UNIVERSITAT ROVIRA i VIRGILI

Universitat Rovira i Virgili (URV) y Universitat Oberta de Catalunya (UOC)

Master in Computational and Mathematical Engineering

FINAL MASTER PROJECT

Area: Distributed Systems

Study of the Feasibility of Serverless Access Transparency for Python Multiprocessing Applications

---

Authors: Gerard Finol Peñalver and Aitor Arjona Pérez

Tutor: Dr. Pedro García López

Barcelona/Tarragona, June 20, 2021

I, Dr. Pedro García López, certify that the students Gerard Finol Peñalver and Aitor Arjona Pérez have elaborated the work under my direction and I authorize the presentation of this memory for its evaluation.

Credits/Copyright

FINAL PROJECT SHEET

Title:	Study of the Feasibility of Serverless Access Transparency for Python Multiprocessing Applications
Authors:	Gerard Finol Peñalver and Aitor Arjona Pérez
Tutor:	Dr. Pedro García López
Date (mm/yyyy):	06/2021
Program:	Master in Computational and Mathematical Engineering
Area:	Distributed Systems
Language:	English
Key words	Transparency, Disaggregation, Serverless

Acknowledgments

This work has been carried out as part of a research project led by the CLOUDLAB-URV research group in the context of the European CloudButton Horizon 2020 project during the authors' stay in the group as project grant-holder fellows.

Abstract

Access transparency means that both local and remote resources are accessed using identical operations. Transparency simplifies the complexity of programming a distributed system because the system is perceived as a whole rather than a collection of independent components. With access transparency, we can treat disaggregated compute, storage, and memory resources as if they were a single monolithic machine. This would considerably simplify the creation and execution of parallel applications in the Cloud in a scalable manner.

In this work, we evaluate the feasibility of access transparency over state-of-the-art Cloud disaggregated resources. We propose an alternative implementation of Python's multiprocessing API that transparently runs distributed processes on serverless functions and that leverages disaggregated in-memory storage to maintain the shared state of processes consistent and mediate their communication.

To evaluate transparency, we have used four parallel stateful applications intended to be executed locally (Uber Research's Evolution Strategies, Baselines-AI's Proximal Policy Optimization, Pandaral·lel's dataframe and ScikitLearn's Hyperparameter tuning), and, without changing the code, we have scaled them with serverless technology. We compare execution time and scalability of the same application running over disaggregated resources using our library, with the single-machine Python libraries in a large VM. Despite the higher latency and lower throughput of communication, we achieve comparable results and we observe that the applications can continue to scale beyond VM limited resources leading to a better speedup and parallelism.

Keywords: Transparency, Disaggregation, Serverless

Contents

List of Figures

List of Tables

Introduction

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components [15]. In their book, Colouris et al. define eight types of transparency: access, location, concurrency, replication, failure, mobility, performance and scaling. **Access transparency** enables local and remote resources to be accessed using identical operations, and **Location transparency** enables resources to be accessed without knowledge of their physical or network location. They state that access and location are the two most significant types of transparency, since their presence or absence has the most impact on how distributed resources are used. An example of application would be Dropbox [17]. Dropbox provides both access and location transparency, because the remote files are accessed using the file explorer, which transparently combines local and remote files, and the user does not know the exact location of those files. The conjunction of access and location transparency is known as **Network transparency**.

Transparency allows to hide the complexity of distributed systems from the user, applications and programmers. This fact substantially increases the simplicity of applications and the programmer's productivity. Since there is a separation between the interface layer and the distributed system layer, the latter can change and adapt without the programmer or user awareness. For instance, following the Dropbox example, the user sees that a file can be moved to a shared folder and the same file appears a brief moment later in another device's folder automatically. Underneath, the system is storing the file in a database that the user does not need to know about, and even an automatic replication of the file is taking place for availability and fault tolerance purposes, where the user is also unaware of it.

Ideally, using local operations for both local and remote resources would be completely equivalent. Then, full and perfect transparency could be achieved. This would have significant implications, since we could program an application following a centralized and local vision of the system, but that could be deployed in a distributed way and scale beyond the limits imposed by a monolithic machine. Transparency facilitates moving legacy applications that were originally designed to be deployed in a monolithic manner to more modern Cloud scenarios

where resources are flexible and can be adapted to the usage required by the application. In such a case, new iterations of software modernization for legacy applications and methods such as "lift and shift" [19] could be avoided.

However, early studies suggest that providing access transparency is not ideal. In [31], the authors conclude that, in the context of Object Oriented Programming, the usage of remote objects as if they were local is incorrect and leads to performance issues and general unreliability. The main drawback is the huge latency overhead for remote method invocations and distributed memory access compared to their local counterparts. Also, failures are harder to handle since the components in a distributed system fail independently while a centralized system fails as a whole. In short, the programmer has to keep in mind that they are programming a distributed system even if local operations are being used. Although the authors did not discuss access transparency in their article, their conclusions suggest that it is not feasible.

Trend in public Clouds is to offer disaggregated services where resources are accessed through the network. The main advantage of disaggregated resources is that the user does not have to worry about administration, since the provider is in charge of provisioning resources, guaranteeing security and availability, and scaling the resources on demand [21]. One example of disaggregated compute service is Function as a Service (FaaS) [2], where the provider offers computing resources to execute arbitrary user code in response to events (like an HTTP request). Serverless functions only run when necessary, and they scale back to zero when they are not used. This results in a pay-per-use billing model, where the user only pays for used resources instead of allocated resources. The counterpoint is the billing models of other services like EC2, where the user pays a fixed per-hour price for allocated resources, regardless of whether they are being used or not. Serverless functions have proven to be effective for massive parallel computing applications [29, 44] and even Big Data processing at scale [42, 23]. Disaggregated storage services are also present in all major public Cloud providers. For example, Amazon Web Services (AWS) offers S3 as a disaggregated object storage service, where arbitrary data blobs are stored in persistent, consistent [3], replicated and fault tolerant storage organized into objects and buckets. Lastly, disaggregated memory services are also available, like AWS Elasticache [1]. AWS Elasticache is a managed Redis or Memcached service, where the user pays for the resources allocated but does not have to worry about security and management, since it is provided by the Cloud provider.

As a result of advances in reducing the latency of access to disaggregated resources, it is estimated that achieving transparency of access is becoming increasingly feasible [35]. Reductions in network latency are already enabling resource disaggregation [24, 7], which in turn makes it feasible to run unmodified single-machine applications over disaggregated resources. Ultimately, lower latencies imply disaggregation which implies transparency.

In this thesis, we evaluate whether the inherent scalability of serverless functions, together with a disaggregated and consistent in-memory storage component, enables to transparently run unmodified parallel Python applications over disaggregated serverless compute resources at scale.

The open questions studied in this thesis are:

1. Can we program the Cloud as if it were a local machine?
2. Can we just grab some local parallel application and massively scale it at the *Serverless Super-Computer*[48]?
3. Can we already give up middleware and achieve true access transparency?

Our contributions with this research work are:

1. Design and implementation of Python's multiprocessing API interface that transparently runs distributed processes over serverless functions and communicates them using disaggregated in-memory storage. We also designed several optimizations (work queue, shared state, synchronization) that mitigate existing communication overheads.
2. Validation of the feasibility of access transparency by comparing the execution of 4 unmodified applications (Evolution Strategies, Proximal Policy Optimization, Scikit-Learn Grid Search and Pandas dataframes) on a single VM and over serverless functions to analyze the speedup and overheads.
3. We outline key insights from this study: it is possible to run and scale unmodified Python multiprocessing applications with minor degradation compared to a VM, existing serverless overheads were partially masked by Hyper-Threading inefficiencies observed in a single VM and Python's multiprocessing shared memory abstractions clearly facilitated transparency. Other languages directly accessing memory references or pointers would be more complex to intercept.

Chapter 1

Background

Before we get started, in this chapter we introduce the core concepts discussed in this thesis. In the first section, we will look at the evolution of the infrastructure and architecture of distributed applications, from *on-prem* to Cloud. Then we will see what Serverless is and what benefits it offers compared to other computing models in the Cloud. Then we will present the problems of Cloud adoption for legacy applications and how transparency can facilitate this process. We also highlight the implications of transparency for the distributed systems' community. Finally, we summarize the parallel programming model of the Python programming language, which will be useful for our validation and contributions.

1.1 Cloud and Serverless computing

In the previous years, we have witnessed a general adoption of the public Clouds by multitude of companies [26]. Some even rely solely on the resources offered in public Clouds for the deployment of their entire web infrastructure. To understand how we got to this point, we must first revisit what was there prior to the Cloud.

In the past, tech companies used to deploy their software architectures and applications on their self-owned physical infrastructure. This early solution, although effective, was very inefficient. When an application or service had to be deployed, the system administrators had to prepare the hardware, install the operating system on each machine with its drivers, prepare the network, and finally deploy the software. In this case, the deployment unit is a physical server. This solution is known as *on-prem* and the applications were developed following a *monolithic architecture*. The main shortcoming of this solution is that the resource capacity is fixed, i.e., it will always be the same regardless of the work load. This presents a problem in supporting fluctuating workloads. For example, a website could become very busy from one day to another because someone shared it elsewhere and it got a lot of attention and visibility. If

the infrastructure is not prepared to support a sudden increase of traffic, the site will probably go down and it will lose interest quickly. In brief, *on-prem* architectures are inefficient, costly and hard to maintain, provision and scale. The deployment and preparation of new servers could take several days up to weeks or months, and worst of all, it is difficult to foresee the required resources needed.

In 2006, Amazon Web Services pioneered as a public Cloud [37]. From the lessons learned over the years by the internal engineers that developed and made it what `amazon.com` is today, they proposed to lend their owned infrastructure to anyone who needed it and only charge per hour for the resources that were used. Elastic Compute Cloud (EC2) was born, and with it, the term *Infrastructure as a Service* (*IaaS*), although at the time it was not yet conceived. *IaaS* allows users to access fundamental computing resources (servers, network and storage) on-demand and on a pay-as-you-go basis [47]. These computing resources are shared among all cloud provider customers (more commonly called *tenants*). Virtualization is used to prevent multiple applications from different users colliding on the same physical server and causing conflicts. OS-level virtualization simulates an operating system (called *guest*) running on top of another operating system (called *host*) which runs a special software (called *hypervisor*) that allows to run several and isolated *virtual machines* on one shared and physical machine. *IaaS* is a big step forward compared to *on-prem*. First, system administrators can forget about having to manage, deploy and maintain physical hardware, and not just servers, but everything that goes with it (network, operating system...). The Cloud provider is in charge of all administration tasks, but also the provider guarantees security, availability and integrity of compute resources. Not only that, but resource flexibility is greatly enhanced. The clients can now dispose of the exact resources that best fit their needs, and they will pay for the exact amount of resources they have allocated. If in the future the customer wishes to adapt and scale the resources of their infrastructure, now it's as easy as ordering new virtual machines. In *IaaS*, the unit of deployment is a virtual server.

Over time, new services flourished in the offerings of public Clouds. Each service has a specific functionality, and when combined, it is possible to create complex web architectures entirely in the **Cloud**. According to IBM Cloud Learning Hub [47], the definition of Cloud is:

> "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."

Although *IaaS* solves many of the problems that *on-prem* solutions present, administrators still have to manage servers. That is, administrators are no longer responsible for maintaining

physical servers, but are still responsible for deploying software, preparing dependencies and managing auto-scaling policies. Administrators often over-provision resources in order to have a capacity margin in the event of a sudden raise in user traffic to the application or web service. As a result, customers end up with more allocated resources than they really need, leading to both an energetic and monetary waste, since the CSP charges for allocated resources regardless of their use.

In November 2014, AWS introduced a new service called AWS Lambda. This innovative service allowed developers to run arbitrary code in response to events on an environment provided and managed by the CSP, where allocated resources scale up or down to zero automatically based on user requests and the user is only charged for resources used at millisecond granularity. Collaterally, this service supposed the next big step forward in Cloud computing: the Serverless computing model. According to IBM Cloud Learn Hub [21], the definition of serverless is:

> "Serverless is a cloud computing execution model that provisions computing resources on demand, and offloads all responsibility for common infrastructure management tasks (e.g., scaling, scheduling, patching, provisioning, etc.) to cloud providers and tools, allowing engineers to focus their time and effort on the business logic specific to their applications or process."

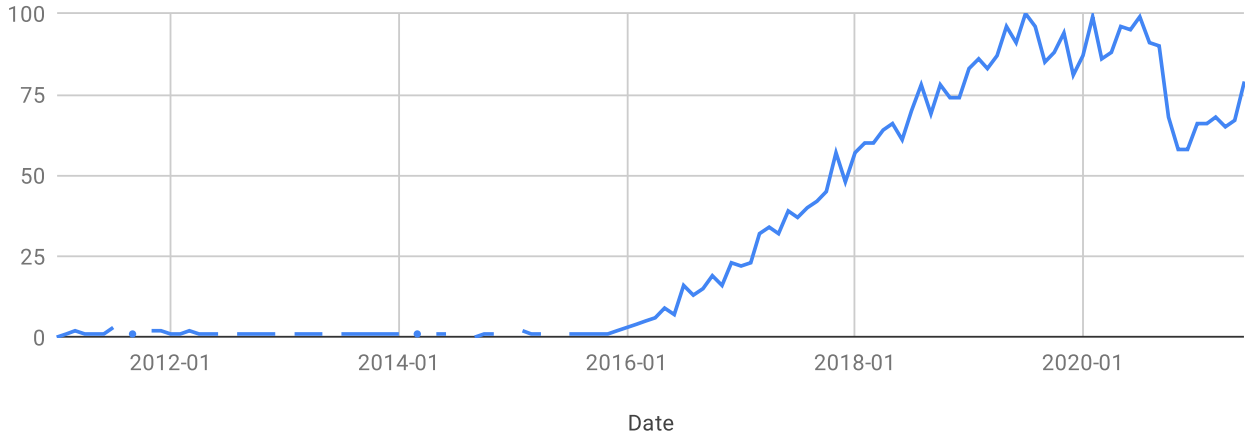

Figure 1.1: "Serverless" term Google search interest over time [27]

The serverless paradigm quickly became relevant thanks to its huge benefits compared to the *IaaS* model (Figure 1.1). Castro et al. review in their article "The rise of Serverless computing" [13] the importance of this new paradigm. They present Serverless as a simpler programming model for building cloud applications that abstracts away most of *dev-ops* concerns. Administrators do no longer have to worry about things like availability, scalability, fault

tolerance, over- or under-provisioning virtual machine resources, server management, and other infrastructure hassles. Instead, developers can concentrate on the commercial aspects and logic business of their applications. In addition, by charging for execution time rather than resource allocation, the cost of maintaining the infrastructure is lower, since the price is adjusted to the resources that are actually needed at any given time.

The term "serverless" can be sometimes confusing and may lead one to think that there are no servers. Obviously, servers still exist – the term refers to the fact that serverless services are server *worry*-less. The most common form of serverless is Function as a Service or *FaaS*. Although serverless and *FaaS* are frequently confused, the truth is that *FaaS* is, in fact, a subset of serverless. Serverless is focused on any service category where the configuration, management, and payment of servers are transparent to the end user, such as compute, storage, database, messaging, API gateways... *FaaS*, on the other hand, is centered on the event-driven computing paradigm, in which application code, or containers, only run in response to events or requests, like an HTTP request [18]. In *FaaS*, the deployment unit is a *function*.

With the growing popularity of cloud-native and scalable architectures, such as micro-services, where a single application is decomposed in many loosely coupled and independently deployable smaller components [20], serverless has certainly found a place in many web architectures. The great flexibility and scalability of serverless services, and more specifically *FaaS*, has given way to the emergence of applications that take advantage of serverless potential and capabilities for other uses beyond web services. One of the projects that had the most impact in its day was PyWren, presented in Eric Jonas et al. article *Occupy the cloud: Distributed computing for the 99%* [29]. In their paper, the authors presented PyWren, a Python library to easily deploy sequential and local code over hundreds of serverless functions. PyWren goal was to port embarrassingly parallel jobs and massively scale them leveraging AWS Lambda. Based on PyWren, *numpywren* [44] was another project that implemented linear algebra algorithms, such as matrix multiplication and factorization, over serverless functions. In [42], the authors present IBM-PyWren, a fork of PyWren built for IBM Cloud. IBM-PyWren expanded the original PyWren project, that was only available for AWS Lambda, and adapted it to IBM Cloud. Using IBM Cloud Functions had an advantage over AWS Lambda because the former allows to use a container image (Docker containers) as runtime, whereas in AWS Lambda, the runtime was set by the provider. Using a container image as runtime improves the flexibility and capabilities, and allows a wider range of possibilities. In a container image, the user can dispose of more complex Python modules or system libraries that otherwise were not available in AWS Lambda. In addition, the authors expanded the capabilities of the library by being able to run *map-reduce* workloads on serverless functions. In their paper, the authors explore the usage of serverless functions for mass processing of Big-Data stored in the IBM Cloud Object

Storage service. They carried out an experiment that read and applied natural language processing and sentiment analysis on thousands of AirBnB comments. They were able to achieve a speedup of $135x$ using IBM-PyWren compared to the same application running sequentially on a virtual machine. Eventually, the project was spun off from PyWren and was renamed to Lithops. Lithops [41] is a serverless framework that allows to easily ship local sequential code to the cloud and execute it at massive scale using thousands of parallel serverless functions. Lithops offers more than IBM-PyWren since it follows a multi-cloud and multi-service model, being able to use serverless functions, or Kubernetes, or containers, or virtual machines in *IaaS* to execute and scale the user's code, all using a simple and consistent API through clouds and platforms.

