Gil Arasa Verge

Serverless OCaml genomic pipeline parallelisation engine

Final Degree Project

directed by Dr. Pedro García López

Computer Engineering

UNIVERSITAT ROVIRA i VIRGILI

Tarragona
June 2022

Resum

La comunitat científica utilitza cada cop més el *cloud computing* per evitar o minimitzar els costos inicials d'infraestructura i manteniment, proporcionant recursos informàtics sota demanda per impulsar fluxos de treball complexos d'anàlisi de dades. Aquests fluxos de treball requereixen una mentalitat de *big data*, manejant una quantitat inigualable de dades sota els registres d'accés obert disponibles, minimitzant les transferències de dades innecessàries fora del núvol. Com que la bioinformàtica és un camp que canvia ràpidament, és comú trobar patrons de programació enganxats en el passat, no ajustats adequadament a les tendències actuals de la computació. Això es deu principalment al creixement de les dades de genòmica, que ha augmentat més ràpidament que el ritme al qual els desenvolupadors podrien escalar o optimitzar els seus programes. La transparència al núvol significa accedir a recursos locals i remots mitjançant operacions idèntiques, amagant la complexitat en termes de càlcul, memòria, emmagatzematge i escalat de grans volums de dades que no seria possible tenir en configuracions d'aplicacions científiques paral·leles locals. L'objectiu del nostre treball és ampliar el coneixement actual d'interoperabilitat entre el núvol i el programari desenvolupat localment en un paradigma agnòstic del llenguatge, seguint principis de transparència per migrar una arquitectura OCaml de multiprocessament particular a *serverless*. Com a punt final, comparem l'escalabilitat del rendiment d'un model realment *serverless* amb una màquina virtual. Els nostres resultats mostren que els desenvolupadors poden beneficiar-se dels principis de *serverless*, especialment de l'escalabilitat del rendiment, fins i tot quan mantenen la seva arquitectura de codi heretada amb modificacions menors.

Resumen

La *computación en la nube* está siendo utilizada cada vez más por la comunidad científica para evitar o minimizar los costos iniciales de infraestructura y mantenimiento, proporcionando recursos informáticos bajo demanda para impulsar flujos de trabajo de análisis de datos complejos. Esos flujos de trabajo requieren una mentalidad de *big data*, manejando una cantidad incomparable de datos bajo las bases de datos públicas disponibles, minimizando cualquier transferencia de datos innecesaria fuera de la nube. Dado que la bioinformática es un campo que cambia rápidamente, es común encontrar patrones de programación estancados en el pasado, que no se ajustan adecuadamente a las tendencias actuales de la computación. Esto se atribuye principalmente al crecimiento de los datos genómicos, que ha aumentado más rápido que el ritmo al que los desarrolladores pueden escalar u optimizar sus programas. La transparencia de la nube significa acceder a recursos locales y remotos mediante operaciones idénticas, ocultando la complejidad en términos de cómputo, memoria, almacenamiento y escalado de grandes volúmenes de datos que no sería posible tener en configuraciones de aplicaciones científicas paralelas locales. El objetivo de nuestro tra-

bajo es ampliar el conocimiento actual de la interoperabilidad entre la nube y el software desarrollado localmente en un paradigma independiente del idioma, siguiendo los principios de transparencia para migrar una arquitectura OCaml de multiprocesamiento particular a *serverless*. Como punto final, comparamos la escalabilidad del rendimiento de un modelo verdaderamente *serverless* con una máquina virtual. Nuestros resultados demuestran que los desarrolladores pueden beneficiarse de los principios de *serverless*, en particular la escalabilidad del rendimiento, incluso cuando mantienen su arquitectura de código heredada con modificaciones menores.

Abstract

Cloud computing is increasingly being used by the scientific community to avoid or minimize up-front infrastructure costs and maintenance, providing on-demand computing resources to power complex data analysis workflows. Those workflows require a *big data* mindset, handling an unparalleled amount of data under the available open access registries, minimizing any unnecessary data transfers outside the cloud. As bioinformatics is a rapidly-changing field, it is common to find programming patterns stuck in the past, not properly adjusted to the current trends in computing. That is mainly attributable to the growth of genomics data, which has increased faster than the pace at which developers could scale or optimize their programs. Cloud transparency means accessing local and remote resources using identical operations, hiding the complexity in terms of compute, memory, storage and scaling big data volumes that would not be possible to have in local parallel scientific application setups. The aim of our work is to broaden current knowledge of interoperability between cloud and locally developed software in a language-agnostic paradigm, following transparency principles to migrate a particular multiprocessing OCaml architecture to *serverless*. As a final point, we compare performance scalability of a truly *serverless* model against a virtual machine. Our results show that developers can benefit from *serverless* principles, particularly performance scalability, even when maintaining their legacy code architecture with minor modifications.

Contents

List of Figures

Code Index

List of Tables

1 Introduction

The aim of this work is to study the migration to the cloud of an existing parallel framework written in OCaml using tools available in other programming languages. This framework has been extensively used in developing individual stages for genomics data analysis pipelines. Genomics data analysis workflows or pipelines are composed of different processing stages where each stage represents a computation, specified in terms of complex algorithms/code to accomplish a concrete task.

This project has been a collaboration of Rovira i Virgili University CloudLab research group and The James Hutton Institute under the *H2020 CloudButton: Serverless Data Analytics Platform* project [7].

