as it is feasible, and then update the shared state only when necessary. In addition, bulk operations for accessing and updating the shared state should be used where possible, for minimizing the number of requests to the Redis node.

The first challenge of porting stateful machine learning algorithms to serverless is to identify what parts of the algorithms can be parallelized, and specifically, what partial results can be computed independently and concurrently by the workers. In the case of the k-means algorithm, the partial results are the cluster totals and cluster counters, whereas in logistic regression, the partial results are the sub-gradients. By using the partial results, the global results are computed, i.e., the centroids in k-means, and the weights and global gradients in logistic regression. The partial and global results are stored in the shared state, and at the end of each iteration, one of the workers (i.e., the first worker) fetches the partial results of all workers from the shared state, aggregates the partial results to obtain the global results, and then updates the shared state with the global results. The limitation of this approach is that it causes additional overheads by executing operations that are not typically executed in a sequential implementation, i.e., fetching the partial results of all workers, aggregating the partial results and updating the shared state with the global results. For instance, large overheads have been demonstrated in one of the experiments executing the serverless k-means algorithm with a data set containing 500 clusters, in which the total duration of updating the shared state was of 57.46 seconds, whereas the total duration of aggregating data from the shared state was of 153.01 seconds.

To ensure the correctness of the results of the stateful machine learning algorithms, the BSP synchronization protocol has been employed in the serverless implementations. For this purpose, two barrier objects have been used as synchronization points at each iteration. However, this has the limitation that waiting on barriers can yield substantial overheads, especially when the number of workers is high. In the context of

the k-means algorithm, it has been shown that when running 300 workers, the average time spent waiting on a barrier is of 2.72 seconds. Taking into account that there are two barriers at each iteration, and that iterative machine learning algorithms such as logistic regression are typically executed over hundreds or thousands of iterations, this can cause enormous overheads. In one of the experiments, it has been shown that executing the serverless logistic regression algorithm with 300 workers had a total execution time of 893.81 seconds, whereas the synchronization overheads caused by barriers accounted for 832.55 seconds, i.e., 93% of the total execution time.

Two versions have been proposed for both algorithms, a lock-based implementation, and a lock-free implementation which aims to be an optimization of the lock-based implementation. In the lock-based implementation, there is one single copy of each data object being stored globally in the shared state. Therefore, locks have been used for preventing concurrent serverless functions from simultaneously accessing the same data objects from the shared state. However, locks have the limitation of carrying additional overheads. As such, the lock-free version of the algorithms has been implemented for avoiding the overheads caused by locks. For this purpose, in the shared state, there is stored one copy of each data object for every worker. With this approach, each worker has his own memory space in the global shared state and each worker only accesses data objects that are associated with him, and consequently, access conflicts are avoided. Nevertheless, it has been demonstrated in the context of the k-means algorithm that the lock-free implementation is only slightly faster than the lock-based implementation when having a small number of clusters, however, the lock-based implementation is faster when having a large number of clusters. The time spent aggregating the shared state in the lock-free implementation increases with the number of clusters, as more data objects from the shared state must be aggregated at each iteration, considering that in the shared state there is one copy of each data object for every worker. Although removing the locks speeds up the process of updating the shared state, the performance improvement does not compensate for the performance degradation in the aggregation of shared state. Through experiments, it has been shown that when executing the k-means algorithm with a number of 3 clusters, the lock-free implementation is faster by an average of 1.5 seconds than the lock-based implementation. Therefore, the optimization proposed by the lock-free implementation provides a negligible performance improvement. Furthermore, it has been shown that with 500 clusters, the lock-free implementation is faster by 30.46 seconds in the phase of updating the shared state, however, it is slower by 97.74 seconds in the phase of aggregating the shared state, and consequently, it is slower overall than the lock-based implementation.