1.2 Moving applications to the cloud

Cloud adoption by businesses has been progressive over the years. According to the 2020 IDG Cloud Computing Survey [28], 51% of businesses had at least one application or portion of their infrastructure was on a Cloud in 2011. Nowadays, the percentage exceeds 93% and is expected to reach 95% in the coming year. Although we clearly see that the Cloud is becoming more and more prevalent mainly for its flexibility and lower administration cost, the paradigm shift from *on-prem* to Cloud is so fundamentally different that makes it difficult to adapt existing applications and architectures for a Cloud environment. It is understandable that companies which own legacy complex applications struggle to move them to the Cloud. These applications usually have a huge source code, have been reliable for years, are basically their whole business architecture and were designed following a monolithic architecture. In comparison, the applications that are being built today are already designed and thought for a Cloud environment. The most commonly used architecture is *micro-services*, as it is the most efficient and effective for Cloud environments but also the complete opposite of the monolithic architectures that legacy applications implemented.

As a result, we see that the Cloud has a potential that is not being fully exploited, and Cloud migration is still a relevant issue. In Flexera 2021 State of the Cloud Report [22], 59% of respondents stated that migrating more workloads to the Cloud was one of their goal to achieve in 2021, and 39% stated that they were aiming to move *on-prem* software to *SaaS* (Software as a Service). As we can see, moving applications to the Cloud is not only relevant but also a problem that remains one of the top priorities for businesses. Furthermore, in the same survey we see that 61% of respondents are looking to optimize their cloud usage in order to lower the cost and 45% are trying to improve their finances and reports on their Cloud spending. Also, respondents estimate that their organizations waste 30% percent of cloud spend, and,

on average of 24%, their public cloud spend was over budget. It is remarkable to note that optimizing existing Cloud usage (cost reductions) and moving additional workloads to cloud have been the top two priorities for the fifth year in a row. Although it may not seem apparent, these two aspects are actually correlated. The Cloud offers mechanisms and flexible services to adjust and adapt resources to the actual usage based on the workload, as in the case of *FaaS*. However, only cloud-native or architectures can really leverage and take advantages of these services. A legacy application rarely makes use of these services. For this reason, businesses are investing heavily and make efforts in adapting and transforming their existing applications to be deployed to the Cloud.

In [33], they distinguish three levels of software modernization to be moved to the Cloud:

1. **Re-hosting**: Also known as "Lift & Shift" [19]. It implies transferring a legacy application to the cloud "as-is". Re-hosting an application is a rapid, relatively secure, and simple migration solution, but does not benefit from the full potential that the Cloud can offer.

2. **Re-platforming:** Moving a legacy application to the Cloud with minor architecture and code modifications. This option leverages some Cloud services, such as moving from an on-premises database to a managed database such as AWS RDS.

3. **Re-factoring/Re-architecting:** It implies major modifications to the whole application, such as splitting up a legacy monolith architecture into smaller self-contained components or micro-services. This scenario leverages a wider variety of Cloud services for simpler application operation, as well as Cloud features that involve scalability, availability, and failure resilience. This option is the most expensive but the most beneficial in the long term.

1.3 Transparency implications in software modernization

So far we have seen aspects of moving applications to the Cloud and its relevance, but we have not yet seen what implications transparency has on this software modernization. In fact, these two aspects are directly related. Through transparency, we can avoid the process of software modernization and port applications to the Cloud automatically, efficiently and directly.

However, providing transparency is not trivial, and assuming that an application intended to be run locally will have the same performance as it running in a distributed environment with disaggregated resources is a mistake.

For example, in [51], the authors tested the performance of two Data Base Management Systems, PostgreSQL and MonetDB, running on disaggregated resources without prior modifications. The result was a terrible performance, with overheads of up to an order of magnitude greater. The problem was mainly memory access, since the time to access local memory is several times faster than accessing remote memory. In another study [7], the authors are pessimistic and think that it will not be possible to obtain good results from applications running on disaggregated resources unless the operating system expose the nature of disaggregated resources to the applications and let them exploit it for their benefit. For example, OSs can expose where processes are located in order to exploit memory locality, boost performance and lower overheads generated from data transfers. However, this implies losing the point of transparency, since we are (albeit to a lesser extent) modifying the application to fit a distributed and disaggregated environment.

However, we believe that, although there is clear evidence that transparency and disaggregation are two complicated concepts to weave together, some applications can benefit from the elasticity and flexibility of Cloud disaggregated and serverless resources to achieve better performance. In the article [34], the authors compile a series of predictions and advancements that, given today's advances and research interests, could see the light in the near future. Recent advancements in network latency reductions [40, 10] might lead in achieving access transparency to disaggregated resources. Also, new RPCs technologies [32, 30] that can achieve times as low as micro-seconds (µs) are an incentive to transparency.

The feasibility of access transparency in distributed systems has important impacts and consequences [34]:

1. **Software development:** Although developers will still need to care about parallel programming, they will not need to deal with distributed systems-related stuff: middleware, transport layers, data marshaling, web-based protocols, Cloud-specific APIs...

2. **Simplicity:** Transparency can have a huge impact on development costs due to increased productivity. Writing remote apps will become as straightforward as programming local applications, but with unlimited resources.

3. **Democratization of the Cloud:** Transparency will also increase the usage of Cloud resources by less expert users, allowing for the production of applications in different (scientific or not) fields. With the help of new visual and even *no-code* tools and GUIs, more users will be able to build custom applications leveraging Cloud resources.

4. **Competing companies**: Transparency will also lead to more competitiveness between companies both in the Cloud sector (Amazon, Google, Microsoft...) to show who has the

best and cheapest services, but in other markets as well, such as edge devices (Apple, Google (Android)...) to see who makes the best use of new Cloud capabilities. Competition ends up being beneficial for consumers as we get better products and services at more competitive prices.

It should be noted that in this thesis we are not presenting a definitive and magical solution for disaggregated access transparency completely. We are not claiming that access transparency is completely viable and has no drawbacks. In fact, we start from the assumption that transparency presents serious problems related to memory access overheads. This thesis aims to study where the barrier to access transparency lies using current Cloud technologies.

In the following chapter, we will go through the current state of the art and see some research proposals that deal with resource disaggregation and transparency, and how they manage to run single-machine applications in a distributed environment with disaggregated resources.

1.4 Multiprocessing

The aim of this thesis is to leverage Python's multiprocessing library as means to validate transparency for local-parallel Python applications. In this section, we provide an overview of the main classes and abstractions of the module, so that readers unfamiliar with this library can understand the design decisions made in this research.

In Python, concurrency using threads is not possible due to the global interpreter lock (GIL) implemented in the CPython interpreter. According to the Python glossary [5], the GIL is defined as:

> "The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as dict) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines."

For this reason, Python contains the `multiprocessing` [6] module in its standard library. The `multiprocessing` package offers the developer an API that supports spawning processes and allowing concurrency, side-stepping the Global Interpreter Lock by using subprocesses instead of threads. `multiprocessing` is considered the module par excellence to obtain concurrency in Python, and it can be really complex since it has many abstractions and classes.

In `multiprocessing`, processes are spawned by creating a `Process` object and then calling its `start()` method. In the constructor of the `Process` class, you must define the function that

it will execute and its arguments. An example of the use of this class can be seen in the Listing 1.

Listing 1 Process usage example from [6].

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Communication between processes can be done through the use of two types of abstractions, `Pipes` and `Queues`. `Pipes` offer a one-to-one communication paradigm. When you create a `Pipe`, you get two connections (representing the two ends of the `Pipe`) that processes can use to write or read data. Instead, the `Queue` offers one-to-all communication by offering an abstraction where sent data is stored, and it is not removed until a process reads it. Listing 2 and 3 contain examples of the usage of `Pipe` and `Queue` respectively.

Listing 2 Pipe usage example from [6].

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

For process synchronization, multiprocessing offers typical abstractions like `Lock`, `Semaphore` and `Barrier`. `Lock` objects can be used with its two main methods acquire and release. `Barriers` have a method `wait` to block execution until all processes have made their `wait()` calls.

The `multiprocessing` module provides two methods to share state between processes: Shared memory and `Managers`. Data can be stored in a shared memory map using `Value` and `Array` objects, that are process and thread-safe. This objects only support basic C types

Listing 3 Queue usage example from [6].

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

predefined in the submodule `multiprocessing.sharedctypes`, and data type must be provided when creating `Value` or `Array` objects. A `Manager` object controls a server process that maintains a shared state based on various types of Python objects. The `Manager` allows you to manipulate these objects ensuring the consistency of their state. It supports many types of objects such as `list`, `dict`, `Lock`, `Barrier`, `Queue`...

One of the most used and known abstractions of multiprocessing is `Pool`. `Pool` is a class that represents a pool of worker processes to which jobs can be submitted. The constructor of this class allows you to specify how many workers processes will be used. In case this parameter is not specified, the number of CPUs available in the system is used. The `Pool` implements a set of methods that allow different types of tasks to be carried out, some of the best known methods are: `map`, `apply`, `imap`, `starmap` and their corresponding asynchronous implementations. For example, the function `map()`, which receives as parameters a function f and an iterable object (such as a list), applies the function f to each element of the iterable object. The methods of the `Pool` distribute their tasks among all its workers processes in an automatic and transparent way to the programmer, thus facilitating parallel programming. Listing 4 contains an example code using `Pool`.

1.5 Hyper-threading

Hyper-Threading technology, proposed by Intel[1], was designed to maximize the use of a CPU's resources. To do this, different elements are duplicated within the processor chip, such as general purpose registers, control registers, pipeline registers, or first-level caches. This allows two or more threads to run on each CPU. For example, if one thread waits for an interrupt

[1]Intel is not the only manufacturer that offers this type of technology and practically all manufacturers have an implementation of a similar technology. Even so, Intel's implementation is the best known and its name is often generalized to other manufacturers.

Listing 4 Pool usage example from [6].

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))
```

or performs I / O tasks, another thread can use the ALU to perform logical operations and thus the ALU is not left unused. Thus, for each physically present CPU with Hyper-Threading technology enabled, the operating system will perceive two (or more) CPUs. These CPUs seen by the operating system are often called logical or virtual CPUs.

Note that the main execution resources such as arithmetic logic unit (ALU), address generation unit (AGU) or floating-point unit (FPU) are not duplicated. Due to this, the number of arithmetic operation (or floating point operations) per second is limited by the ALU (or FPU) and the Hyper-Threading technology does not allow us to exceed that limit, but it does allow us to approach that limit when threads don't use those resources completely.

Most cloud providers that offer a FaaS products allow you to configure the number of virtual CPUs (vCPUs) available for each function. The virtualization technology of most cloud providers ends up mapping the vCPUs to the logical CPUs of the Hyper-Threading technology. Therefore, when we execute code in a FaaS service, it may be that we are sharing physical CPU with the code of other users. Something similar happens with virtual machines, cloud providers offer and classify them based on the vCPUs that will be available, but in reality the amount of physical CPUs is usually half, since they use Hyper-Threading technology by default.

Chapter 2

Related work

In this chapter we present related work on the transparency of access to distributed resources in the Cloud. First, we show that achieving transparency is not new and that studies have already been conducted in the past. Then, we present two research lines in terms of disaggregation and transparency: OS-level transparency and application-level transparency. For each of them, we present the research projects that sustain these researches and what problems they present. Finally, we relate our research work to the related work and explain how we improve it.

2.1 Achieving transparency

Attempting to hide the complexity of distributed systems is not a new goal. Waldo et al. [31] already argued back in 1994 that treating a distributed system as if it was a local system leads to problems related to latency, shared memory access, concurrency and partial failures. In their paper, the authors predict that future hardware and network improvements will help to mitigate latency problems, which implies that the access to local and remote resources may eventually be indistinguishable. However, even if latency could be eventually obviated at some point, the tolerance to partial faults and concurrency issues still remain as open problems.

2.2 OS-level Transparency

A current research trend is studying full resource disaggregation in Disaggregated Data Centers (DDC), which is the counterpart to the traditional monolithic approach where computational resources reside in a centralized "black-box" unit (i.e. a server). A DDC opts for a fully disaggregated architecture, where hardware components are attached to a network and they are accessed remotely. This vision could significantly improve resource utilization, elasticity, and heterogeneity. In DDCs, the Operating System (OS) transparently leverages disaggregated

resources like compute, memory and storage, so that the application has a virtual view of a single-node architecture when in reality it is completely distributed and disaggregated.

2.2.1 Lego-OS

LegoOS [43] is a disaggregated OS that implements a subset of the Linux system call interface so that existing unmodified Linux applications can run on top of it. The authors of LegoOS demonstrate how two unmodified applications can be executed in it using distributed disaggregated resources: Phoenix (a single-node multi-threaded implementation of MapReduce) and TensorFlow. Nevertheless, LegoOS is not demonstrating scalability or complex scenarios involving mutable memory. In addition, implementing the entire Linux system calls API for disaggregated resources is a very complex and error-prone task, which makes this solution far from stable and still requires further maturation until we see viable results.

2.2.2 GiantVM

Another approach in access transparency at the operating system layer is GiantVM [50]. GiantVM uses virtualization to run an unmodified guest OS over a distributed compute cluster. In contrast to the traditional many-to-one virtualization paradigm (running multiple OS in one machine), GiantVM implements one-to-many virtualization (running a single OS in many machines). GiantVM uses *Infrastructure as a Service* (IaaS) to run the distributed OS over a cluster in a Cloud setting. Each node runs KVM and virtualizes the guest OS using QEMU. Each node is interconnected so that the guest OS has a centralized view of all resources distributed among the cluster. The memory is shared and distributed across the host nodes. The guest OS has a virtual continuous view of all memory space. They validate GianVM using a stress benchmark suite, where the results show that this approach is somewhat slower ($1.34x$) than using the same benchmark on a single virtualized machine, where the biggest overheads are generated by the distributed memory management. They show another application (word count and inverted indexing) where GiantVM outperforms the same application running on a Spark cluster with the same resources. The way I/O operations are implemented on Spark explain the degradation of performance compared to GiantVM. The main drawback of GiantVM is that it does not provide fault tolerance at the moment, meaning that if a node fails, the guest OS stops working and can't be automatically recovered. Although these approaches that use current cloud architectures are currently more feasible until DDCs are available to general public, providing fault tolerance to these types of systems is complex and shows that more research is still needed in this field in order to achieve a production-stable solution.

2.3 Application-level Transparency

Another way to achieve transparency is at the application level, instead of at the operating system level like the examples stated above. If the interface that is used by the application to access local resources is replaced by another implementation that accesses remote resources instead, we could then transparently run unmodified local code in a distributed fashion.

2.3.1 Fiber

For instance, Fiber [52] is a library developed by Uber Research that interfaces Python's multiprocessing API to run remote processes in a distributed Kubernetes cluster. In their article, they execute multiple stateful Artificial Intelligence (AI) applications that are programmed for local parallel execution using Python's multiprocessing library. By replacing multiprocessing with Fiber, those unmodified applications can transparently scale and exploit parallelism on a distributed Kubernetes cluster.

One benefit of using Kubernetes is that processes can leverage direct communication for faster data transfer. However, one disadvantage of using a complete distributed model is that synchronization abstractions (like a lock) are harder to implement, and require a centralized component that manages synchronization, which is prone to failures. In fact, the authors did not implement all stateful and synchronization abstractions available on Python's multiprocessing module. Their justification is that AI algorithms rarely use synchronization primitives, so they obviated their implementation.

Actually, Fiber's objective is not to study transparency at all. Instead, they wanted to provide a distributed programming framework so that AI developers unfamiliar with distributed programming could easily scale their algorithms using a library with known and familiar abstractions such as Python's multiprocessing library. This is why Fiber does not implement all stateful abstractions of the multiprocessing library nor does it follow a strict implementation of the interface contracts. Instead of being a drop-in replacement for multiprocessing to scale already-written applications without much effort, Fiber is intended to be used in newly developed applications that are specially programmed to be used with it. A plus point of Fiber is that it is designed to be deployed on Kubernetes. Kubernetes is a container orchestration platform that is widely adopted by all major Cloud providers and has been proven to be robust in production through the years. Creating a Kubernetes cluster, whether it is managed in a public Cloud service or on-prem, is very simple nowadays. Also, containers can leverage the Container Network Interface (CNI) to have direct communication between containers, resulting in a higher performance for data transfers. Fiber leverages `nanomsg` library to provide direct communication between processes. However, although Kubernetes clusters are able to

scale, their scalability is limited. For example, in a managed Kubernetes service, the scalability is managed by the cloud provider through their metrics observability services and auto-scale policies. An on-prem cluster deployed with, for example, Rancher, is normally fixed in size and it does not scale. Also, the time it takes for a VM to be provisioned and added to the Kubernetes worker pool can range from seconds up to a minute. In addition, the time it takes for a container to be created is usually several seconds, which is a much higher time than other services such as FaaS, where the time it takes to create a function is usually in the hundreds of milliseconds.

Listing 5 Fiber Monte-Carlo Pi Estimation example