1.1 CloudButton

CloudButton is a project inspired by the following comment from a professor of computer graphics at UC Berkeley: "Why is there no cloud button?". He described how his pupils simply desired they could "click a button" and have their existing, optimized, single-machine code run on the cloud. Thanks to serverless technologies, CloudButton attempts to "democratize big data" by greatly simplifying the development cycle and programming methodology.

The CloudButton project mainly targets multidisciplinary topics such as bioinformatics (genomics, metabolomics) and geospatial data (LiDAR, satellital) but also encompasses computer science fundamentals at its core [7].

Genomic analytics researchers require big data ready infrastructure to run experiments on. This infrastructure drags management, maintenance, provisioning, and operating costs which end up being a limitation on the scaling of the experiments.

As a solution to these problems, moving the algorithms to the cloud would let laboratories with lesser resources to access the computing resources from anywhere in the world and allow researchers to use cutting-edge computing and storage facilities that would not be plausible to maintain on-premises for intermittent usage, billing only based on the actual amount of resources consumed by the studies. Additionally, moving pipelines to the cloud would allow the genomics field to increase concurrent studies by running them in parallel with almost unlimited on-demand allocation.

This thesis is contemplated in one of such genomics use-cases where a code designed to typically run on a single machine is adapted to completely run on a serverless cloud paradigm.

1.2 Transparency

The term *transparency* refers to the ability to hide the complexity of distributed programming such as remote locations, failures, and scaling. To achieve full transparency, we must be able to compile, debug, and run unmodified single-machine code over effectively infinite compute, storage, and memory resources. Accessing disaggregated resources in a transparent manner involves a new type of lightweight, flexible virtualization that does not exist now. This virtualization must intercept computation and memory management in order to offer access to disaggregated resources, and it must do so with native-like performance and without the need for programmer input [8].

Transparency is easier to achieve in stateless applications than it is in stateful applications, where locality and coordination demand a different programming and resource management strategy. The most prevalent criticism to transparency is that distant memory can never match local memory in terms of speed. However, not all parallel applications involve frequent memory access; in fact, many rely only on disaggregated communication and synchronization primitives. The computation time in the local and remote versions of embarrassingly parallel workloads has been proven to be similar using cloud functions. Even when all network connection overheads, container management, and remote execution are included in, the results for disaggregated computations are competitive in terms of performance in existing clouds. Those computation workloads are long enough to make 100 millisecond overheads negligible [8, 9].

As transparency can be easily implemented at the application level rather than at the operating system level, we can seamlessly run unmodified local code in a distributed form if the application's interface for accessing local resources is replaced or wrapped by another implementation that accesses disaggregated resources instead [8].

A fundamental question for cloud transparency is whether it is possible to program the cloud as if it was an infinite multi-core machine, with virtual CPUs replacing physical CPUs, virtual memory replacing physical memory, and cloud storage replacing physical storage, along with with other infrastructure requirements.

1.3 Objective

The objective of this project is to port to the cloud a genomics pipeline step with black box testing implementation conditions. As the data analysis workflow for genomics tends to be based on different steps, usually written in different programming languages or even scripting languages, the design should be applicable to a single step in the whole pipeline, allowing it to be plugged and chained to different genomics pipelines, granting the benefit of reusability and genericity when composing bigger data workflows. Other key points for the proposed solution are:

1. **Scalability**: Following serverless principles, use cloud services to leverage the scaling of memory, compute, and storage automatically.

2. **Legacy code and transparency**: The migration to the cloud must tolerate legacy code to run in similar or better conditions, providing serverless benefits to existing code and using the least amount of code modifications to run the same single-machine code in the cloud with disaggregated resources instead of local resources.

3. **Adequate resource allocation and billing**: Guarantee being only charged for the precise resources consumed during program execution, avoiding the expense of overprovisioned idling resources during demand drops.

4. **Simplicity/Productivity**: Avoid the need for complex infrastructure setups. Embrace *NoOps* philosophy.

1.4 Methodology

To begin with, an analysis of the existing parallel framework will be made with a sample program that implements this architecture, provided and programmed by the framework code author himself. The architecture is the most relevant part of this project and the use-case implementation provided is just a guideline to perform assessment.
After the analysis, a new architecture will be designed that covers the needs of the cloud. Subsequently, a prototype will be implemented to validate the proposed solution, in addition to proposing a deployment of the architecture as a service in the cloud. It is important that the sample code, already used in production environments, still works after the modifications. Finally, a series of validations will also be carried out to ensure that the architecture meets the objectives of the project.

2 Background

2.1 Cloud computing

Cloud computing is a concept that first appeared in late 1996 [10] but that did not gain traction until a decade later when companies started offering the service open to the public [11].

The main objective of cloud computing is providing on-demand network access to a shared pool of customizable computing resources (e.g., networks, servers, storage, applications, and services) that may be swiftly supplied and released with minimum administrative effort or service provider contact [12]. Databases, web apps, business analytics, e-commerce, computing systems of all sorts are among the services and products available in modern cloud systems.

Cloud computing data centers contain huge reserves of processing power, which means that their benefits lay not only in the wide range of services offered, but also in the ability to scale. In this case, scaling implies that response times remain constant as the number of concurrent users or work increases. Scaling is necessary for mission-critical applications as well as computationally expensive (big data) scientific investigations.

Multitenancy in cloud computing refers to the use of the same computing resources by many customers of a cloud vendor. Despite sharing resources, cloud customers are unaware of one another, and their data is kept separate. It is an important feature of cloud computing; without it, cloud services would be much less useful [13]. Multitenancy allows multiple clients to share the cost of the environment, making their own application less expensive to maintain. All consumers share the costs of maintenance and management. Unlike in a single-tenant environment, where resources are likely to be underutilized, available resources in a multi-tenant environment are utilized to the fullest extent possible since they are shared by several users [14].