Although the serverless implementations inherently induce overheads, the performance improvement achieved by parallelizing the algorithms compensates for the overheads. An experiment has been carried out to evaluate the performance of the serverless (lock-based) implementation of the k-means algorithm compared to its sequential implementation. The results of the experiment have shown that the serverless implementation achieves an 87-fold speedup compared to the sequential implementation when running the k-means algorithm with 350 workers.

The performance of the serverless (lock-based) implementation of the k-means algorithm has been compared to a serverful implementation in an experiment. The results of the experiment have shown that the performance of the serverful implementation

slightly increases with the increase in the number of workers, however, the performance declines when all vCPU are used. When all vCPU are used, the total execution time of the serveful implementation increases due to the performance degradation in the phases of computation and of reading the data set. In contrast, the serverless implementation achieves a steady performance improvement with the increase in the number of workers, due to the parallelization of computation and of reading the data set.

An experiment has been carried out to evaluate the scalability of the serverless and serverful implementations of the k-means algorithm. The serverless implementation achieved a scale-up factor of 0.96 with 40 workers, but this declines to 0.91 with 100 workers, due to synchronization overheads. The serverful implementation was executed on two EC2 instances, with 8 and 16 vCPU. When running on the EC2 instance with 16 vCPU, the serverful implementation achieved a scale-up factor of 0.98 with 8 workers, but this lowers down to 0.65 with 12 workers, and to 0.48 with 16 workers. The decline in scalability is caused by the performance degradation of the computation phase with the increase of used vCPU.

Intra-function parallelism has been employed as an optimization technique to improve the performance of the stateful machine learning algorithms. At every iteration of the algorithms, each worker launches a number of inner workers, where the number is dictated by the number of vCPU of the serverless function. Each worker splits his data set partition into smaller partition chunks, and then each inner worker performs the computation phase in parallel over the smaller partition chunks. The drawback of this approach is that it incurs additional transfer overheads by sending the results from the inner workers to the (parent) workers and overheads caused by the aggregation of the results received from the inner workers. The k-means algorithm has been executed in an experiment with an increasing number of concurrent serverless functions, where each serverless function had 6 vCPU allocated. The performance of the algorithm has been compared in the case in which intra-function parallelism is not employed, to the case in which intra-function parallelism is employed by leveraging the multicore computing resources of each serverless function. The results of the experiment have shown that one can achieve better performances by up to 68% when leveraging intra-function parallelism. This experiment does not carry out a fair evaluation, as for example, executing 50 workers with 3 inner workers each, does not yield the same level of parallelism as 50 workers without inner workers. For this purpose, additional experiments have been carried out to evaluate the performance improvement obtained when leveraging intra-function parallelism, whilst maintaining the same level of parallelism. The experiments have confirmed that better performances are achieved with a lesser number of workers, where each worker has inner workers, compared to a higher number of workers, that do not have inner workers.

In future work, it would be beneficial to validate the generality of the findings from this research project using other stateful machine learning algorithms. Further work could involve porting the algorithms to other cloud providers and then comparing the obtained results to the ones obtained in this research project. Moreover, a different line of work could involve using a programming language other than Python, to avoid the limitation of the Python GIL, which prevents true multithreading. In addition, the existing data partitioner from Lithops could be extended to leverage intra-function parallelism for partitioning the data sets.

Bibliography

[1] AWS, *What is cloud computing?* URL: `https://aws.amazon.com/what-is-cloud-computing` (Last Accessed: 2023-09-01).

[2] L. Qian, Z. Luo, Y. Du, and L. Guo, "Cloud Computing: An Overview," in *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1*, pp. 626–631, Springer, 2009.

[3] L. Wang, G. Von Laszewski, A. Younge, X. He, M. Kunze, J. Tao, and C. Fu, "Cloud Computing: a Perspective Study," *New Generation Computing*, vol. 28, pp. 137–146, 2010.

[4] IBM, *Cloud computing*. URL: `https://www.ibm.com/topics/cloud-computing` (Last Accessed: 2023-09-01).

[5] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "Above the Clouds: A Berkeley View of Cloud Computing," *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, vol. 28, no. 13, p. 2009, 2009.