```
from fiber import Pool
import random

NUM_SAMPLES = int(1e6)

def is_inside(p):
    x, y = random.random(), random.random()
    return x * x + y * y < 1


def main():
   pool = Pool(processes=4)
   pi = 4.0 * sum(pool.map(is_inside, range(0, NUM_SAMPLES))) / NUM_SAMPLES
    print("Pi is roughly {}".format(pi))


if __name__ == '__main__':
    main()
```

Listing 5 shows an example of Fiber to estimate Pi using the Monte-Carlo method[1]. As we can see, `fiber` module is similar to `multiprocessing` module, so we can use the same abstractions (in this case, `Pool`) of parallel processing for both local computation using multiprocessing, and distributed computation using Fiber, thus achieving access transparency.

2.3.2 Crucial

Another example of transparency at the application level is Crucial [9]. Crucial is a library that implements Java's threading interface which allows threads to be transparently executed as serverless functions. Unlike Fiber, which uses Kubernetes, Crucial leverages serverless functions to highly improve scalability thanks to the inherent elasticity of *FaaS*. Crucial also provides

[1]Example code can be found at https://github.com/uber/fiber/blob/master/examples/pi_estimation.py

stateful abstractions based on distributed shared objects that reside in a disaggregated in-memory layer (Infinispan). Unlike Fiber, adapting a parallel Java program using threads to Crucial requires additional manual work. Although the source code implementing the logic can remain unchanged, shared state accesses must be reviewed one by one and replaced by its Crucial equivalent. Fiber, on the other hand, by (partially) implementing the `multiprocessing` library interface, this manual process is not required. Later, in the insights section, we emphasize more on Python's parallel programming model and how it facilitates access transparency to disaggregated resources.

Listing 6 Crucial Monte-Carlo Pi Estimation example

```
public class PiEstimator implements Runnable{
        private final static long ITERATIONS = 100_000_000;
        private Random rand = new Random();
        @Shared(key="counter")
        crucial.AtomicLong counter = new crucial.AtomicLong(0);

        public void run(){
                long count = 0;
                double x, y;
                for (long i = 0L; i < ITERATIONS; i++) {
                        x = rand.nextDouble();
                        y = rand.nextDouble();
                        if (x * x + y * y <= 1.0) count++;
                }
                counter.addAndGet(count);
        }
}

List<Thread> threads = new ArrayList<>(N_THREADS);
for (int i = 0; i < N_THREADS; i++) {
        threads.add(new CloudThread(new PiEstimator()));
}
threads.forEach(Thread::start);
threads.forEach(Thread::join);
double output = 4.0 * counter.get() / (N_THREADS * ITERATIONS);
```

Listing 6 shows an example of Crucial to estimate Pi using the Monte-Carlo method. We can see how Crucial implements the threading Java interface, and how access to a shared atomic counter is implemented.

2.3.3 Faasm

Faasm [45] is a high-performance stateful serverless runtime that leverages WebAssembly to achieve multi-tenant isolation but allows functions to share regions of memory for low-latency

and concurrent data access.

Faasm presents access transparency since it exposes an interface that implements some POSIX syscalls or parallel frameworks APIs such as OpenMP and MPI. Faasm intercepts this interface and allows to transparently execute existing HPC applications over serverless compute resources without prior modifications to the code [35].

2.4 Comparing our research with the related work

Our work is based on Fiber's and Crucial's lines of research. On the one hand, we want to use Python's multiprocessing interface since, as we have seen with Fiber, its parallel programming model facilitates the implementation of shared-state abstractions and disaggregated processes. On the other hand, following Crucial model, we want to use *FaaS* for high scalability.

This research follows the work from [39]. In [39], the author also used Python multiprocessing over serverless functions to transparently scale local parallel applications. However, they did not do an exhaustive comparison with a single-machine system in order to find possible overheads. Moreover, they took applications related to stateless AI, while we propose to widen the range of possibilities and study general-purpose applications. Finally, they did not implement the whole multiprocessing interface, so not all applications will work without changing the code. That is, they studied **scale transparency** and we will study **access transparency**.

We propose another implementation of Python's multiprocessing library based on the Lithops framework, which is used to run jobs on serverless functions. In addition, we want to validate access transparency with current state-of-the-art technologies and Cloud services. Fiber's authors did not make an exhaustive comparison of its model compared to a local execution model. For Crucial, while the authors validate its model with a local machine, they does so for a synchronization problem and not for an application that requires shared state with data transfers.

Therefore, we are going to execute 4 applications (Evolution Strategies, OpenAI Baselines, Scikit-Learn Grid Search and Padaral·lel) on a single VM and over serverless functions and compare the speedup and overheads, always keeping the code unchanged to ensure access transparency. We anticipate that the overheads generated by shared state data transfers will suppose a bottleneck, since the latency and throughput of local memory is not comparable to disaggregated memory. However, we believe that highly parallel applications can take advantage of *FaaS* scalability so that these overheads can be compensated with the gains produced from speeding up the execution, resulting in a lower run time.

Chapter 3

Serverless way to transparency

In this chapter we will see the technical and design aspects that have been considered for the implementation of the multiprocessing library over Lithops for serverless functions and Redis. First, we will see an overview of what the Lithops framework is and how it works. Then we will see how the different Python multiprocessing parallel computing and shared-state abstractions have been adapted for serverless functions, object storage and Redis. Then, we review an optimization that has been implemented, the serverless worker pool, in order to mitigate possible overheads caused by serverless functions. Finally, we will discuss the implications of this design and implementation in relation to fault tolerance.

3.1 Lithops: Lightweight Optimized Serverless Processing

We have leveraged the Lithops framework to execute parallel local applications over disaggregated serverless functions.

Lithops framework [41] enables the execution of local serial code to be run over massively parallel serverless functions. Lithops acts as an abstraction layer that simplifies the exploitation of the main FaaS services present in public clouds for highly parallel tasks. One of Lithops design principles is to ensure portability between clouds. The same application can be seamlessly ported from one cloud provider to another, which prevents vendor lock-in.

We have extended Lithops with a `multiprocessing` module which implements in its entirety the original Python `multiprocessing` interface. Computation abstractions (like `Process` and `Pool`) use Lithops *FunctionExecutor* API. Inter-process Communication (IPC) and synchronization abstractions (like `Lock` and `Queue`) are implemented using Redis as a key-value in-memory database. In Section 3.2 we describe the implementation details of the IPC and

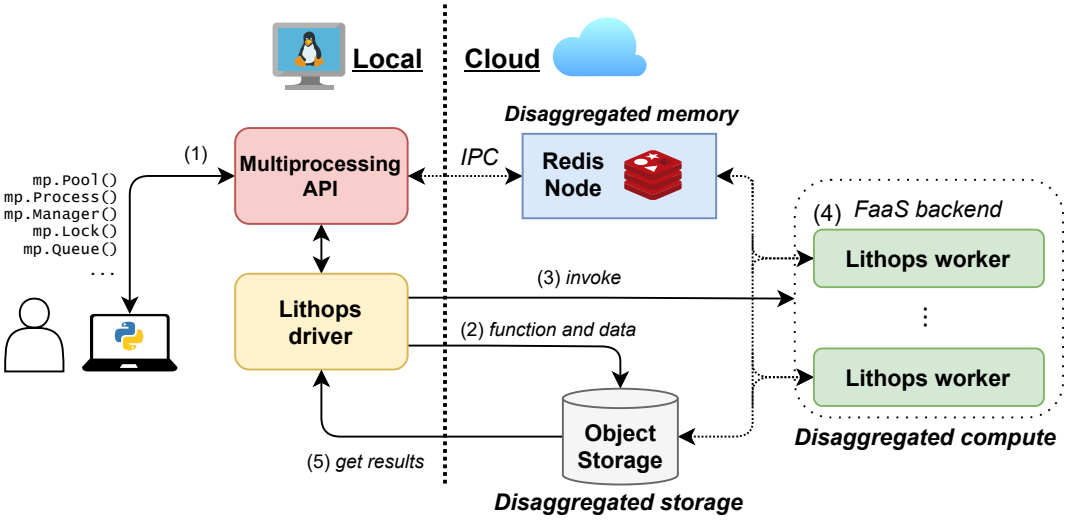

Figure 3.1: Lithops architecture

synchronization abstractions using Redis.

Lithops follows a master/worker architecture where a local process acts as the orchestrator and coordinator of the workers that are deployed and executed in the FaaS backend as serverless functions. A diagram of the general operation of Lithops is shown in Figure 3.1.

In broad strokes, the functionality of Lithops is as follows:

1. First, the user interacts with Lithops multiprocessing API, which is built atop Lithops framework.

2. Lithops automatically detects the processes' dependencies, and it serializes them together with the process function code and input arguments. The serialized data is then uploaded to a disaggregated storage service (like AWS S3).

3. Next, the Lithops orchestrator invokes the corresponding number of serverless functions against the FaaS backend. For example, every `Process` corresponds to a single function. The number of serverless functions for parallel `Pool` operations like `map` or `starmap` will be determined by the number of elements in the iterator being processed by the `map` operator, with a mapping of one function per element.

4. The Lithops worker is a generic serverless function that handles the execution of Lithops job tasks. It downloads the previously uploaded code, data and dependencies from storage, deserializes them and executes the user's function in a wrapper that handles errors. When the function has finished its execution, the result is uploaded back to storage.

5. The Lithops orchestrator provides two mechanisms for function join detection (6). The first is pull-based, where the driver lists the contents of the storage and detects that

a function has finished when the result key is listed. The second is push-based using an external messaging service like RabbitMQ. Upon completion, each function sends a termination event to a queue. The driver blocks consuming and aggregating the events in that queue until all have been received. Finally, the results are downloaded and returned to the parent application.

3.2 Multiprocessing abstractions

Since current FaaS offerings lack mechanisms for function addressing and direct communication, a serverful component acting as an *intermediary* is needed to keep the shared state and to mediate inter-process communications.

We have chosen Redis for its simplicity of deployment, in-memory storage and high performance. Redis differs from other traditional key-value databases in that the value has a type, such as `LIST`, `STRING`, `HASHSET`, etc. The different operations available on these data types facilitate the implementation of communication and synchronization abstractions present in Python's `multiprocessing` library. A single node Redis deployment guarantees consistency and the correct order of read and writes, since Redis is single-threaded, but fault tolerance is not provided. A distributed deployment of a Redis cluster can scale horizontally and fault tolerance is guaranteed thanks to the replicas, but consistency is dismissed in favor of availability. Fault tolerance is discussed in Section 3.5.

In Python's `multiprocessing`, processes need a reference to the objects that represent a shared state resource (such as queues or shared memory). The parent process creates all resources and then passes a reference to the child processes when they are forked. With Lithops, objects and references passed to functions have to be serializable. To maintain the same behavior, we have followed a pattern in which each resource object (`Queue`, `Pipe`...) acts as a proxy to the key-value pair in Redis, which is where the state resides. Each object is uniquely identified and corresponds to a specific Redis key-value pair.

The different abstractions available and a brief description of how they have been implemented are listed below.

- **Pipes**: Pipes are implemented using the `LIST` type. A `Pipe` is used for duplex communication between a pair of processes, and it's not meant to be shared with more than two. A `Pipe` instantiation returns a tuple of two `Connection` objects. Each `Connection` corresponds to a Redis `LIST`. Data can be written to the `Pipe` using `send()` on one of the `Connection`s, and it can be read using `recv()` on the other `Connection`. The `send()` method executes an `LPUSH` command to put data in the tail of the list, and the `recv()` method executes an `BLPOP` command to get data from the head list. This way, the list

is treated as a FIFO queue. The `BLPOP` gets and removes the head item of the list, or blocks until there is an element available.

- **Queues**: `Queues` are implemented the same way as Pipes, the difference being that more than two processes can put or remove items from the queue. Redis maintains the order of puts and gets consistent.

- **Array** and **Value**: `Array` and `Value` are used to share memory in Python multiprocessing. Only basic C type values can be put into the array or value. They are implemented using the `LIST` type. Processes can read and write to specific indexes or slices of the array. A `Value` is an `Array` of size 1. We have opted for using the `LIST` type instead of `STRING` because `STRING` values are limited to 512 MB in size, while in lists, each element of the list can will be at most `sizeof(long double)` in size, and lists can hold up to $2^{32} - 1$ elements.

- **Semaphore** and **Lock**: `Semaphore` and `Lock` are implemented using the `LIST` type. When the semaphore is created, N tokens are added to the list, where N is the initial value of the semaphore. Every *acquire()* of the semaphore will execute a `BLPOP` command, which removes a token from the list. The `release()` method puts the token again in the list with a `LPUSH` command. If there are no tokens when a `acquire()` is called (meaning that N processes are currently in the critical section), the `BLPOP` command will block until other process performs a `release()` and puts back a token into the list. Note that in this implementation, the value of the semaphore will always be greater than zero. A `Lock` is a generalization of a semaphore where N is 1.

- **Barrier**, **Event** and **Condition**: For `Barrier`, `Event` and `Condition`, each condition has a set of *notification lists* which are used to notify blocked process that are waiting for the condition event. When a process reaches the condition and hangs on the `wait()` method, the process registers a new list to the *notification list set* and blocks to that list with the `BLPOP` command. The process that satisfies the condition will add an element to each list that is in the *notification list set*, so that all waiting processes are unblocked and the execution resumes. `Barriers` and `Events` are specific cases of `Conditions`. For example, the condition of a `Barrier` is that all processes arrive to the `wait()` method.

- **Managers**: `Managers` allows creating Python resources in a separate process, and they are accessed via sockets. Managers are used to share a basic Python data type with multiple processes, such as a `dict` or `list`. The implementation of those types is trivial using Redis, since it provides `HASHSET` and `LIST` types natively. A `Manager` also permits the creation of complex resources, for example, a custom class, which resides instantiated in

the Manager and other processes use Remote Method Invocation to access it. To provide a similar behavior using Redis, we have made each process have a local instance of the `Manager` class, but the state of the instance (i.e. its variables) is stored in Redis as simple key-value pairs. Each time a process accesses the `Manager` instance, the instance state is downloaded from Redis and the local attributes are updated with the newly downloaded ones. Then, the method is invoked using the local attributes. Finally, after the method is invoked, the attributes are sent back to Redis. A `Lock` ensures that attributes are accessed by only one process at a time.

3.3 Storage abstractions

Lithops also implements a replica of Python's built-in `open` function and the `os.path` module which allows to transparently read and write files and directories stored on a disaggregated storage service (like S3) as if it were a local file system. This is especially useful for *FaaS* since the volume that is mounted in the function container is volatile and the data stored there is lost when the execution finishes. In this way, we offer serverless processes a transparent way to save or recover their state. However, it should be noted that, since we are working on immutable data, it is not possible to modify or expand a file as would be done in a traditional network file system without having to rewrite the entire file, which can be problematic for large files. However, as seen in Section 4.2.4, disaggregated storage services provide much higher parallel read and write throughput than traditional disks used in monolithic machines. Applications that require reading lots of data in parallel (for example, video encoding) can benefit from disaggregated storage to achieve lower execution times.

3.4 Serverless Worker Pool

Python uses processes to achieve parallelism and to overcome the Global Interpreter Lock (GIL), which prevents threads from executing in parallel on multi-CPU machines. In both local and serverless, creating processes is expensive. Especially when we use Lithops to launch processes, if the granularity of the tasks is very small, the overhead of invoking many functions can be prohibitively expensive. For this reason, Python multiprocessing implements the `Pool` abstraction. A `Pool` represents a fixed-size pool of processes that are created at the time of the pool instantiation. It has methods like `starmap()`, `apply_async()` or `map_async()` which allows tasks to be offloaded to the worker processes at a higher level, instead of creating and managing `Process` objects manually. In a process pool, each operation (*map*, *apply_async*...) creates one or more jobs. These jobs are queued to a *job queue*. The worker processes get and

execute jobs from the queue. This avoids the need to create new processes for each task, which considerably reduces the fork overhead. In addition, it is useful to initialize global worker-scope variables, as they only need to be initialized once at worker process creation.

To support a similar functionality with Lithops with serverless functions, we have implemented the job queue pattern for the Lithops multiprocessing `Pool`. In Lithops multiprocessing, workers are long-lived functions that are invoked when the `Pool` object is created. Operations on the pool (`map()`, `apply_async()`) generate Lithops tasks, but instead of invoking new functions to execute those tasks, they are queued in a Redis list. The worker functions pick up and execute tasks from the queue as they are generated. Once the Pool is closed (`terminate()`), a message is sent to the workers to terminate their execution.

The main advantage of this implementation is that the overhead of submitting a set of tasks to a Redis list is much lower than invoking a function for every task. With Redis, we can submit all tasks at once with a single `LPUSH` command, while invoking functions is sequential and the overhead depends on the API and architecture of each FaaS service. Also, reusing functions to execute multiple tasks avoids stragglers caused by cold invocations. However, the main drawback is the function execution time limit. Although the time limit has been increasing over the years (for example, AWS Lambda now supports invocations of up to 15 minutes), this limit prevents running longer executions.

3.5 Fault tolerance

The fault tolerance of our solution is based on the assumption that the underlying disaggregated resources are fault-tolerant. When programming a monolithic local system, fault tolerance is not taken into account because local resources do not fail (or at least, the way we expect distributed systems to fail). When we move to a distributed environment, if the disaggregated resources (compute, memory and storage) mask the possible failures that may occur, then the application programmer can also assume that they will not fail, and we can continue with the same programming model that does not contemplate error handling and rely on the same local programming model.

On the one hand, we have the computation layer provided by Lithops and the *FaaS* provider. Lithops is able to detect failures at the time of function invocation, and can and re-invoke failed invocations. It follows an *all-or-nothing* model, so if a function of a *map* job fails, the whole *map* job is assumed as failed and the operation is aborted. In addition, Lithops is able to detect failures occurring within the function container, such as user code failures (Python exceptions) or when the memory limit is surpassed. In these cases, the error is returned to the Lithops driver process and the user can decide whether to abort the operation or keep only the result of

successful invocations. The Lithops driver process is also able to detect functions that time-out. However, if the Lithops driver process fails (e.g., the process is killed by the user), there is no way to recover the invoked functions. Lithops, in turn, assumes that the *FaaS* service is also fault tolerant. For example, an invocation of a function might be submitted correctly but a failure in the service prevents allocating resources for it. In these cases, AWS Lambda retries the invocation up to three times. AWS Lambda SLA foresees that the service must be available 99.95% of the time before it is considered that it failed, so we can assume that AWS Lambda rarely fails.

On the other hand, we have the shared state and communication layer provided by a disaggregated in-memory layer. For the time being, there is no offering of fault-tolerant disaggregated in-memory storage on the public Clouds. As for the compute layer, the disaggregated memory service should also mask failures and provide fault tolerance transparently. For our solution, we use a Redis instance running in `IaaS` (EC2) as a temporary workaround. In this article, we do not discuss the fault tolerance of the in-memory layer, as it is out of the scope of this research. We refer here to existing research works offering fault-tolerant in-memory services like Anna Key Value Store[46], Crucial replicated in-memory layer [9] or CloudFlare Workers KV commercial service.

Chapter 4

Validation

In this chapter, we evaluate the feasibility of serverless access transparency with state-of-the-art technologies and services. In particular, we want to validate that:

- Access transparency can be obtained in some *multiprocessing* Python-based applications.
- It is possible to scale an application beyond the limits of a VM using serverless technology without hardly modifying its code.
- We can simulate the vertical scaling of a VM using serverless technology.

First, in Section 4.1 we list the configuration of the used Cloud resources for the validation. In Section 4.2 we measure specific overheads and performance of the disaggregated serverless compute and storage resources using Lithops. This will indicate the magnitude of the main overheads of the system to understand its behavior in more complex scenarios. In Section 4.3 we perform experiments to evaluate access transparency with real complex programs that use the *multiprocessing* library and we compare Lithops performance and scalability with a VM. Finally, in Section 4.4 we present a summary of the results obtained in the experiments.

4.1 Configuration and parameters

Unless otherwise stated, the experiments have been run with the following settings: Lithops using AWS Lambda as compute backend with a Redis services as storage backend. Lithops host runs on a m5.2xlarge EC2 instance with Ubuntu 20.04, Lambdas use a containerized Python 3.8 runtime with 1792 MB of RAM [1] as Serverless function and Redis 6.2 instance runs on the host machine with Docker.

[1] According to AWS documentation, it is equivalent to a whole vCPU.

The Lithops host machine and the AWS Lambdas are in the same VPC private subnet, region and availability zone (us-east-1 A), so traffic does not go through a NAT gateway nor the public internet. In case of using S3 as storage backend (will be stated), the S3 bucket is located in the same region and access to S3 is done via a private endpoint. All Lambda functions have been executed using warm containers.

We have used on-demand EC2 instances with different number of vCPUs for the local monolithic executions to compare them with Lithops. In particular, we have used the following EC2 instances:

- c5.4xlarge with 16 vCPUs
- c5.9xlarge with 32 vCPUs
- c5.18xlarge with 64 vCPUs
- c5.24xlarge with 96 vCPUs

Note that, by default, c5.9xlarge and c5.18xlarge have different numbers of vCPUs[2], these have been changed using the EC2 virtualization settings. All of these c5 EC2 instances are running Ubuntu 20.04 and located in the region us-east-1.

4.2 Micro-benchmarks

The objective of this section is to evaluate the system with micro-benchmarks that allow us to identify potential overheads and where are they generated, which will help us understand the validation results of the real applications.

4.2.1 Fork-join overhead

One of the main overheads when doing parallel computing is the cost of creating a new thread or process. The purpose of this experiment is to measure the overheads generated when invoking multiple parallel serverless processes and analyze how they scale. Lithops allows the usage of different storage backends and monitoring systems for serverless functions. In this experiment, we compare the performance between using S3 or Redis as storage backend and also the usage of RabbitMQ for task monitoring. The difference between using RabbitMQ or Redis/S3 for monitoring functions is the method of join detection. When using Redis or S3, Lithops lists the storage keys to see if the functions have uploaded their status objects. With RabbitMQ, the functions send their status to the driver through queues and messages. In this experiment,

[2] c5.9xlarge has 36 vCPUs and c5.18xlarge has 72.

when using RabbitMQ as task monitoring, we used an instance of RabbitMQ 3.8 running in the Lithops host machine via Docker.

Listing 7 contains the main code of this micro-benchmarks. As it can be seen, the experiment consists of performing a multiprocessing Pool *map* of several sleep functions (line 17), each one would sleep for 5 seconds. The overhead time is calculated by subtracting the sleep time from the total execution time (line 20).

Listing 7 Main fork-join micro-benchmark code.