The term *on-premises* refers to the usage of a company's own servers and IT infrastructure. The company itself is responsible for the daily operation and maintenance of the system: data security (backups), power-supply, IT professionals, among other necessary elements.

Historically, developers have been increasingly interested in shifting their operations responsibilities to third parties, libraries or open source tools. See figure 1 to examine how focus in business logic increases as resource abstraction increases using current available resource types. It is not by chance that the first compute resources were on-premises and the newest tools are cloud, shifting the responsibility to the cloud provider. Developers also prefer to focus on their own set of use-cases, abstracting logic of unnecessary or unrelated operational elements away.

2.2 Serverless computing

Serverless computing, a cloud-based paradigm for deploying apps and services, provides a step forward in cloud app development, programming models, abstractions, and platforms. Developers don't have to develop auto-scaling policies or describe how system consumption patterns in compute, memory or storage translate to application usage. Instead, they rely on the cloud provider to invest more in resources automatically as demand increases. Consequently, high availability concepts are built in. Developers also expect the cloud provider to look after server maintenance, security upgrades, availability and reliability monitoring [15].

Serverless lowers the bar for developers by requiring them to relinquish all operational complexity and scalability to the cloud provider. This means that the developers require no infrastructure knowledge, reducing friction when creating software for the cloud. This is often referred as NoOps (no operations), in contrast with DevOps, which requires a dedicated

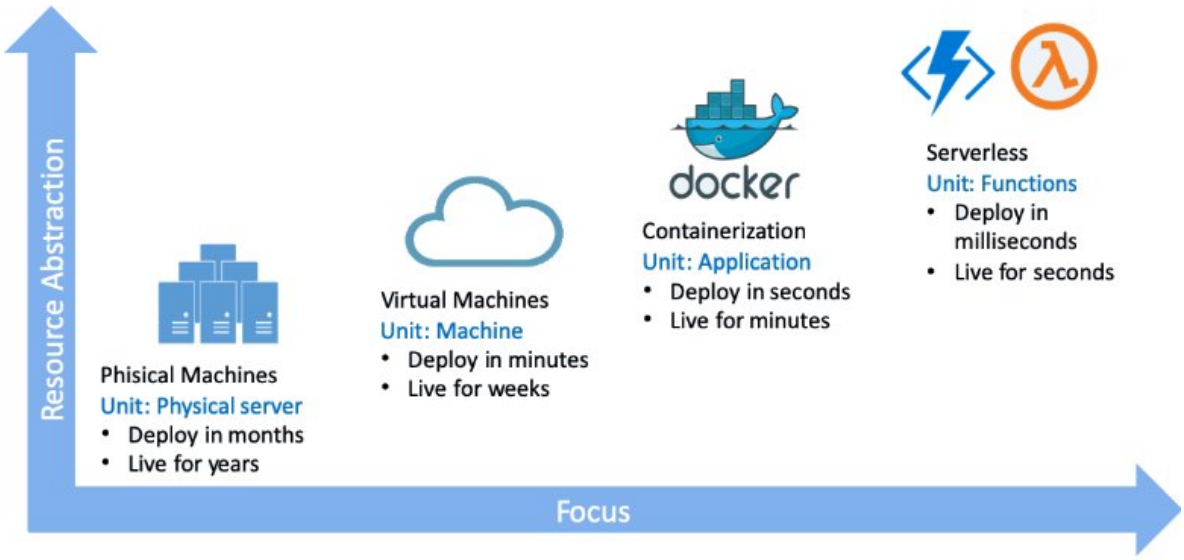

Figure 1: Cloud execution models - focus on business logic / resource abstraction [1]

team with the goal of completely automating the deployment, monitoring and management of applications and the infrastructure on which they run [16].

2.3 Function as a service

Function as a Service (FaaS) is an event-driven execution model that runs in stateless containers and focuses on other services to maintain complex logic and state.

The FaaS model is built on the idea of running function code without having to worry about the underlying infrastructure, like servers or virtual machines. Long-running processes are not the best fit for functions, but they can be operated as a large number of isolated parallel instances that can be launched in a reduced amount of time. As FaaS systems have constraints on memory use or execution time, programs end up having a high level of granularity in task or work division.

The FaaS model provides true pay-as-you-go invoicing (with millisecond precision), billing only for the resources used and avoiding over allocation, preventing idle compute bills [17].

There are multiple cloud FaaS providers such as IBM Cloud Functions, Amazon's AWS Lambda, Google Cloud Functions and Microsoft Azure Functions.

For instance, AWS Lambda functions may be written in a variety of languages and runtime environments, including CSharp.NET, Node.js, Go, PowerShell, Python, Java, and Ruby. There are a few constraints or quotas to consider when using this service, the most relevant ones have been added to Table 1.

Resource	*Quota*
Function memory allocation	128 MB to 10,240 MB, in 1-MB increments.
Function timeout	900 seconds (15 minutes)
Function environment variables	4 KB, for all environment variables in aggregate
Function resource-based policy	20 KB
Function layers	five layers
Function burst concurrency	500 - 3000 (varies per Region)
Invocation payload (request and response)	6 MB (synchronous) 256 KB (asynchronous)
Deployment package (.zip file archive) size	... 250 MB (unzipped) ...
Container image code package size	10 GB
/tmp directory storage	512 MB to 10,240 MB, in 1-MB increments.
File descriptors	1,024
Execution processes/threads	1,024

Table 1: AWS Lambda quotas [5].