[6] J. R. Vacca, *Cloud Computing Security (2nd Edition)*. CRC press, 2020.

[7] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless Computing: One Step Forward, Two Steps Back," *arXiv preprint arXiv:1812.03651*, 2018.

[8] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," *arXiv preprint arXiv:1902.03383*, 2019.

[9] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The Rise of Serverless Computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[10] M. Fowler, *Serverless Architectures*. URL: `https://martinfowler.com/articles/serverless.html` (Last Accessed: 2023-09-01).

[11] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network Requirements for Resource Disaggregation.," in *OSDI*, vol. 16, pp. 249–264, 2016.

[12] Y. Shan, *Distributing and Disaggregating Hardware Resources in Data Centers*. University of California, San Diego, 2022.

[13] A. D. Papaioannou, R. Nejabati, and D. Simeonidou, "The Benefits of a Disaggregated Data Centre: A Resource Allocation Approach," in *2016 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–7, IEEE, 2016.

[14] C. Guo, X. Wang, G. Shen, S. Bose, J. Xu, and M. Zukerman, "Exploring the Benefits of Resource Disaggregation for Service Reliability in Data Centers," *IEEE Transactions on Cloud Computing*, 2022.

[15] N. T. Pemberton, *The Serverless Datacenter: Hardware and Software Techniques for Resource Disaggregation*. University of California, Berkeley, 2022.

[16] V. Yussupov, U. Breitenbücher, A. Brogi, L. Harzenetter, F. Leymann, and J. Soldani, "Serverless or Serverful? A Pattern-Based Approach for Exploring Hosting Alternatives," in *Service-Oriented Computing: 16th Symposium and Summer School, SummerSOC 2022, Hersonissos, Crete, Greece, July 3–9, 2022, Revised Selected Papers*, pp. 45–67, Springer, 2022.

[17] AWS, *Amazon EC2 Instance Types*. URL: `https://aws.amazon.com/ec2/instance-types` (Last Accessed: 2023-09-01).

[18] P. García-López, M. Sánchez-Artigas, S. Shillaker, P. Pietzuch, D. Breitgand, G. Vernik, P. Sutra, T. Tarrant, and A. J. Ferrer, "Servermix: Tradeoffs and Challenges of Serverless Data Analytics," *arXiv preprint arXiv:1907.11465*, 2019.

[19] AWS, *Deploying Lambda functions*. URL: `https://docs.aws.amazon.com/lambda/latest/dg/lambda-deploy-functions.html` (Last Accessed: 2023-09-01).

[20] AWS, *Amazon VPC*. URL: `https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html` (Last Accessed: 2023-09-01).

[21] AWS, *Configuring AWS Lambda functions*. URL: `https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html` (Last Accessed: 2023-09-01).

[22] AWS, *Parallel Processing in Python with AWS Lambda*. URL: `https://aws.amazon.com/blogs/compute/parallel-processing-in-python-with-aws-lambda` (Last Accessed: 2023-09-01).

[23] AWS, *Amazon S3 Guide*. URL: `https://docs.aws.amazon.com/AmazonS3/latest/userguide` (Last Accessed: 2023-09-01).

[24] Redis, *List Commands*. URL: `https://redis.io/commands/?group=list` (Last Accessed: 2023-09-01).

[25] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 445–451, 2017.

[26] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin, "Task Parallelism and Data Distribution: An Overview of Explicit Parallel Programming Languages," in *Languages and Compilers for Parallel Computing: 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers 25*, pp. 174–189, Springer, 2013.

[27] I. Foster, "Task Parallelism and High-Performance Languages," *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pp. 179–196, 2005.

[28] M. Khan, Y. Liu, and M. Li, "Data Locality in Hadoop Cluster Systems," in *2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pp. 720–724, IEEE, 2014.