```
1  import time
2
3  def worker_function(sleep_seconds):
4      time.sleep(sleep_seconds)
5
6  args = get_args()
7
8  # Import the selected multiprocessing library
9  if args.backend == 'mp':
10     import multiprocessing as mp
11 elif args.backend == 'lithops':
12     import lithops.multiprocessing as mp
13
14 # Perform the experiment
15 t1 = time.time()
16 p = mp.Pool(processes=args.workers)
17 map_result = p.map(worker_function, [args.sleep] * args.workers)
18 t2 = time.time()
19
20 print((t2 - t1) - args.sleep)
```

Figure 4.1 represents the overall overheads for different parallel executions. We see that the scenarios that use Redis have the best scalability and a lower overhead. Redis is designed specifically to support fast lookups and to store small objects in memory, mainly focused on caching. Unlike Redis, S3 is designed to store large amounts of data on disk and provide a high throughput of concurrent reads, which would explain the higher overhead. We can see that the overhead using only Redis compared to using Redis & RabbitMQ is slightly better. This is because RabbitMQ consumes the messages sequentially and Redis can list all the keys at once, so although doing polling from the Lithops host to the Redis server is less efficient, it is faster than joining the RabbitMQ messages sequentially.

We also have measured where these overheads are generated by Lithops and AWS Lambda invocation.

Figure 4.2 represents the histogram of the map job of 5 sleep seconds for 1024 parallel functions using Redis as storage backend. The chart on the left represents an execution using

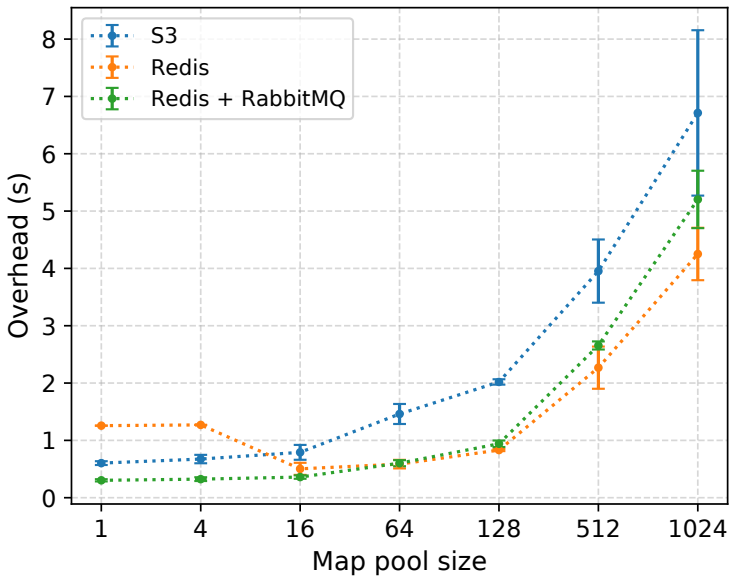

Figure 4.1: Processes pool invocation results.

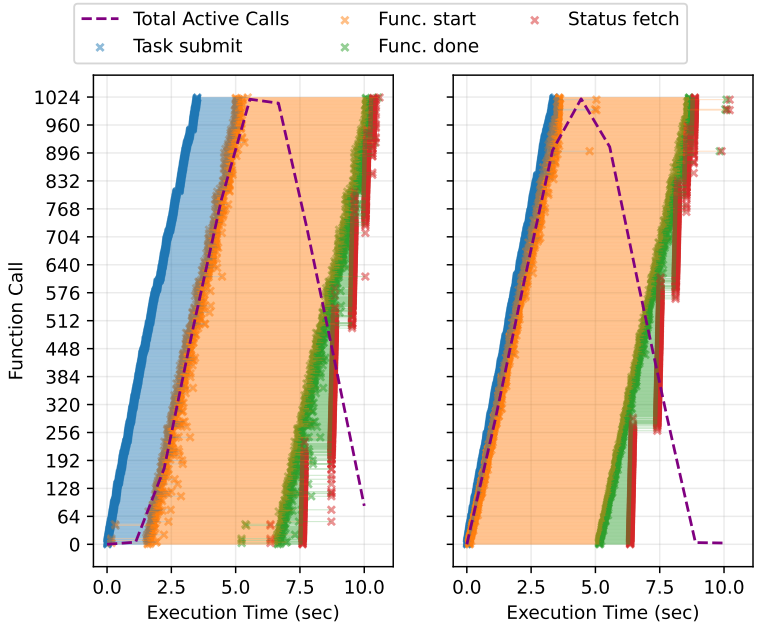

Figure 4.2: Histogram of the `sleep(5)` *map* job. (left) Cold invocation. (right) Warm invocation.

cold containers while the chart on the right represents another execution using warm containers. We can see the difference in the function start-up overhead. When using cold containers, the overhead is higher because the provider has to allocate resources to run the functions, while when using warm containers, the resources are already allocated and the container is already

	Invocation type	
Phase	Cold	Warm
Serialize data and function	0.004 s	0.004 s
Upload dependencies	0.002 s	0.001 s
Invoke	1.719 s	0.258 s
Function setup	0.052 s	0.046 s
Join	0.628 s	0.630 s
Total	**2.407 s**	**0.939 s**

Table 4.1: Decomposition of generated overheads. Average values across all functions of a map job.

up and running, so the overhead is much lower. Specifically, warm invocations typically have an overhead of around 200 ms, while cold invocations can have a more variable and higher overhead. In this case, the overhead of cold invocations exceeds the second and the average overhead obtained is 1.7 s. We can also see that the start of execution is not instantaneous but linear. This is because the asynchronous invocation of all functions of a *map* job is performed sequentially in a loop. This implies that the greater the number of functions, the greater the invocation overhead. It also implies that full parallelism is not achieved immediately. Applications that require exact parallel execution should use some synchronization mechanism (e.g. a barrier).

Table 4.1 shows the decomposition of the overhead introduced by Lithops and by AWS Lambda. The values indicate the average times of all the functions of the same *map* job. We have differentiated two executions, one using warm containers and the other using cold containers. "*Serialize data and function*" indicates the time spent to serialize the function and its dependencies. We can see that they are constant since both use the same function and dependencies. The "*Invoke*" column indicates the time elapsed since the function is invoked until the container begins its execution. We can see that it is longer for executions using cold containers for the reasons exposed above. The "*Function setup*" time indicates the time elapsed from the start of the container until the user function starts executing (i.e. the setup time of the Lithops wrapper). The "*Join*" time indicates the time elapsed between the end of the function and its detection by the Lithops driver. Finally, the sum of all overheads is shown in the "*Total*" column.

The overhead time determines the minimum granularity of the process since processes with lower granularity than the overhead will not benefit from distributed serverless execution. Moreover, the closer the granularity is to the overhead time, the more noticeable it will be with respect to the total application execution time.

4.2.2 Network latency and throughput

In a distributed system, communication overheads tend to determine and limit the performance of the whole system. This experiment aims to measure latencies and throughput between processes via `Pipe`.

To make the latency measurements we send a single *char* over a `Pipe` between two processes and measure the round-trip time. Each experiment sends a total of 10000 packets, of which, we have measured the mean, maximum and minimum latencies. Listing 8 contains the main parts of the code. It can be seen that we create a `Pipe` (line 25) that is passed as argument to a new `Process` that executes the `pinging` function (line 28). This process will send each packet to the other process and will wait for a response (line 11 to 16). The other process executes the `pinged` function (line 26), which waits for each packet and returns the same packet (line 22).

In this experiment, we have used an AWS m5.large machine as Lithops host. For local executions, another m5.large VM has been used.

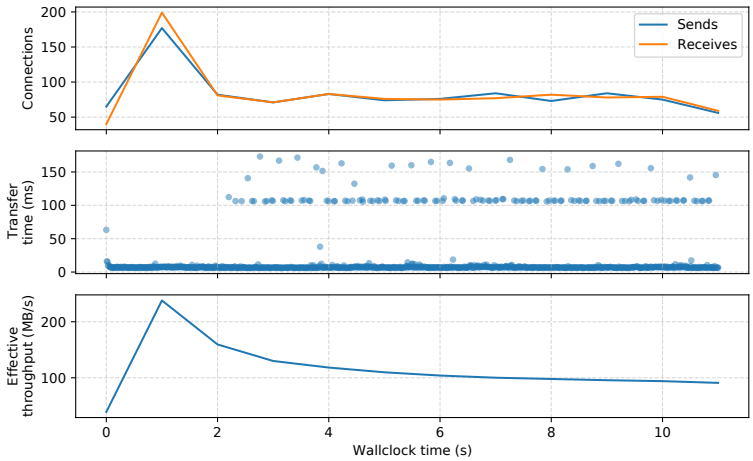

Figure 4.3: Latency results.

Pipe type	*Mean*	*Max*	*Min*
Local	29,7 ns	0.3 ms	22.8 ns
Between EC2 instances	0.5 ms	1446 ms	0.2 ms
AWS Lambda (Redis)	1 ms	149 ms	0.7 ms

Table 4.2: Table of latencies.

Results on latency are arranged in Table 4.2. We see that the latency of a Local `Pipe` is far from its distributed implementation, so the performance is not comparable. The latency for a `Pipe` between EC2 instances uses direct communication, so the latency is much lower compared to the Redis implementation.

Listing 8 Latency measurements code.