As seen in Table 1, serverless functions must have relatively short-lived executions and lightweight environments. This fact allows cloud providers and developers to make assumptions on the burstable nature of FaaS, shifting the long-lived execution applications for other cloud technologies such as containers, virtual machines or bare metal (see Figure 1).

2.4 Cloud storage

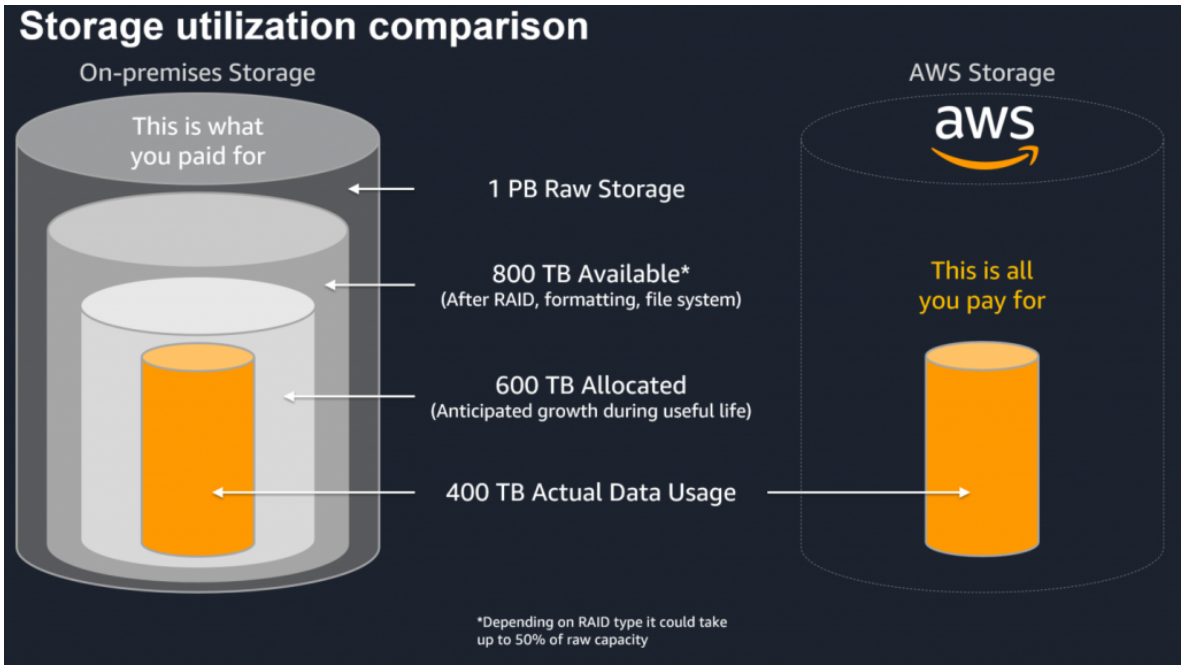

Figure 2: AWS Storage Utilization Comparison [2].

Cloud storage is a more cost-effective and scalable option than on-premise hard drives or storage networks for storing content. The quantity of data that can be stored on a computer

hard drive is limited and users must transfer files to an external storage device when they run out of space. To archive data and files, companies have traditionally created and managed Storage Area Networks (SANs). Storage Area Networks are costly to maintain because as the amount of data stored grows, firms must invest in additional servers and infrastructure to meet the rising demand. See figure 2 to graphically compare on-premises storage and cloud storage.

There are three main cloud storage types [2, 18]:

- **Object Storage**: Objects are a set of a unique identifier, data and metadata. Objects save data in the format in which it is received and allow metadata to be customized in ways that make the data easier to access and analyze. They are stored in a single, broad, flat namespace, with no hierarchy or tree structure like a standard filesystem. The tremendous scalability inherent in object storage systems is enabled by these flat namespaces.
- **File Storage**: A replica of traditional networked file systems (NFS) with a hierarchical file and folder structure. It is designed to expand to petabytes on demand without affecting applications, dynamically increasing and shrinking on data addition or deletion, removing the need for capacity provisioning and management.
- **Block Storage**: Blocks are used by cloud providers to distribute massive volumes of data over several storage servers. Block storage resources, which have low IO latency (the time it takes for a connection between the system and the client) and are especially suited to huge databases or data lakes. They are equivalent in concept to Storage Area Networks.

Object storage has the added benefit of being billed on a temporary basis, providing very cheap storage for short durations of time, see Figure 3. The chart is still relevant in 2022 as the price remains at $0.025/GB per month.

2.5 Available cloud-aware libraries and Lithops framework

Attempting to search for stable libraries to carry cloud functionalities to OCaml did not bring suitable results because of:

1. A low number of contributors.
2. The needed cloud services were not polished or not even in the roadmap.
3. There was not any type of long-term support guarantees.
4. Functionalities were scattered among different libraries (no jack of all trades).