[29] M. McCool, A. D. Robison, and J. Reinders, "Chapter 8 - Fork–Join," in *Structured Parallel Programming* (M. McCool, A. D. Robison, and J. Reinders, eds.), pp. 209–251, Boston: Morgan Kaufmann, 2012.

[30] Z. Wen, Y. Wang, and F. Liu, "Stepconf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pp. 1868–1877, IEEE, 2022.

[31] G. Shroff, *MapReduce and extensions*, p. 131–143. Cambridge University Press, 2010.

[32] A. Elsayed, O. Ismail, and M. E. El-Sharkawi, "MapReduce: State-of-the-Art and Research Directions," *International Journal of Computer and Electrical Engineering*, vol. 6, no. 1, p. 34, 2014.

[33] A. B. Bondi, "Characteristics of Scalability and Their Impact on Performance," in *Proceedings of the 2nd international workshop on Software and performance*, pp. 195–203, 2000.

[34] Python, *Multiprocessing API*. URL: `https://docs.python.org/3/library/multiprocessing.html` (Last Accessed: 2023-09-01).

[35] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless Data Analytics in the IBM Cloud," in *Proceedings of the 19th International Middleware Conference Industry*, pp. 1–8, 2018.

[36] J. Sampe, M. Sanchez-Artigas, G. Vernik, I. Yehekzel, and P. Garcia-Lopez, "Outsourcing Data Processing Jobs With Lithops," *IEEE Transactions on Cloud Computing*, 2021.

[37] Python, *Concurrent Futures API*. URL: `https://docs.python.org/3/library/concurrent.futures.html` (Last Accessed: 2023-09-01).

[38] Python, *Futures*. URL: `https://docs.python.org/3/library/asyncio-future.html` (Last Accessed: 2023-09-01).

[39] A. Arjona, G. Finol, and P. G. López, "Transparent serverless execution of Python multiprocessing applications," *Future Generation Computer Systems*, vol. 140, pp. 436–449, 2023.

[40] S. Haseena, M. B. B. Pepsi, and S. Saroja, "Multi Criteria Decision Making Technique for Machine Learning Algorithms: Iterative and Non-Iterative Algorithms," *International Journal of Scientific & Technology Research*, vol. 9, no. 1, pp. 2392–2403, 2020.

[41] K. Sud, P. Erdogmus, and S. Kadry, *Introduction to Data Science and Machine Learning*. IntechOpen, 2020.

[42] S. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[43] A. M. Ikotun, A. E. Ezugwu, L. Abualigah, B. Abuhaija, and J. Heming, "K-means Clustering Algorithms: A Comprehensive Review, Variants Analysis, and Advances in the Era of Big Data," *Information Sciences*, 2022.

[44] E. U. Oti, M. O. Olusola, F. C. Eze, and S. U. Enogwe, "Comprehensive Review of K-Means Clustering Algorithms," *Criterion*, vol. 12, pp. 22–23, 2021.

[45] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd Edition)*. Springer, 2009.

[46] D. Jurafsky and J. H. Martin, *Speech and Language Processing (3rd Edition draft)*. 2023.

[47] J. M. Hilbe, *Practical Guide to Logistic Regression*. CRC Press, 2016.

[48] T. M. Mitchell, *Machine Learning*. McGraw-Hill Education, 1997.

[49] E. P. Xing, Q. Ho, P. Xie, and D. Wei, "Strategies and Principles of Distributed Machine Learning on Big Data," *Engineering*, vol. 2, no. 2, pp. 179–195, 2016.

[50] D. Steinkraus, I. Buck, and P. Simard, "Using GPUs for machine learning algorithms," in *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pp. 1115–1120, IEEE, 2005.

[51] S. Mittal and S. Vaishay, "A survey of techniques for optimizing deep learning on GPUs," *Journal of Systems Architecture*, vol. 99, p. 101635, 2019.

[52] D. Schlegel, "Deep machine learning on GPUs," *University of Heidelber-Ziti*, vol. 12, 2015.