```
from lithops.multiprocessing import Process, Pipe

def pinging(pinged_conn, results_conn, packets, warmup_packets=0, packet_size=1):
    times = []
    data = '0' * packet_size

    for i in range(warmup_packets):
        pinged_conn.send(data)
        pinged_conn.recv()

    for i in range(packets):
        t1 = time.time()
        pinged_conn.send(data)
        pinged_conn.recv()
        t2 = time.time()
        times.append(t2-t1)

    results_conn.send(times)

def pinged(pinging_conn, packets, warmup_packets=0):
    for i in range(packets + warmup_packets):
        pinging_conn.send(pinging_conn.recv())

# Perform measurements
conn1, conn2 = Pipe()
p = Process(target=pinging, args=(conn1, conn1, packets, warmup_packets, packet_size))
p.start()
pinged(conn2, packets, warmup_packets)
p.join()

results = conn2.recv()

conn1.close()
conn2.close()
```

Similar to the latencies measurements, to compute the throughput we used a `Pipe` to communicate two processes, through which we send 1000 messages with a size of 1MB (a total of 1 GB). Listing 9 contains the main code of this experiment. As it can be seen, the sender process waits until the receiver is up (lines 3-4 and lines 21-23). Later the sender starts sending the packets (line 9 to 12) iteratively and measures the elapsed time.

We can see in Figure 4.3 that, for a single `Pipe`, Redis is capable of handling 150 connections per second (75 for writing and 75 for reading). The time elapsed for sending a message is stable at 15 ms, although we can observe that some connections take longer, which could be caused by the fact that we use shared network resources on both the Redis node and in the Lambda host machines. The total transmission takes around 10.5 seconds, so the effective throughput

Listing 9 Throughput measurements code.

```
def sender(pipe, dict_result, batches, batch_size):
    print('Wait until receiver is up', flush=True)
    pipe.send('Are you ready?')
    res = pipe.recv()
    print(res)

    data = os.urandom(batch_size)
    print('Start sending...')
    t0 = time.time()
    for i in range(batches):
        pipe.send(data)
    t1 = time.time()

    elapsed = t1 - t0
    dict_result['sender'] = elapsed
    print(f'Sender elapsed: {t1 - t0}', flush=True)


def receiver(pipe, dict_result, batches, batch_size):
    print('Start receiver', flush=True)
    msg = pipe.recv()
    print(msg)
    pipe.send('Yes!')

    print('Start receiving...')
    t0 = time.time()
    for _ in range(batches):
        data = pipe.recv()
    t1 = time.time()

    elapsed = t1 - t0
    dict_result['receiver'] = elapsed
    print(f'Receiver elapsed: {t1 - t0}', flush=True)
```

rate is around 90 MB/s.

4.2.3 Computational performance

The goal of this experiment is to measure computational performance in an embarrassingly parallel example and to compare the execution time and scalability between a large VM and a serverless infrastructure with Lithops.

To carry out the experiment, the classic example of the calculation of π with the Monte Carlo method has been used. In particular, this test is based on sampling 3,200,000,000 random points and calculating the number of points that are within the unit circle to extract an approximation of π. The amount of points to sample is distributed between all the processes, so execution

time should decrease when increasing the number of processes. Listing 10 contains the code for this experiment. As can be seen in the code, we perform a `Pool.map` of the `in_circle` function (line 30). Each invocation of the `in_circle` receives the number of points to sample (line 28) and the random seed. Each function samples its points (line 11), counts how many are inside the unit circle (lines 12-13) and returns this value. The number π is approximated using the results of the map (line 33).

Listing 10 Code of π Monte Carlo experiment.

```
1  import time
2  import random
3  import math
4  import argparse
5
6  def in_circle(args):
7      batch_size, seed = args
8      random.seed(seed)
9      s = 0
10     for _ in range(batch_size):
11         x, y = random.random(), random.random()
12         radius = math.sqrt(x**2 + y**2)
13         s += radius <= 1
14     return s
15
16 args = get_args()
17
18 # Import the selected multiprocessing library
19 if args.backend == 'mp':
20     import multiprocessing as mp
21 elif args.backend == 'lithops':
22     import lithops.multiprocessing as mp
23
24 # Perform experiment
25 batch_size = args.sample_points // args.workers
26 pool = mp.Pool(args.workers)
27
28 batches = [[batch_size, i] for i in range(args.workers)]
29 t1 = time.time()
30 results = pool.map(in_circle, batches)
31 t2 = time.time()
32
33 pi = (4.0 * sum(results) / (args.workers * batch_size) )
34 print(f'Map time: {t2-t1}')
35 print(f'Pi = {pi}')
```

The results of Figure 4.4 show that the scalability capacity that can be obtained with Lithops using FaaS goes much further than what a single machine could achieve, despite the fact that the executed code is exactly the same. It can also be observed that the performance

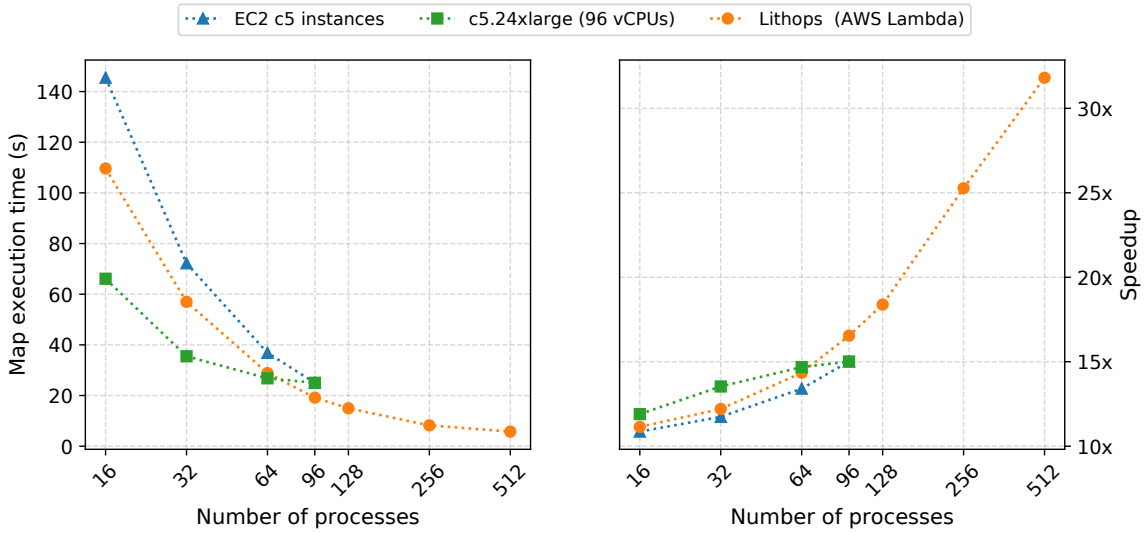

Figure 4.4: π Monte Carlo results.

of the disaggregated system is, between 20% and 25%, superior when, a priori, it could be expected to be inferior due to invocation and communication overheads. The performance of the disaggregated system is better than VMs' one because, in a VM, the processes share physical CPU cores between them thanks to the use of Hyper-Threading technology, but despite this, the number of floating-point operations is still limited by the physical CPU cores used. On the distributed system using FaaS, functions are less likely to share physical CPU cores between them and chances are to do so with functions with little computational demand. So, existing serverless overheads were partially masked by Hyper-Threading inefficiencies observed in a single VM.

4.2.4 Disk performance

The objective of this experiment is to measure the disk read and write capacity for Lithops' processes running on FaaS, transparently emulating the disk of a machine even though the program runs on a distributed system. Another aspect to measure is the scalability of these read and write rates depending on how many processes are used.

The experiment has two phases, one where writes to disk are performed and another where what has been written to disk is read. In the first phase, a certain number of processes are launched, each of these will write a file with 1GB of data to disk. Then, in the second phase, we relaunch a number of processes, each of which will read from disk what was written in the previous phase. The experiment is performed for different numbers of processes to study scalability. Peak read and write rate measurements are taken. Here the storage backend has been changed to S3.

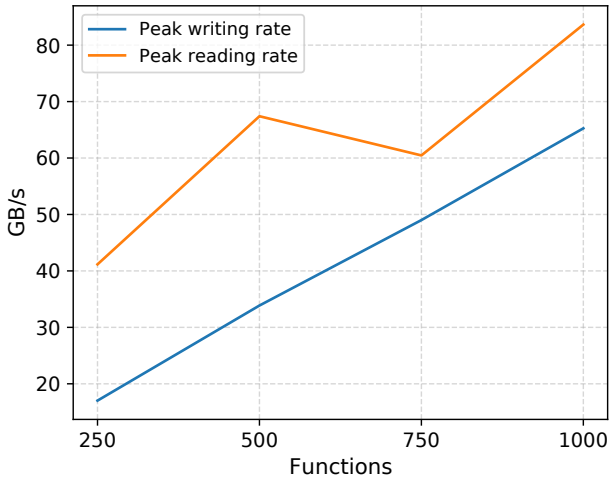

Figure 4.5: Disc reading and writing rates.

The results of the experiment show great scalability in disk read and write operations that are much higher than the read and write rates that can be achieved on a local system. As can be seen in the Figure 4.5, these rates reach peaks of read speeds higher than 80 GB/s and 65 GB/s for writes. This is useful for applications that require reading data from object storage in parallel, since with FaaS, we can achieve higher throughput and thus a lower execution time.

4.2.5 Shared memory performance

We also want to validate the performance loss when using remote shared memory. For this experiment, we have implemented a parallel sorting algorithm using shared memory and compared local and serverless execution. The sorting algorithm consists of splitting in chunks an array, performing quick sort on each chunk in parallel, and then recursively merge pairs of sorted chunks following a tree pattern. This algorithm makes heavy usage of the shared array as it is iterated over multiple times and the list items are constantly changing positions. We have implemented three alternatives to this algorithm. The first uses multiprocessing's `RawArray` to store the array and the operations are performed in-place, directly accessing the array indexes. Although the array is stored in shared memory, each process only accesses its corresponding chunk, so there is no need to provide mutual exclusion and critical sections. The second alternative also uses a shared multiprocessing `RawArray`, but each process copies its chunk to a local variable, performs the operations on this local array and then copies back the array to the shared memory array. The third implementation is completely decentralized and does not use a shared array. Instead, the parent process sends the chunks to the worker processes using pipes, and the workers perform the merge phase also passing their chunks using pipes between

them.

We have implemented these three alternatives to show that, although all three are correct for local execution, the way memory is accessed has a huge impact on the performance when using disaggregated memory.

Array size	5 M		10 M	
Implementation	*Local*	*Lithops*	*Local*	*Lithops*
Shared array	23.66 s	-	79.68 s	-
Local copy	15.693 s	356.60 s	47.11 s	-
Pipes	14.27 s	17.30 s	45.16 s	45.63 s

Table 4.3: Execution time of the different parallel quick sort implementations with different array size.

Table 4.3 shows the execution times of the three alternative implementations using different array sizes (5 M and 10 M) on both local and serverless using 64 processes. For the local execution, we have used a c5 EC2 instance with 64 vCPUs. We can see that Lithops was not able to execute the algorithm using the in-place shared array implementation. This is because each access to a list index is equivalent to a put or get on Redis of the element of that index. This causes a prohibitive overhead that prevents obtaining competent results using this shared memory method. The local copy implementation is presented as a low-effort improvement over the in-place shared array implementation. Still, Lithops struggles to perform due to the high data transmission overhead, since local memory is not comparable to remote memory. Finally, the decentralized alternative using Pipes is presented as the proper way to implement this algorithm using disaggregated memory. Since shared memory is no longer used, Lithops is now capable of providing competent performance compared to local execution. For the 10 M array size execution, we can see that both local and serverless executions result in the same execution time. We also can see an improvement in the local execution compared to the shared memory alternative. This tells us that, even if we have fast-access local shared memory, it is sometimes not the best alternative even in local executions.

4.3 Applications

In this section, we evaluate the behavior of Lithops in four different real use cases in order to test access transparency and to measure scalability and performance. The scenarios used are: the implementation of the OpenAI's Proximal Policy Optimization (PPO) algorithm in its Baselines repository, the POET modifications in Evolution Strategies made by the Uber research team, parallel Pandas' DataFrame transformations using Pandaral·lel, and a hyperparameter tuning

using Scikit-learn's Gridsearch running over Joblib. To adapt these applications to serverless, we only had to replace the `multiprocessing` import with `Lithops.multiprocessing`. Since Lithops fully implements the multiprocessing interface, the rest of the code did not need any further modification.

Application	*Algorithm type*	*Shared state*	*Message passing*	*Parallel map*
Evolution Strategies	Iterative pool map	✓	✗	✓
Pandaral·lel	Scatter Gather	✗	✓	✓
Gridsearch	Broadcast Gather	✗	✗	✓
PPO	Master Worker	✗	✓	✗

Table 4.4: Applications used and their algorithms.

Table 4.4 describes the applications and the type of algorithm used in them. We also specify what kind of stateful abstractions are used. In each section, we go into detail about how each algorithm handles the shared state and message passing.

We have taken the PPO and Evolution Strategies applications from the Fiber [52] validation since they are real and complex applications that use the multiprocessing library. However, we consider that the comparison with Fiber is not necessary. Although both Fiber and Lithops use the multiprocessing API, Fiber is based on Kubernetes, which is not serverless and processes can have direct communication. Conversely, we use serverless disaggregated services, so the comparisons would not be fair. Moreover, they do not indicate enough parameters to replicate their experiments. Finally, they compared themselves with other distributed computing frameworks, while we focused on finding the differences between local and distributed execution and exploit the high scalability of serverless functions.

Note that the use of these experiments helps us to have complex scenarios in which to check if access transparency can be achieved, in no case it is intended to study the results of the experiments in the fields of artificial intelligence, machine learning or data analysis. The results show that it is possible to obtain access transparency in real and complex use cases despite the existing overheads seen in the previous section.

4.3.1 Evolution strategies

In this experiment we have used the Paired Open-Ended Trailblazer (POET) [49] implementation, which is a large Python application with about 4000 lines of code using different *multiprocessing* abstractions like `Pool` or `Manager.dict`. This algorithm is part of the Evolution

Strategies category, in which, evolutions of an initial population are carried out iteratively, and those evolutions are executed in parallel. The objective of this test is to analyze and compare the performance and scalability of Lithops in an iterative algorithm that maintains and uses a shared state between processes.

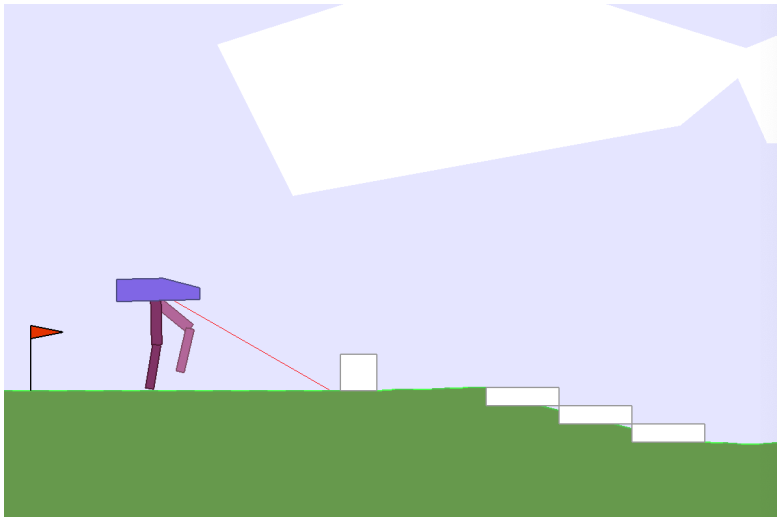

Figure 4.6: Bipedal walker environment visualization.

In this experiment, POET is used to train a model capable of overcoming the bipedal walker scenarios from the OpenAI GYM [12]. Figure 4.6 shows an example of a bipedal walker environment and Figure 4.7 shows the POET algorithm flow. POET algorithm consists of generating a population of one-to-one paired environments and agents[3] (1). In each iteration, it tries to optimize (2) the agents so that they are able to solve their paired environment, and it also tries to evolve the population of environments to obtain more diversity and complexity (2). As in any Evolution Strategies algorithm, at the end of each iteration, both the population of environments and agents are evaluated (3). The best ones become candidates to be reproduced or pass directly to the next epoch (4). At the end of the iterative process, a population of agents capable of solving the problem of the bipedal walker is obtained. Note that the behavior of each bipedal walker is determined by the parameters of its policy. In the POET algorithm, the `Pool` workers update these parameters and share them with the rest of the workers through a `Manager.dict` in each iteration. This means that exists a share state between the pool workers that needs to be consistent.

POET uses a shared noise table which is used to generate randomness in the evolution process. This noise table is originally implemented using shared memory. However, it is initialized when the module is loaded, so it is not using Lithops multiprocessing implementation

[3]In the machine learning field, an agent is an autonomous entity which acts, directing its activity towards achieving goals.

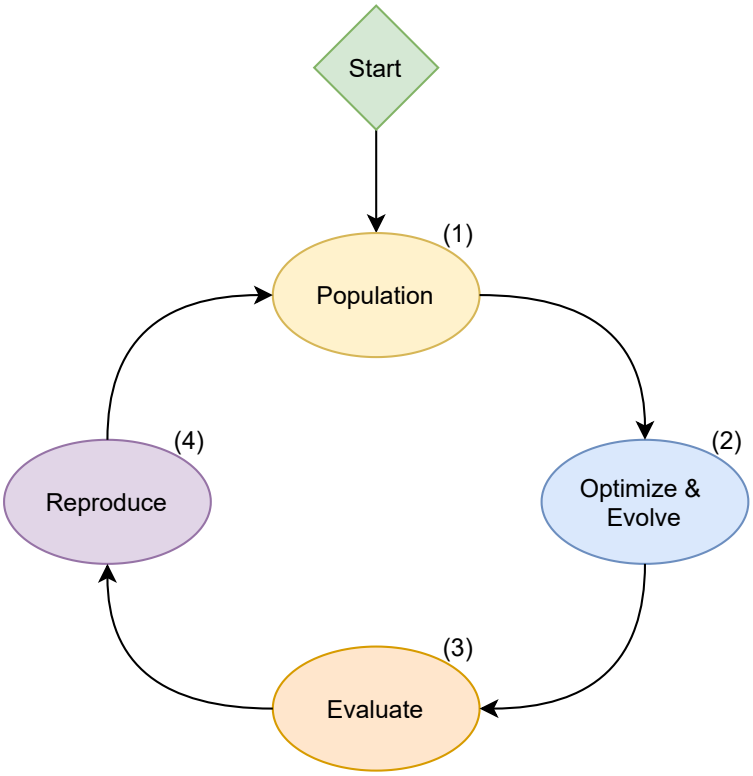

Figure 4.7: POET algorithm flow.

for shared memory. Instead, since this table is read-only, each function can initialize its noise table independently of the shared memory. This modification has been made in the code to run the program with Lithops efficiently and without problems. This modification implied a change in just two lines of code, changing this piece of code of Listing 11 by the similar piece of code of Listing 12

Listing 11 POET original noise table.