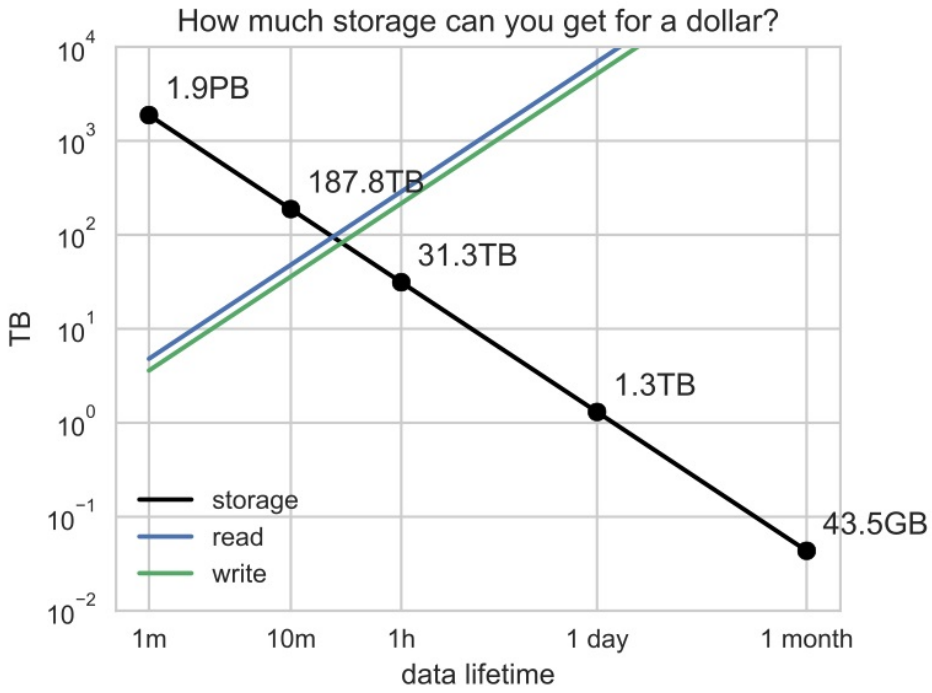

Figure 3: AWS Storage cost effectiveness on a temporary basis [3].

This resulted in the idea of using a wrapper using the freely available Lithops library [19], written in python and backed by the same research group in the University Rovira i Virgili and IBM, already successfully used in other projects.

Lithops allows developers to execute regular python code at massive scale in the cloud, using serverless computing resources and a local client to orchestrate and coordinate them. The processes' dependencies, process function code, and input arguments are automatically detected, serialized, and uploaded to storage by Lithops itself. It simplifies the task of spawning hundreds or thousands of cloud functions to execute a large job in a few seconds from start and offers an adaptable backend architecture (compute, storage) that may be used with a variety of cloud providers as well as on-premise backends. Consequently, the same python code can run in IBM Cloud, AWS, Azure, Google Cloud, Aliyun, Kubernetes, or OpenShift without any changes [19].

Using Lithops allows to easily switch from different cloud services like Lambda to Batch. Lambda is a serverless, compute service that lets run code for virtually any type of application or backend service without provisioning or managing servers for short duration executions. On the other hand, Batch is free serverless managed service that provides the same managed and extra customization benefits, in addition to queuing, usually advantageous for long time running jobs. Having support for different backends, Lithops can be used to dynamically switch from a critical runtime performance environment to a queued batch environment that

does not impose time constraints and can be offered for a much cheaper pricing and a longer execution time.

2.6 Genomics data analysis workflows and cloud

As the number of biomedical research initiatives and large-scale partnerships grows, so does the amount of genetic data created, which today ranges from 2 to 40 billion gigabytes each year [20]. Researchers are attempting to extract useful data from such complex and massive databases in order to better understand human health and illness.

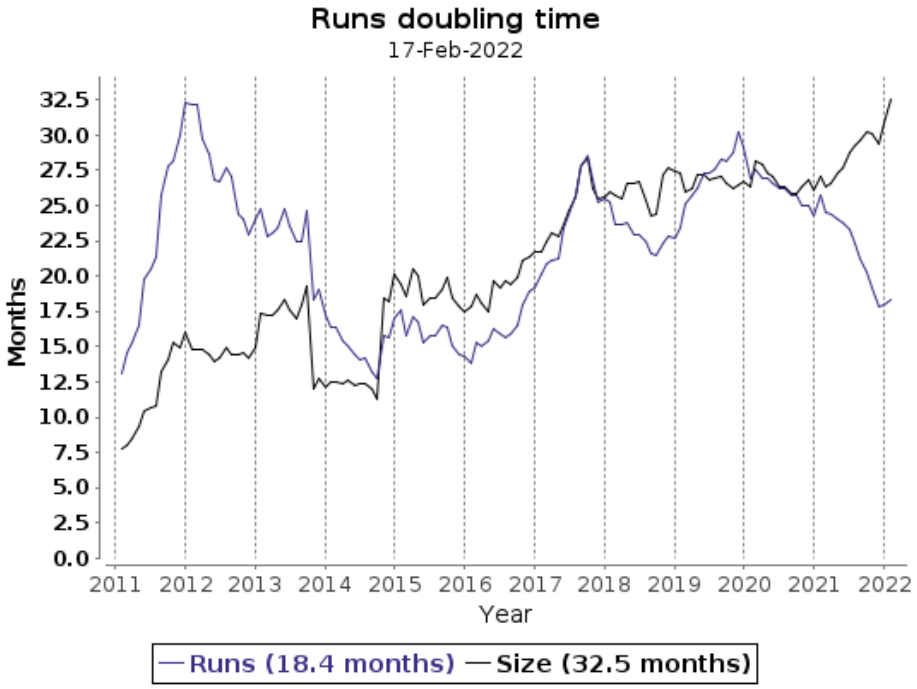

Figure 4: European Nucleotide Archive and Sequence Read Archive doubling time [4].