[53] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A CPU and GPU math compiler in Python," in *Proc. 9th python in science conf*, vol. 1, pp. 3–10, 2010.

[54] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A Survey on Distributed Machine Learning," *ACM Computing Surveys*, vol. 53, no. 2, pp. 1–33, 2020.

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[56] E. Bressert, "SciPy and NumPy: an overview for developers," 2012.

[57] M. E. Mirabelli, P. García-López, and G. Vernik, "Bringing scaling transparency to Proteomics applications with serverless computing," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pp. 55–60, 2020.

[58] Y. Kim and J. Lin, "Serverless Data Analytics with Flint," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 451–455, IEEE, 2018.

[59] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure," in *NSDI*, vol. 19, pp. 193–206, 2019.

[60] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 115–130, 2020.

[61] S. Fouladi, R. S. Wahby, B. Shacklett, K. 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 *NSDI*, vol. 17, pp. 363–376, 2017.

[62] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, "Understanding Ephemeral Storage for Serverless Analytics," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 789–794, 2018.

[63] A. Bhat, H. Park, and M. Roy, "Evaluating Serverless Architecture for Big Data Enterprise Applications," in *2021 IEEE/ACM 8th International Conference on Big Data Computing, Applications and Technologies (BDCAT'21)*, pp. 1–8, 2021.

[64] 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*, pp. 41–54, 2019.

[65] H. Wang, D. Niu, and B. Li, "Distributed Machine Learning with a Serverless Architecture," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 1288–1296, IEEE, 2019.

[66] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards Demystifying Serverless Machine Learning Training," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 857–871, 2021.

[67] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful Serverless Computing with CRUCIAL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–38, 2022.

[68] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pp. 15–28, 2012.

[69] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless linear algebra," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 281–295, 2020.

[70] Z. Jia and E. Witchel, "Boki: Stateful Serverless Computing with Shared Logs," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 691–707, 2021.

[71] P. G. López, M. Sánchez-Artigas, G. París, D. B. Pons, Á. R. Ollobarren, and D. A. Pinto, "Comparison of FaaS Orchestration Systems," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 148–153, IEEE, 2018.

[72] D. Barcelona-Pons, P. García-López, Á. Ruiz, A. Gómez-Gómez, G. París, and M. Sánchez-Artigas, "Faas Orchestration of Parallel Workloads," in *Proceedings of the 5th International Workshop on Serverless Computing*, pp. 25–30, 2019.

[73] 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*, pp. 13–24, 2019.

[74] S. Shillaker and P. Pietzuch, "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 419–433, 2020.

[75] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," in *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pp. 923–935, 2018.

[76] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful Functions-as-a-Service," *arXiv preprint arXiv:2001.04592*, 2020.

[77] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in Anna," *Proceedings of the VLDB Endowment*, vol. 12, no. 6, pp. 624–638, 2019.

[78] Z. Jin, Y. Zhu, J. Zhu, D. Yu, C. Li, R. Chen, I. E. Akkus, and Y. Xu, "Lessons learned from migrating complex stateful applications onto serverless platforms," in *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 89–96, 2021.

[79] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, and B. Jackson, *Social Network - One Project of DeathStarBench*, 2023. URL: `https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork` (Last Accessed: 2023-09-01).

[80] ASC LAB, *LAB Insurance Sales Portal - A Simplified Insurance Sales System*, 2021. URL: `https://github.com/asc-lab/micronaut-microservices-poc` (Last Accessed: 2023-09-01).

[81] Stan, *Robot Shop - A Sample Microservice Application*, 2023. URL: `https://github.com/instana/robot-shop` (Last Accessed: 2023-09-01).

[82] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless Computing: State-of-the-Art, Challenges and Opportunities," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1522–1539, 2022.

[83] M. Kiener, M. Chadha, and M. Gerndt, "Towards Demystifying Intra-Function Parallelism in Serverless Computing," in *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, pp. 42–49, 2021.