```
1     self._shared_mem = multiprocessing.Array(ctypes.c_float, count)
2     self.noise = np.ctypeslib.as_array(self._shared_mem.get_obj())
```

Listing 12 POET modified noise table.

```
1     generator = np.random.Generator(np.random.PCG64(seed))
2     self.noise = generator.random(size=count, dtype=np.float32)
```

The algorithm also uses a shared table of parameters that are modified in each iteration. This shared data structure is implemented as a shared multiprocessing `Manager.dict()` dictionary, and it has not been changed. Therefore, there is a certain transmission of data that

could imply a significant overhead. Figure 4.8 shows the resultant architecture for the POET when it is run with Lithops.

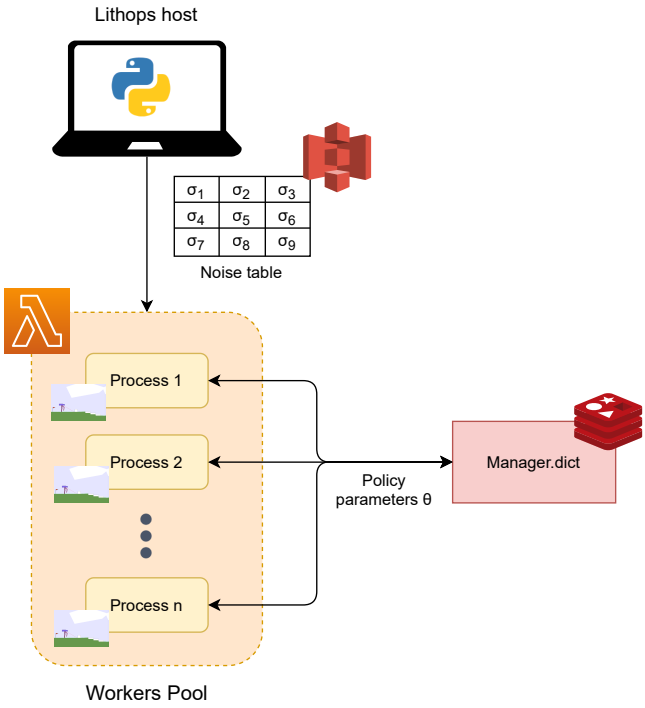

Figure 4.8: POET architecture.

The rest of the application code does not need any further modification for correct execution. But, for efficiency reasons and in order to make fair comparisons, the way in which tasks are sent to the Pool of workers with the Pool.apply_async method in each iteration has been modified. This is because, in Lithops, it is more efficient to make a single map than to do an iteration with several apply_async. The change has been very small and simple, in particular the code from Listing 13 has been replaced by the two sentences of Listing 14.

Listing 13 POET original task invocation.

```
1 for i in range(batches_per_chunk):
2     chunk_tasks.append(
3         pool.apply_async(runner, args=(self.iteration,
4             self.optim_id, batch_size, rs_seeds[i])+args))
```

Note that each iteration of the Evolution Strategies performs a `Pool.starmap_async()` operation. The task granularity for our parameter settings is small, about 3 seconds. To try to mitigate overheads, we used the implementation of the pool with job queue to reuse containers and reduce invocation overhead. Also, the noise table is only initialized once per worker, and it

Listing 14 POET modified task invocation.

```
map_args = [
    (self.iteration, self.optim_id, batch_size, rs_seeds[i]) + args
    for i in range(batches_per_chunk)
]

chunk_tasks = pool.starmap_async(runner, map_args)
```

can be reused throughout the execution of the program, so the overhead of creating it in each `Pool.starmap_async()` is prevented.

To carry out the measurements, we have executed the application with the following parameters: 5 iterations with 512 batches per chunk and a batch size of 5. All local executions have been run on a `c5.24xlarge` EC2 instance.

The results in Figure 4.9 show that, despite the data transmission and invocation overheads, Lithops maintains constant scalability similar to the scalability of the VM. The maximum speedup of the VM is about $40x$, while Lithops is capable of reaching a speedup of around $53x$, improving the best result of the VM by a 32.5%.

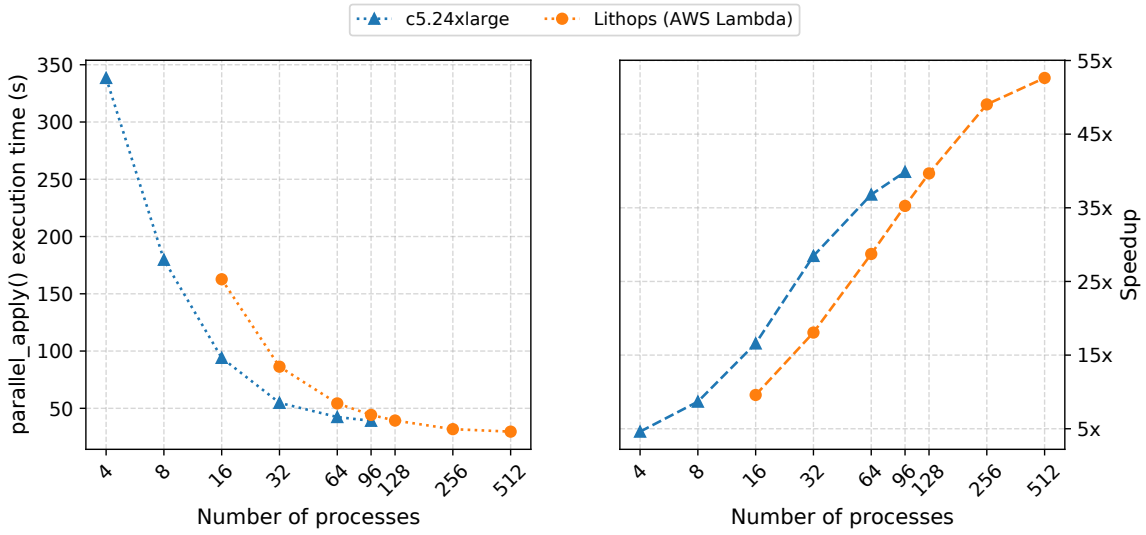

Figure 4.9: Evolution Strategies execution results.

4.3.2 Pandaral·lel

Pandas is a Python package, with more than half-million lines of code, that provides data structures to work with relational data. Pandas is designed to be able to work with large datasets, it allows to perform in-memory data analysis or data transformations. Some data

transformation functions from Pandas are limited to sequential execution and do not have an available parallel implementation. Pandaral·lel [38] is a Python module with about 1700 lines of code that extends the Pandas functionality adding parallel transformation functions. To do so, Pandaral·lel relies on the `multiprocessing` API. This experiment aims to measure Lithops' behavior in an embarrassingly parallel task with relatively large data transmissions and analyze how it handles that overhead.

Pandas is based in two main realational data structures: Series (1-dimensional) and DataFrames (2-dimensional). These data structures have a group of sequential methods that can transform them, such as *apply*, *map* or *applymap*. Pandaral·lel adds the methods *parallel_apply*, *parallel_map* and *parallel_applymap* to DataFrames and Series objects. When these methods are used on a Pandas data structure, Pandaral·lel first partitions the DataFrame or Series according to the number of available workers on the *multiprocessing*'s Pool. Then, it serializes the content of each partition and the function used. Tasks are then invoked (using `Pool.map_async`) to perform the data chunk computation. When all tasks are finished, the result is returned to the parent process using a `Queue`.

As said before, Pandaral·lel relies on the `multiprocessing` API, so replacing it by the `Lithops.multiprocessing` API allows us to execute it in a distributed environment. Pandaral·lel relies on the *dill* library for data and function serialization. When the function to apply to a dataset has external dependencies, *dill* does not serialize them with the function correctly. To solve that problem, it has been enough to change the serialization library by *cloudpickle*, that implements the same API. Lithops automatically detects the values (DataFrames or Series partitions) passed by parameter of each function and transfers them to the storage. Each FaaS downloads its chunk of data, applies the transformation and finally returns the result with the `Queue` implemented with a Redis service. In order to migrate Pandaral·lel from a parallel execution to a distributed environment, we just had to change the two lines of code where the libraries are imported (Listing 15) by two other lines of code importing the libraries specified above (Listing 16).

Listing 15 Pandaral·lel original imports.

```
1 from multiprocessing import get_context
2 import dill
```

In this experiment we used Pandaral·lel `apply()` function on a Pandas DataFrame to perform a sentiment analysis. The sentiment analysis has been done using the *textblob* Python module. Before doing the sentiment analysis, we do a text preprocessing using the regular expressions' library *re* that is part of the Python standard libraries. As dataset, we use the

Listing 16 Pandaral·lel modified imports.

```
from lithops.multiprocessing import get_context
import cloudpickle as dill
```

Sentiment140 [25] dataset, which contains 1,600,000 tweets extracted using the Twitter API. The dataset is loaded in memory from a CSV file with a size of 200 MB, but the dataset representation in a Pandas DataFrame is about 600 MB. Notice that this data will have to be transmitted from the Lithops host entirely to the storage backend before applying the transformation function. The main advantage of the parallel execution in a VM is that, as data is already in memory, it does not need to be transmitted nor loaded to anywhere. The function to apply to each tweet in the dataset is the *discretize_sentiment* function and the code of the experiment is available in Listing 17.

Listing 17 Twitter sentiment analysis code.

```
import time
import pandas as pd
from textblob import TextBlob
from pandarallel import pandarallel

def discretize_sentiment(text):
    text = preprocess_tweet(text)  # Remove metions, hashtags, links, emojis...
    polarity = TextBlob(text).sentiment.polarity
    if polarity < 0:
        return 'negative'
    if polarity == 0:
        return 'neutral'
    if polarity > 0:
        return 'positive'

args = get_args()
pandarallel.initialize(use_memory_fs=False, nb_workers=workers)

df = pd.read_csv('tweets.csv',
    header=None, sep=',',
    names=['target', 'id', 'date', 'flag', 'user', 'text'],
    engine='python'
)
serie = df['text']

t1 = time.time()
polarity = serie.parallel_apply(discretize_sentiment)
t2 = time.time()

print(f'{t2-t1} seconds')
```

The results in Figure 4.10 show that Lithops obtains a 7% lower performance compared to the best result from the VM due to data transmission and process invocation overheads. It can also be seen that Lithops is capable of maintaining correct scalability up to 96 vCPUs. Observe that the differences between VMs and Lithops grow as the number of vCPUs increases. This is mainly due to the fact that in this scenario Lithops has fixed overheads (the transmission of the dataset to the storage backend) and some overheads that grow with the number of vCPUs (the invocation of the functions). Also note that the greater the number of vCPUs, the granularity of the task becomes much smaller. When the granularity of the task is so small (note that with 96 vCPUs the total execution time is below 7 seconds), the Lithops overheads become a significant percentage of the total execution time. For this reason in 16 or 32 vCPUs the difference between Lithops and VMs is much smaller than when using 64 or 96 vCPUs.

If the granularity of the task was greater, with a larger dataset or with the use of a more intense computation on the data, the execution time would be greater and the proportional difference would be smaller since the Lithops overheads would be practically the same. Note that Lithops would allow working with larger datasets, since, as observed in the microbenchmarks, the reading and writing capacity of S3 is very high and its limit has not been reached in this experiment. Even so, we have shown that this type of data science applications with Pandas can be migrated transparently to a distributed system with serverless technologies with a small loss of performance. This can be compensated for the immediate availability of the resources that FaaS offers in contrast to the previous provisioning and configuration of the resources when working in a traditional monolithic system.

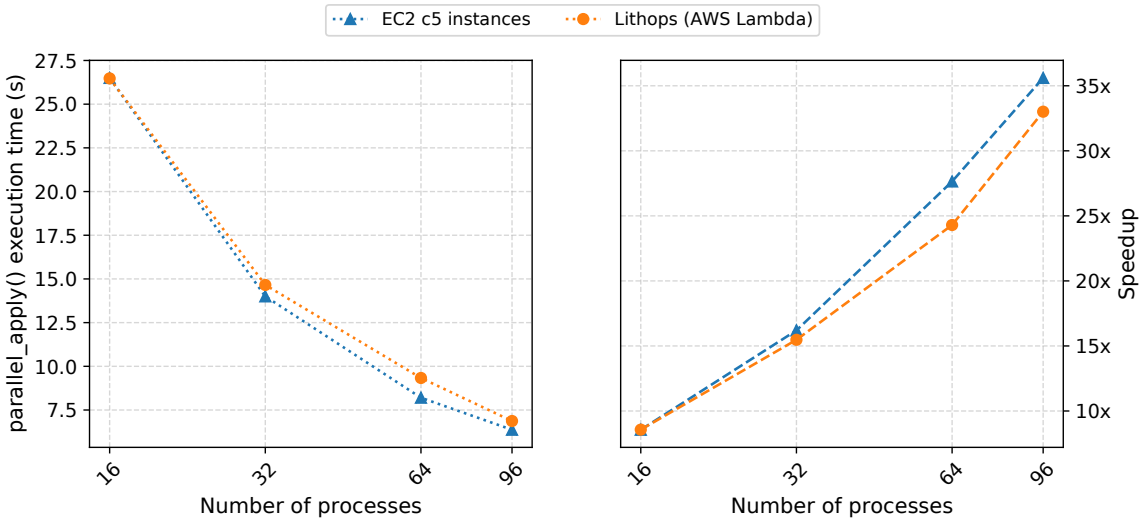

Figure 4.10: Pandaral·lel execution results.

4.3.3 Scikit-learn hyperparameter tuning

Scikit-learn is a Python module for machine learning built on top of SciPy. It is widely used and considered a standard to build machine learning models to solve classification, regression and clustering problems in the Python ecosystem. Here we want to test Lithops behavior on an embarrassingly parallel scenario with low data transmission.

Some scikit-learn utilities can parallelize their execution via the Joblib library. Joblib provides a lightweight pipelining in Python, in particular an API to do simple parallel computing. Joblib allows using different parallel backends such as *loky* or *multiprocessing*, but it also allows you to use and develop your own parallel backend. We have created a Lithops backend for Joblib using the original *multiprocessing* backend as a template. Changes with respect to the original *multiprocessing* parallel backend are minimal. To use the Lithops backend, the user has to first register this backend into Joblib, and then select it to run the parallel jobs. After that, scikit-learn transparently handles all job lifecycle while the original API calls remain unmodified.

With this new Lithops Joblib backend we can execute scikit-learn jobs in serverless functions. In particular, we can use the scikit-learn's Gridsearch over Lithops. The `model_selection` module that implements the `GridSearchCV` functionality has around 6650 lines of code, which makes it a complex application. In this experiment, we use Gridsearch to do a hyperparameter tuning on a SGD Classifier. The experiment consists of loading a chunk of a dataset and creating a model based on a scikit-learn `Pipeline` with a Hashing Vectorizer and a SGD Classifier. Then the parameters space is defined, in this experiment we are using a 5 dimensional parameters space with a total of 192 possible parameters combinations. Finally, the `GridSearchCV` functionality will perform a cross-validation with 5 folds using the data chunk and find the best parameters for the Pipeline model. Listing 18 contains the hyperparameter tuning core code.

To be able to use this code with Lithops, it is only required to register the Lithops Joblib backend and call `joblib.parallel_backend` with the backend name. To do so it is only required to add two lines of code, available in Listing 19.

As a data chunk for our hyperparameter tuning, we used 30 MB of an Amazon Reviews [11] dataset. In this experiment we also wanted to compare the behavior of Lithops using two different storage backends (S3 and Redis) so we used them with the default experiment settings explained previously.

The results in Figure 4.11 show that, with the same number of vCPUs, the execution time of the VMs is between 3% and 5% lower than the execution time of Lithops with S3 and between 3% and 7% higher than the execution time of Lithops using Redis. However, for this type of problems, the scalability of Lithops follows the same progression as the VM and Lithops quickly scales to much higher levels than the VM maximum, obtaining a speedup of up to

Listing 18 Hyperparameter tuning core code.