High-throughput sequencing reads have skyrocketed exponentially for the past decade [21], generating big volumes of genomic data that are saved in publicly accessible archives. These data sets are a valuable resource for researchers investigating critical diseases. See figures 4 and 5.

Some of these databases are automatically moved to cloud storage, accessible directly from the cloud internal network with no extra cost for scientific purposes, well suited for large-scale analysis because of the reduction in the bandwidth required for genomic data delivery. Only highly processed outputs are downloaded, drastically lowering the amount of computer power required for distribution [22].

Labs are forced to rapidly expand their computational resources with no top limit in sight, because sequencing productivity was and is still outpacing Moore's law. According to computational research, Cloud computing could be utilized to examine data straight from the

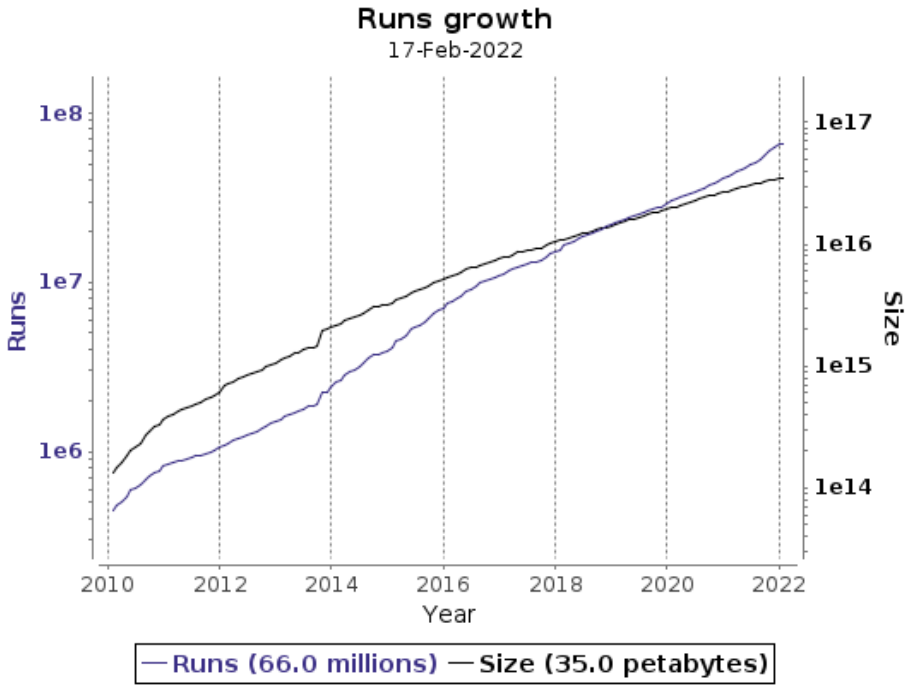

Figure 5: European Nucleotide Archive and Sequence Read Archive growth [4].

sequencer at an acceptable speed and cost [23].
Friction points for the researchers to cloudify their existing applications and potentially new research projects include adapting their complex pipelines and validating them with the new architectures. This process could be simplified if there was a standardized way to move applications to the cloud.

2.7 Moving applications to the cloud

Moving applications to the cloud provides a lot of benefits that are impossible to have when deploying software on-premises. However, the cloud has different layers of abstraction and it is important to consider them depending on the application to develop. It is different to move a dependency-less functionality into serverless than to move a monolithic application with many dependencies at the operating system level. For the first case, it would be wise to use a serverless-based solution as the developer and the application have no constraints on the underlying software, whereas in the second case, it is unwise to explore a serverless solution as execution environment is not as flexible (vendor choice, non-customizable) as it would be in a virtual machine.

Virtual machines provide the developers more flexibility in exchange for more infrastructure responsibility. Thus, it is a tradeoff between responsibility or customization and a turn-key solution rewarded by the cloud provider. This dilemma makes moving applications to the cloud a complex problem and hard to generalize. Furthermore, there are difficulties

in porting the same high-level software layer, such as different programming languages and the availability of vendor toolsets to provide deployment ease for their services. Various approaches have been proposed to model this problem as an input/output issue. Making it an input/output problem boosts adaptability for not inner logic but data only. Other branches in science have long studied this term as "Black box system" [24].

For instance, there is a research project that tries to adapt the Unix shell for serverless architectures [25]. The Unix shell bases its workings on reading and writing from executable binaries independently of their programming topology or language, only based on their input and output. Using this idea, the serverless shell runs shell scripts on a serverless platform much like with a regular computer, built around a small set of components that includes a new inter-process communication layer for serverless.

Another example is Nextflow, a reactive workflow framework using a custom *Domain Specific Language* [26] based on the dataflow programming model [27] to simplify writing complex pipelines. In practice, a Nextflow pipeline script is made by joining together different processes. Each process can be written in any scripting language that can be executed by the Linux platform (Bash, Perl, Ruby, Python, etc.). Processes are executed independently and are isolated from each other, i.e. they do not share a common (writable) state. The only way they can communicate is via asynchronous FIFO queues, called channels. Any process can define one or more channels as input and output. The interaction between these processes, and ultimately the pipeline execution flow itself, is implicitly defined by these input and output declarations.

3 Architectural breakdown and proposed changes

3.1 The big picture

Originally, the code is single-machine multiprocessing. Having no need for network stack, it uses Unix fork to spawn processes and Unix pipes to communicate different processing stages.

There are three main distinct prototype functions that must be implemented for each use-case (business logic):

1. The **chunking** function reads the input and verifies its integrity before distributing the chunks to the worker functions. It is directly connected to all the workers.
2. The **map/worker** function is directly connected to the chunking function to retrieve work and also directly connected to the reducer, in order to send the processed data. Multiple invocations of this function are done in parallel, using different processes and retrieving different workloads from the same chunker. Two workers will never process the same chunk even when running concurrently.
3. The **reduce** function gathers workers processed chunks, preserving order and optionally saving state to generate final outputs.

See figure 6 to appreciate the information flow between the three functions.

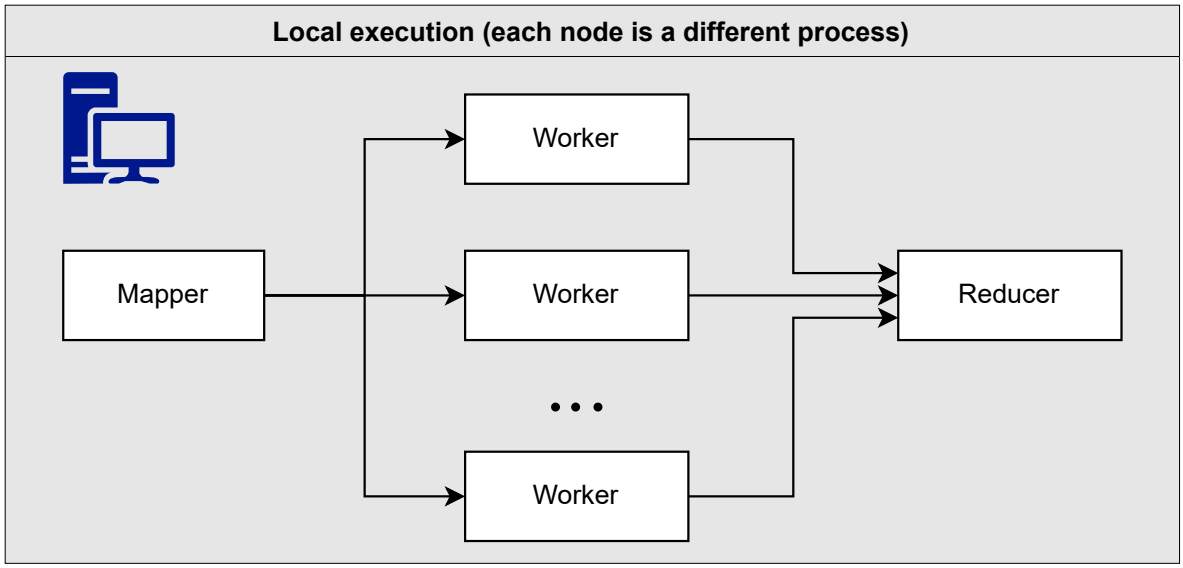

Figure 6: Original local execution architecture

3.1.1 One-way message passing workflow

The most important rule of any implementation of this framework is to keep relevant work passed through the communication pipes. Thanks to serialization, any type of information

can be sent as a message.

However, a message must be generated, processed, and reduced, being these three steps the main implementation functions. There is a constraint in the flow of information of the implemented functions. The chunk always ends up being reduced as the last step and generated as the first step. The order cannot be inverse. In other words, implemented functions do not have a way to communicate with other functions directly inside the framework, other than following this flow, although it is not problematic for the embarrassingly parallel use cases.

The framework pipes use a custom stop-and-wait flow control. Each process sends control bytes to start and stop reading the pipes at their ends, avoiding blocking the process when they are halted by the OS if the pipe memory buffer is full when multiple data blocks are sent. Also, the chunk size can be controlled by a parameter that is given to the chunker, so the pipe is never blocked with a single big chunk.

3.1.2 Program state

The parallel framework is very flexible because it does not apply constraints on the user code that is run before it is launched. However, it does not protect the programmer from the risk of writing code that may not work as intended if communication between functions happens outside the framework communication mechanism.

On one hand, the worker and reducer can be stateless (implementation decision). On the other hand, the chunker reads from a file descriptor and consumes it, meaning that it is not stateless by itself and must handle the end of file, propagating it to the rest of the program. However, even if two of the functions are stateless, the program is allowed by the framework design to keep state in global variables, accessible internally by each of the function implementations on their own. Ultimately, it is up to the developer to use global variables or not. This will have implications later in the cloud migration process caused by separating the execution of the reducer and the workers in different nodes.

As communication is done mostly through Unix pipes, there are several consequences that must be considered when dealing with the framework:

(1) There is no integrated shared memory mechanism.

(2) Forking processes allows leveraging virtual memory and Copy-On-Write kernel mechanisms to share variables from the parent process.

(3) Message passing arbitrary structured data between processes is possible thanks to serialization.

Point (3) has special significance in flexibility. OCaml being able to serialize most of its structured and primitive types, the sample program is able to send a tuple of a line counter

and an arbitrary buffer type through pipes. There is always a single chunking instance and a single reducer instance, but an indefinite number of workers. The number of workers must be explicitly provided to the framework by the implementation code.

In the sample program, the number of workers is parsed from a command-line argument outside of the parallel framework, near the entry point of the main program when all of the other parameters are parsed and bound-checked. Arguments are usually implemented as global variables, later accessible using the *Copy-On-Write* (COW) virtual memory from the Unix forking (memory is originally unique and arguments are shared to all forked processes). However, attributable to the COW nature of this memory, any modification done after the fork is not communicated to the other processes and implies that a copy of the memory is done for the modified process.