```
import joblib
from lithops.util.joblib import register_lithops
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import Pipeline
from time import time

args = get_args()

# Register Lithops backend as joblib backend
register_lithops()

# Build the model
n_features = 2**18
pipeline = Pipeline([
    ('vect', HashingVectorizer(n_features=n_features, alternate_sign=False)),
    ('clf', SGDClassifier()),
])

parameters = {
    'vect__norm': ('l1', 'l2'),
    'vect__ngram_range': ((1, 1), (1, 2)),
    'clf__alpha': (1e-2, 1e-3, 1e-4, 1e-5),
    'clf__max_iter': (10, 30, 50, 80),
    'clf__penalty': ('l2', 'l1', 'elasticnet')
}

grid_search = GridSearchCV(pipeline, parameters, error_score='raise',
                           refit=args.refit, cv=5, n_jobs=args.n_jobs)

with joblib.parallel_backend('lithops'):
    X, y = load_data(30)  # 30 MiB
    t0 = time()
    grid_search.fit(X, y)
    total_time = time() - t0
    print("Done in {}".format(total_time))
```

Listing 19 Joblib backend registration.

```
from lithops.util.joblib import register_lithops
register_lithops()
```

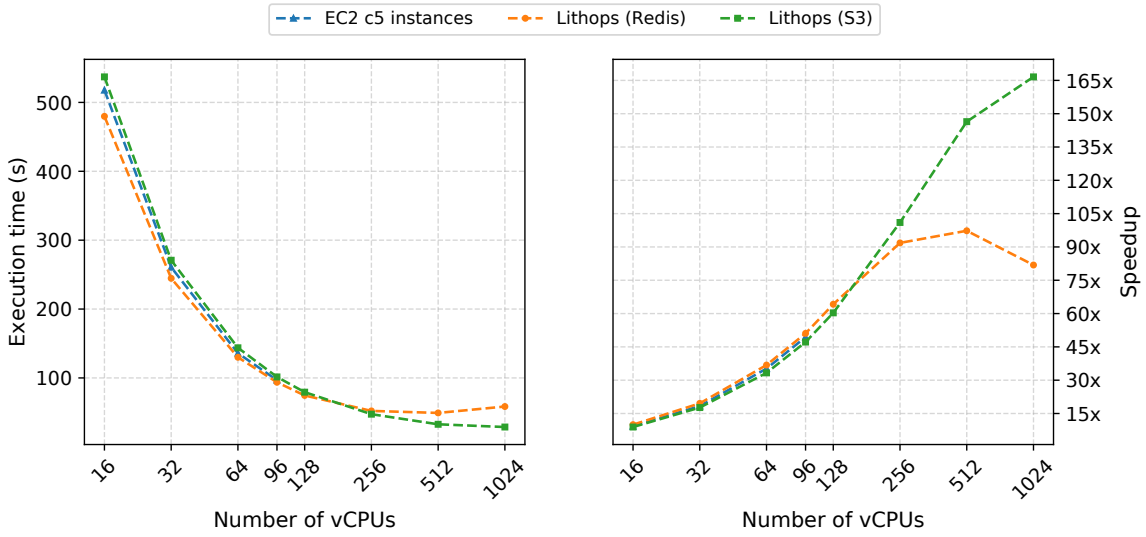

Figure 4.11: Hyperparameter tuning execution results.

3.6x times greater. Finally, we can see that Redis is about a 10% better than S3 when few processes are used and, from 256 processes, Redis begins to saturate but S3 continues scaling correctly. This is caused by the increasing number of concurrent reads. S3 is able to serve a higher number of concurrent reads compared to Redis which is single-threaded. This implies that S3 has better bandwidth and throughput, which decreases the data read overhead, and consequently decreases the overall execution time.

For this type of highly parallel tasks with little data transmission, the use of serverless disaggregated resources can be faster than the use of the same resources in a single VM. In this experiment we are seeing again how the use of the Hyper-Threading technology favors the disaggregated system since it compensates for the existing overheads in Lithops.

4.3.4 Proximal Policy Optimization

OpenAI Baselines [16] is a set of high-quality implementations of reinforcement learning algorithms. It has been open-sourced to be used as a base, around which, new ideas can be added and as a tool for comparing new approaches in the reinforcement learning field. Baselines' code repository contains more than 16700 lines of Python code, so it can be considered as a complex Python module. In this experiment, we want to verify that thanks to the access transparency provided by Lithops *multiprocessing* we can simulate the vertical scaling of a virtual machine using FaaS as processes.

In this experiment we are using the Proximal Policy Optimization (PPO) algorithm to train a neural network to play the Atari game Breakout, which is available in the OpenAI GYM [12]. Figure 4.12 shows a frame of the Breakout game for the Atari console.

Figure 4.12: Atari Breakout game frame.

We have used the *multiprocessing* implementation of the PPO algorithm from OpenAI baselines. The *multiprocessing* PPO version is the second implementation released by OpenAI, and it inherits some of its structure from the first version, which was based on MPI. For that reason, the *multiprocessing* PPO uses a master/worker paradigm relying on `Pipes` for the master to worker communications and vice versa.

Figure 4.13 shows a basic scheme of the algorithm flow for the PPO. The master process is in charge of training the model (a neural network) which, for a given set of states[4] (1), decides the optimum actions[5] to do in order to maximize an objective function (2). The actions to perform are send to the worker processes via `Pipes` (3). The worker processes are used to simulate the environment (the Breakout game) in which the actions are performed and a resultant state is obtained (4). The state resulting from performing the action is sent from the workers to the master using the same `Pipe` (5). At this point, if the training is not finished, the system does another iteration starting on point (1). Notice that each worker only simulates one environment.

The whole PPO algorithm is able to run with Lithops by just changing the original *multiprocessing* import by the *Lithops multiprocessing* module import. Similar to the Evolution Strategies experiment, to obtain a comparable result we have changed how workers are invoked in the original code. It was performed in a loop and it has been changed to a `Pool.map_async` call because in Lithops this method is much more efficient. More precisely, the code of Listing 20, that shows the original code where processes are invoked, has been replaced by the `map_async` statement in Listing 21

As this algorithm requires the use of a GPU in the master process for the neural network

[4]In our experiment, each state correspond to a particular moment in a Breakout game.
[5]The actions here are the posible actions that the player can do, for example move right/left or stay still.

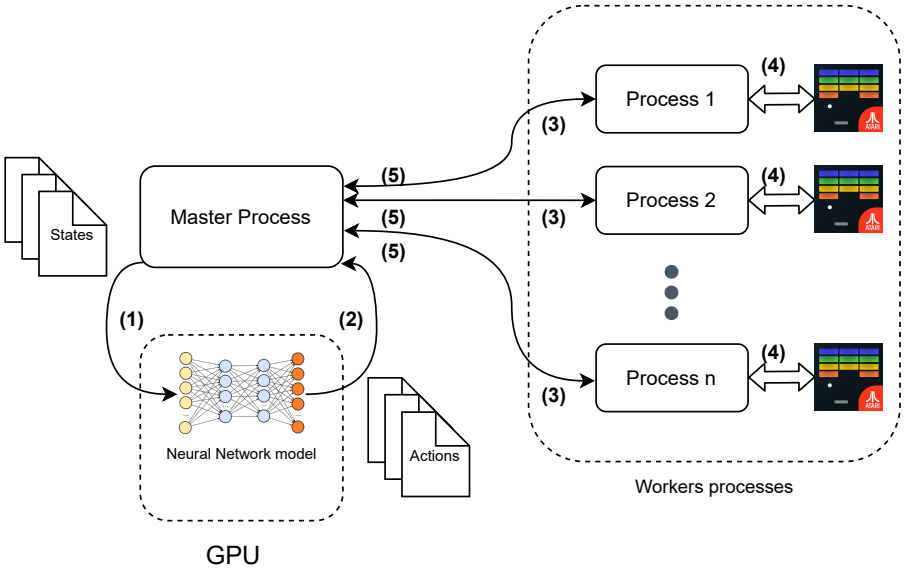

Figure 4.13: Proximal Policy Optimization scheme.

Listing 20 PPO original processes invocation.

```
self.ps = [ctx.Process(target=worker, args=(work_remote, remote, CloudpickleWrapper(env_fn)))
    for (work_remote, remote, env_fn) in zip(self.work_remotes, self.remotes, env_fns)]

for p in self.ps:
    p.daemon = True
    with clear_mpi_env_vars():
        p.start()
```

Listing 21 PPO modified processes invocation.

```
self.ps = ctx.Pool().map_async(
    worker,
    zip(self.work_remotes, (CloudpickleWrapper(env_fn) for env_fn in env_fns))
)
```

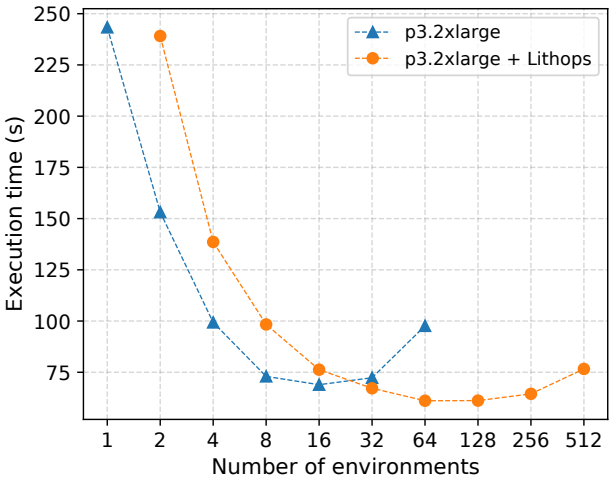

Figure 4.14: PPO execution results.

training, the settings for this experiment have been modified. We have used an AWS p3.2xlarge VM as monolithic system, and we tried to scale it vertically using AWS Lambdas. Since the GPU is just used in the master process that runs in the Lithops host and not in the workers that just do environment simulation, the configuration of the AWS lambdas has not been modified. Also notice that due to the TensorFlow 1 dependency, we have used Python 3.7 in this experiment.

The results, available in Figure 4.14, show that despite the constant communication between processes and the great overhead that this entails, the combination of VM and Lithops achieves a better performance than just the VM and reduces an 11% the execution time. This validates that we can emulate a vertical scaling of the VM, and that it is possible to add vCPUs to a VM instantly and without prior provisioning thanks to the use of FaaS and the access transparency provided by Lithops.

4.4 Summary

Results from Section 4.2 show that the main overheads of the disaggregated serverless compute are: the time it takes for the system to fork/join with cloud functions and the communication overheads, both latency and throughput. On the other hand, it has been seen that in intensive computing tasks, the disaggregated system obtains a performance that is 25% higher. This difference is due to the fact that, despite using the same number of vCPUs, the use of Hyper-Threading technology harms the centralized system. In this way, the existing overheads of the disaggregated system are compensated by the access to a greater number of physical CPUs, step-siding the limitations of Hyper-Threading. Cloud functions are a good tool to exploit the

great scalability of S3 technology, applications with a large number of read or write instructions on disk could benefit from this scalability. Finally, it has been seen that, although it is possible to abstract shared memory elements to be able to use them in a disaggregated environment, their use can cause a prohibitive overhead and it would be advisable to look for algorithms that use communication systems such as Pipes.

The four experiments with the applications in Section 4.3 show that it is possible to obtain access transparency with minimal or even none performance degradation. In the results of Evolution Strategies and hyperparameter tuning imply that it is possible to obtain a superior computational performance beyond the VMs' limit, getting 3.6x (hyperparameter tuning) and 1.3x (ES) better results. However, the Pandaral·lel results show that it is not always possible to improve the performance of VMs, since a performance loss of 7% is obtained. Scenarios where large data transfer is required are particularly affected. Finally, in the experiment with the PPO algorithm it has been found that the vertical scaling of a VM can be simulated using cloud functions. This allows you to add computing power to a system immediately and without prior provisioning.

Chapter 5

Insights

After studying the results of the evaluation, we have learned that we are able to move local multi-processing applications to distributed settings using current serverless offerings transparently. Nevertheless, we have chosen only four representative applications that use the Python multiprocessing library in Python. We just demonstrated that some parallel programming applications can be executed and scaled massively in a transparent way without significant degradation.

In this chapter, we outline several key insights from our experience that have a direct implication for achieving transparency.

5.1 Distributed and parallel programming abstractions

Insight #1: Parallel and distributed programming abstractions have a clear impact in achieving transparency.

The way that Python multiprocessing library is designed has made it easier to implement these parallel programming abstractions in a distributed and disaggregated way. Python `multiprocessing` library has a clean shared memory abstractions to communicate processes.

Structured and consistent access to shared memory requires suitable programming abstractions. If the code is not using adequate shared-state abstractions, full transparency may be impossible. We cannot generalize our claim to all kind of stateful applications. In particular, one of the major problems for achieving this full transparency is how applications deal with shared memory objects. If they access shared memory without using the clean shared object primitives (like `Manager` in the multiprocessing API), transparency might not be possible.

The ability to perform parallel execution in Python using threads is limited by the Global Interpreter Lock (GIL). The GIL prevents multiple threads from running simultaneously on multiprocessor architectures. This fact has had an impact on the way local parallel program-

ming is performed in Python. In Python, it is necessary to use processes to have true parallel execution. For this reason, many of the principles of multiprocessing abstractions, such as Manager, are based on message passing and accessing shared objects (queues, Manager.dict) instead of traditional memory sharing. For example, in a multi-thread application, two parallel threads can access a shared object by a reference pointer. In contrast, in Python multiprocessing, processes have three ways of sharing state. The first involves using shared memory abstractions, such as `multiprocessing.Array`, which can only store basic C types (`int`, `float`...) so its usage is very limited. Another way to share state is by message passing using pipes or queues, which implies serializing the object each time is passed through. Some objects can't be serialized, besides, serializing multiple times an object with a lot of state can add overhead and result in extra memory usage. The last method is using a third process (`multiprocessing.Manager`) that stores sate and other processes access it using Remote Method Invocation. The fact that two Python processes can't share the same address space has facilitated the port of this library to its distributed implementation using disaggregated resources, since the most used methods (Pipes/Queues and Managers) have a clear and clean API that can be easily adapted and re-implemented.

5.2 Main program execution

Insight #2: The application control flow should be managed by the Cloud provider for enhanced fault tolerance.

Another important aspect to highlight is where and how the main program is executed. In our case, we had to run the main program in the Cloud using virtual machines in *IaaS*. Mainly, this decision has been taken to minimize the overheads generated in accessing disaggregated memory (Redis) and computing (AWS Lambda) resources. It is entirely possible to run the same experiments running the main program from a user's laptop from the edge (outside the cloud). However, latencies would be considerably higher.

We propose that future work should address in more depth the possibility of the Cloud to control itself the entire lifecycle of the application. Currently, the application is treated as a "black box" that accesses the Cloud services. If we could somehow remove that barrier and "compile" our application directly to the Cloud, then we would no longer have to worry about the execution of the application, since the Cloud itself would control the application lifecycle in a secure and fault-tolerant way.

A clear example is the Durable Functions [36] in Azure Cloud. Durable Functions are a type of serverless function workflow orchestration where the workflow logic is defined as code (called *orchestrator function*). The *orchestrator function* execution is interrupted each time it calls one

or more serverless functions, does not resume until all called functions have completed their execution. When the *orchestrator function* is re-executed, instead of calling the same function again, it first checks to see if it already has the result obtained from having previously called that function. Another example is Triggerflow [8] workflow as code abstraction. In Triggerflow, the user is able to define a script that uses Lithops for parallel execution of *map* and *map-reduce* jobs. With Triggerflow, similarly to Azure Durable Functions, the application logic flow is managed by the workflow orchestration system in the Cloud, guaranteeing fault tolerance. When the script reaches a Lithops *map* or *map-reduce*, the execution stops and it only resumes when all functions have finished, using the same strategy of event sourcing to restore the state of the script. The difference, compared to Azure Durable Functions, is that Triggerflow is more flexible in terms of what applications it can orchestrate.

As we can see, in these cases we have delegated to the Cloud the orchestration of our application, and the Cloud is responsible for stopping and resuming the code when necessary and in a fault-tolerant manner. However, this solution presents a clear problem, which is that the function must be idempotent. Each time the workflow execution is resumed, the entire *orchestrator function* is re-executed from the beginning, so it must be idempotent. A possible solution to this problem is to use checkpoint/restore. CRIU [4] (Checkpoint/Restore In Userspace) is a project whose objectives is to pause (checkpoint) Linux processes or containers, save their state to disk and resume (restore) the process to the same point where the checkpoint was made. Being able to checkpoint/restore processes or containers is promising and has many advantages, for example, it improves fault tolerance, since we could make regular checkpoints and recover any previous checkpoint at any time. It also improves flexibility because a process can be move "live" from one machine to another. It is even useful for debugging applications. This solution has many advantages over the Durable Functions proposal, since the function does no longer have to be idempotent and it can have an internal state.

In fact, how distributed stateful applications are programmed is still a matter of debate. The distributed systems' community has always been indecisive on what programming abstraction is best: RPCs, remote objects and RMI, actors, or event distributed languages like Erlang or Elixir. A recent and interesting proposal is presented in [14]. The authors propose that distributed Cloud applications should follow a model called *PACT*, which stands for Program semantics, Availability specifications, Consistency guarantees and Targets for dynamic optimization. They argue that a more declarative and asynchronous specification can prevent problems related to parallel interleaving, message reordering and partial failures, compared to an imperative specification, where they are much harder to handle. For availability, the authors state that a declarative specification can also allow programmers to specify the availability of parts of the application, independently of their program semantics. Regarding consistency, they

advocate for data immutability and monotonic transformations, which the programmer must specifically indicate in the application declaration. In their paper, they state that a compiler is necessary to interpret the declarative specification of the application. The compiler can then apply the necessary optimizations in order to maintain and guarantee availability, scalability, consistency and failure resilience of the application.

Our vision is, in part, the opposite of the one presented in the article [14]. Our proposal is to keep the same sequential and imperative programming abstraction of the application and let the middleware, the disaggregated services layer, and the Cloud to be responsible for the code execution flow. This way we can guarantee fault tolerance and achieve consistency and scalability.

5.3 Latencies and overheads

Insight #3: Overheads are still relevant and locality can't be obviated.

Current Cloud settings still show relevant latency in communications, like hundreds of milliseconds to launch a serverless function, or hundreds of microseconds to access in-memory computing services. We have seen that, with equal resources, the overheads generated by creating processes and by the latency of access to shared state are very noticeable. In this line, the granularity of computing tasks is clearly limited by overheads. Very fine-grained computing tasks do not make sense in the current Serverless model, since the overheads can be greater than the task run time.

The interesting point here is that compute-intensive parallel applications have certain advantages in Cloud Serverless settings that can mitigate some overheads. On the one hand, Hyper-Threading may cause performance degradation in virtual machines for compute-intensive tasks using all vCPUs. In Serverless Functions, the load is distributed in several machines and the overall results improve. On the other hand, accessing large volumes of data in Cloud Object Storage from Serverless functions helps to aggregate bandwidth and accelerate data transfers. A single VM cannot compete with parallel data flows from multiple functions.

As we have seen in the evaluation of the parallel quick-merge sort experiment, or in the evaluation of the PPO application, locality is important and should not be overlooked. In the case of the parallel quick-merge sort micro-benchmark, the performance of the application is determined by the way it accesses memory. That is, if the data is close to the computation (e.g., in a cache) and the remote disaggregated memory is not continuously accessed, the execution time is lowered considerably. For the PPO algorithm, locality is determined by the usage of the GPU to train the neural network model. The locality here is important since the GPU must have the data in local to have a better performance. This makes us think that not all

applications can be ported to the Cloud "as-is" using transparency. In some cases the algorithm and the code will have to be modified, as in the case of the parallel quick-merge sort, in order to obtain an feasible result. Even if we use the same operations to transparently access local or remote resources, the way we use these operations directly affects the performance of the application.

5.4 Serverless services

Insight #4: Much work is still needed by Cloud providers to improve disaggregated resources (compute, memory and storage) and lower the latencies, overheads and guarantee fault tolerance.

Serverless disaggregated memory and fine-grained storage is needed. Another important issue is the scaling and fault tolerance of the in-memory layer. We are relying on a dedicated Redis service, which must be properly managed now. If the data flows exceed the capacity of this intermediate node, the experiment would fail. Unlike Serverless Functions or Cloud Object Storage, In-memory computing is still not offered as a managed service with scalability and fault tolerance.

Regarding storage, we are now intercepting file access that is routed to Object Storage. But Object Storage has certain limitations regarding small files or read/write operations. Intensive use of such operations by applications would also preclude transparency.

Finally, we are not considering cost in this evaluation. Nowadays, the economic cost of running the code in an on-demand VM with full resource utilization is cheaper (half the price) than running it on serverless functions. We also expect that the cost of disaggregated serverless services will be reduced in the future to make the idea of transparency economically feasible.

Chapter 6

Conclusions

In this research work, we have demonstrated that Python multiprocessing applications can be transparently executed over serverless functions with minor degradation compared to the same application running on a single VM. We have presented a middleware that implements the Python multiprocessing interface but executes processes on serverless functions and that leverages a disaggregated in-memory storage to implement shared state and synchronization abstractions. We have shown that, despite the overheads generated from function invocation and remote state access, we obtain comparable execution times with respect to the same application running on a monolithic server, in addition to being able to scale the application beyond the limits of server's fixed resources.

6.1 Open problems and future work

This research work is only a first step towards transparency. However, we have only selected four applications as validation, so our results are not fully representative of the local parallel programming landscape using Python multiprocessing. We have seen from the related work that access transparency to disaggregated resources is still very much in its infancy and has a long way to go until it becomes fully viable.

We have not gone into deeper issues related to fault tolerance, since we have assumed that the disaggregated resources on which we mount our application are fault tolerant. However, we know that in distributed systems, fault tolerance is a very important aspect, and we hope that in the future, work will be carried out in order to provide serverless disaggregated resources that are resilient to failures but also are flexible and scalable.

The results obtained make us optimistic that, in the future, when network latencies in data centers can be reduced, disaggregated resources will have a lower overhead, which means that full access transparency could be eventually achieved. Making transparency viable would

have important implications. First, we would stop using monolithic servers as deployment unit and start thinking about granular resources that adapt and scale according to the needs of the application. This would greatly help to facilitate the adaptation of legacy applications to the cloud, avoiding application modernization. Second, the simplicity provided by access transparency would imply that we could directly code the Cloud as a large parallel Super-Computer, thus preventing the application programmer from having to worry about the complexities of distributed systems.

6.2 Suggested readings

If the reader is interested in further researching the topics covered in this thesis, we suggest the following list of recommended readings:

- **The rise of Serverless computing** by Paul Castro, Vatche Ishakian, Vinod Muthusamy and Aleksander Slominski [13]: This article does a very good job in presenting Serverless computing in a very clear and detailed way.
- **Toward Multicloud Access Transparency in Serverless Computing** by Josep Sampe, Pedro Garcia-Lopez, Marc Sanchez-Artigas, Gil Vernik, Pol Roca-Llaberia and Aitor Arjona [41]: This magazine article published in the IEEE Software journal goes into more detail on the architecture and implementation of the Lithops framework.
- **Towards scaling and access transparency in serverless machine learning** by Pol Roca-Llaberia [39]: This final degree project was the precursor to the research carried out in this thesis. It presents an earlier version of the multiprocessing module that did not fully cover the entire multiprocessing interface. This project focuses more on transparency for machine learning while we consider transparency for a broader scope of general-purpose applications. Based on the work done in this final degree project, we continued his research and gone into deeper comparisons of the transparency with a monolithic machine.
- **Serverless End Game: Disaggregation enabling Transparency** by Pedro García-López, Aleksander Slominski, Simon Shillaker, Michael Behrendt and Barnard Metzler [35]: This article explores current transparency proposals and argues why resource disaggregation and serverless computing will enable full transparency in the Cloud.
- **Serverless Predictions: 2021-2030** by Pedro Garcia Lopez, Aleksander Slominski, Michael Behrendt and Bernard Metzler [34]: This paper presents five predictions relating how serverless computing advancements will eventually enable transparency.

6.3 Personal insights

Finally, we would like to present the authors' vision of this work. We want to express our gratitude for having been able to do this work together and in a university research group in the context of the CloudButton 2020 European research project.

We are proud that our work has been submitted to the Middleware 2021 conference and is currently under peer review. During the writing process of the article, we learned how to plan a research study and how to translate the knowledge and results acquired in just 12 pages of double-column text. Without a doubt, it was a tedious but very rewarding task. Hopefully this work can be presented at this prestigious conference, and if it is not accepted, we will work hard to get our work considered again in the future.

We would like to thank our tutor Pedro García and the rest of the members of the CLOUDLAB-URV research group and H2020 CloudButton project research fellows for the opportunity to be part of this research that is very hot and relevant nowadays.

In this work we have had to handle production-grade technologies of distributed systems and Cloud such as AWS EC2, AWS VPC, AWS Lambda, AWS S3, Docker containers, Redis, Kubernetes... And we have had to handle applications with thousands of lines of code such as Lithops, Pandaral·lel, Joblib, Baselines and PPO. In short, the realization of this work and especially the validation has been a challenge, but it has certainly been a very enriching experience and we have learned a lot during these months of research.

Overall, this experience has been very enriching and we think that this research has a lot of work ahead of it, and we will be happy to continue with this line of research in our future PhD studies.

Bibliography

[1] AWS Elasticache, 2021. URL: https://aws.amazon.com/elasticache.

[2] AWS Lambda, 2021. URL: https://aws.amazon.com/lambda/.

[3] AWS S3 Strong Read-After-Write Consistency, 2021. URL: https://aws.amazon.com/s3/consistency/.

[4] Checkpoint/Restore In Userspace (CRIU), 2021. URL: https://criu.org/Main_Page.

[5] Gil definition in the python glossary, 2021. URL: https://docs.python.org/3/glossary.html#term-global-interpreter-lock.

[6] Python multiprocessing documentation, 2021. URL: https://docs.python.org/3/library/multiprocessing.html.

[7] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020. URL: https://www.usenix.org/conference/hotcloud20/presentation/angel.

[8] Aitor Arjona, Pedro García López, Josep Sampé, Aleksander Slominski, and Lionel Villard. Triggerflow: Trigger-based orchestration of serverless workflows. *Future Generation Computer Systems*, 124:215–229, 2021. URL: https://www.sciencedirect.com/science/article/pii/S0167739X21001989, doi:https://doi.org/10.1016/j.future.2021.06.004.

[9] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3361525.3361535.

[10] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[11] Adam Bittlingmayer. Amazon reviews for sentiment analysis. https://www.kaggle.com/bittlingmayer/amazonreviews, 2017.

[12] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. arXiv:arXiv:1606.01540.

[13] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, November 2019. doi:10.1145/3368454.

[14] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Matthew Milano. New directions in cloud programming, 2021. arXiv:2101.01159.

[15] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems - Concepts and Design (5th edition)*. Addison-Wesley Longman, Inc., 2012.

[16] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

[17] Dropbox. Easy file syncing, 2021. URL: https://www.dropbox.com/features/sync.

[18] IBM Cloud Education. IBM Cloud Learn Hub - FaaS (Function-as-a-Service). https://www.ibm.com/cloud/learn/faas, 7 2019. (Accessed on 16/06/2021).

[19] IBM Cloud Education. Ibm cloud learn hub - lift and ship. https://www.ibm.com/cloud/learn/lift-and-shift, 12 2019. (Accessed on 16/06/2021).

[20] IBM Cloud Education. IBM Cloud Learn Hub - Microservices. https://www.ibm.com/cloud/learn/lift-and-shift, 3 2021. (Accessed on 16/06/2021).

[21] IBM Cloud Education. Ibm cloud learn hub - serverless computing. https://www.ibm.com/cloud/learn/serverless, 4 2021. (Accessed on 16/06/2021).

[22] Flexera. Cloud Computing Trends: 2021 State of the Cloud Report. https://www.flexera.com/blog/cloud/cloud-computing-trends-2021-state-of-the-cloud-report/, 03 2021. (Accessed on 16/06/2021).

[23] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, mar 2017. USENIX Association. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[24] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 249–264, 2016.

[25] Alec Go, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision. *Processing*, pages 1–6, 2009. URL: http://www.stanford.edu/~alecmgo/papers/TwitterDistantSupervision09.pdf.

[26] Google Cloud. Google Cloud Customers, 2021. URL: https://cloud.google.com/customers.

[27] Google Trends. "Serverless", 2021. URL: https://trends.google.com/.

[28] IDG. 2020 Cloud Computing Study. https://www.idg.com/tools-for-marketers/2020-cloud-computing-study/, 08 2020. (Accessed on 16/06/2021).

[29] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

[30] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, 2019. USENIX Association. URL: https://www.usenix.org/conference/nsdi19/presentation/kalia.

[31] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. Technical report, USA, 1994.

[32] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, 2019.

[33] Andy Lipnitski. Moving Legacy Applications to Cloud Environments – Why and How. https://www.scnsoft.com/blog/application-migration-to-cloud, 8 2020. (Accessed on 16/06/2021).

[34] Pedro Garcia Lopez, Aleksander Slominski, Michael Behrendt, and Bernard Metzler. Serverless predictions: 2021-2030, 2021. arXiv:2104.03075.

[35] Pedro García López, Aleksander Slominski, Simon Shillaker, Michael Behrendt, and Bernard Metzler. Serverless end game: Disaggregation enabling transparency. *CoRR*, abs/2006.01251, 2020. URL: https://arxiv.org/abs/2006.01251, arXiv:2006.01251.

[36] Microsoft Azure Documentation. What are Durable Functions?, 2021. URL: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[37] Ron Miller. How aws came to be. *Tech Crunch*. URL: https://techcrunch.com/2016/07/02/andy-jassys-brief-history-of-the-genesis-of-aws/.

[38] Manu NALEPA. Pandarallel. https://github.com/nalepae/pandarallel, 2021.

[39] Pol Roca-Llaberia. Towards scaling and access transparency in serverless machine learning. URL: http://cloudlab.urv.cat/web/theses?view=thesis&task=show&id=14.

[40] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's time for low latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, page 11, USA, 2011. USENIX Association.

[41] Josep Sampe, Pedro Garcia-Lopez, Marc Sanchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. Toward multicloud access transparency in serverless computing. *IEEE Software*, 38(1):68–74, 2021. doi:10.1109/MS.2020.3029994.

[42] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, page 1–8, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3284028.3284029.

[43] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, 2018. USENIX Association. URL: https://www.usenix.org/conference/osdi18/presentation/shan.

[44] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra, 2018. arXiv:1810.09679.

[45] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, 2020. URL: https://www.usenix.org/conference/atc20/presentation/shillaker.

[46] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[47] Sai Vennam. IBM Cloud Learn Hub - What is Cloud Computing? https://www.ibm.com/cloud/learn/cloud-computing, 8 2020. (Accessed on 06/16/2021).

[48] Tim Wagner. The Serverless SuperComputer, 2019. URL: https://read.acloud.guru/https-medium-com-timawagner-the-serverless-supercomputer-555e93bbfa08.

[49] Rui Wang, J. Lehman, J. Clune, and K. Stanley. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *ArXiv*, abs/1901.01753, 2019.

[50] Jin Zhang, Zhuocheng Ding, Yubin Chen, Xingguo Jia, Boshi Yu, Zhengwei Qi, and Haibing Guan. Giantvm: A type-ii hypervisor implementing many-to-one virtualization. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3381052.3381324.

[51] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Understanding the effect of data center resource disaggregation on production dbmss. *Proc. VLDB Endow.*, 13(9):1568–1581, May 2020. doi:10.14778/3397230.3397249.

[52] Jiale Zhi, Rui Wang, Jeff Clune, and Kenneth O. Stanley. Fiber: A platform for efficient development and distributed training for reinforcement learning and population-based methods, 2020. arXiv:2003.11164.