3.2 Adapting the ocaml framework with a Lithops python wrapper

As Lithops is especially suited for highly parallel programs with little or no need for communication between processes, our use case is adequate due to only requiring communication between different phases of the execution (map and reduce, for which cloud storage can be used) and not direct communication among nodes.

Moving to the cloud should be as transparent as possible, with minimal code changes so that less code paths have to be tested again and the working logic is intact. In addition, function implementations have to be mostly treated as black-box logic because they should still work after the cloudification process.

The OCaml framework extensively uses fork side-effects to distribute initialization parameters to processes. In our case, fork does only work inside the same function call environment, isolated from the rest. Therefore, we need to take into account that programs require deterministic parameters to be initialized in the same state even if they are run across different execution nodes. This brings us to the importance of replicating the execution using the same launch parameters across all nodes. If this was not the case, each node could take different code paths and the intermediate results would not be equivalent for different nodes with the same task.

3.2.1 Inter-Process Communication

There is no need to replicate or create a protocol to communicate Lithops with the OCaml binary if we can reuse Inter-Process Communication (IPC) ideas. Libraries such as Apache Arrow [28], which has been used previously in genomics, could be a good fit here. However, we do not want to deal with the implications of having to rewrite a different communication mechanism / structural format for each of the programs that would be adapted to the framework (different use cases) and the Lithops wrapper. We can simplify it the most, with

some restrictions/constraints reflected on the framework Application Programming Interface so that the developer is aware of those.

Reading and writing from a pipe is impossible to do single-threaded with a considerable amount of data unless non-blocking reading and writing are used, or a flow control mechanism is used. If synchronous reading and writing is used, the process has the risk of being hung up by the kernel waiting for the other end of the pipe to be read. If the other end of the pipe is never read for any reason, a deadlock occurs and the program is blocked forever. A common reason for both sides being blocked is that one end waits for the other to send all the data but does not consume it. There is a pipe buffer limit parameter that can be increased but is not effective in this case because it does not solve the problem for single huge chunks of data.

Non-blocking mechanisms include both:

(1) An event-loop. For instance, libuv or asyncio.

(2) A tight loop with an asynchronous kernel function (polling).

Using the asyncio python framework we should be able to read and write to and from pipes using a single thread asynchronously from the python wrapper without complex protocols, not to be bothered about thread overheads or the Global Interpreter Lock when reading pipes. Most work is I/O so CPU load is well distributed, reading and writing to pipes is a good use case for an event loop. In addition, other concurrent steps such as the multipart upload (see section 3.5.2) can be integrated into the same.

```python
async def async_subprocess_io_multipart(storage, ocaml_diff_role, result_key, write_stdin_function):
    storage_conf = lithops.Storage().storage_config['aws_s3']
    aiob_session = aiobotocore_sess.get_session()
    async with aiob_session.create_client('s3', storage_conf['region_name'],
                                          aws_secret_access_key=storage_conf['secret_access_key'],
                                          aws_access_key_id=storage_conf['access_key_id'])
                                          as multipart_client:

        multipart_writer = MultipartUploader(multipart_client, storage.bucket, result_key)
        await multipart_writer.setup()

        async def read_stdout(stdout):
            while True:
                buf = await stdout.read(MultipartUploader.PART_SIZE)
                if not buf:
                    break
                await multipart_writer.write(buf)
            await multipart_writer.flush()

        async def read_stderr(stderr):
            while True:
                buf = await stderr.read(2*1024*1024)
                if not buf:
                    break
                print(buf)

        cmd = ' '.join(args.program_and_args)
        print(f'To execute: \'{cmd}\'')
        p = await asyncio.create_subprocess_shell(cmd,
                                                  stdin=asyncio.subprocess.PIPE,
                                                  stdout=asyncio.subprocess.PIPE,
                                                  stderr=asyncio.subprocess.PIPE,
                                                  env=dict(
                                                      os.environ, DIFF=ocaml_diff_role),
                                                  cwd="/tmp")

        await asyncio.gather(
            read_stderr(p.stderr),
            read_stdout(p.stdout),
            write_stdin_function(p.stdin))

        await p.wait()

        if p.returncode != 0:
            raise RuntimeError(f"Return code: {p.returncode}")
```

Code 1: Python wrapper map and reduce event loop workflow code.

In code section 1, a helper function that defines the async reading functions to retrieve the outputs from standard error and standard output in the executed binary (subprocess) is shown. This function also initializes a multipart uploader class (see section 3.5.2) which is later used to upload the outputs to cloud storage.

3.3 Chunker

We assume that the previous step in the pipeline saves the input file in cloud storage, be it a sequencer that uploads the sequenced data directly to the cloud or a preprocessing step.

3.3.1 Problem statement

The chunker may be adequate when processing local data because it is mostly I/O bound as it is always reading, up to the limits of the hard disk speed, there is no need to parallelize it. Also, being a separate process, the chunker can always be streaming data unless workers are all busy (if the implemented business logic is CPU bound), and the communication costs are relatively low (RAM and disk access order of magnitude).

In contrast, it is inconvenient to chunk sequentially when processing data in the cloud. That is because the cloud is prepared for much higher aggregate bandwidth and parallelization levels, being able to read different sections of the files in cloud storage at the same time.

It is also fundamental to partition based on cloud storage and not using a local chunker outside the cloud network (using an internet gateway) as it could create a bottleneck based on the client available bandwidth and packet loss while uploading the chunks to the workers.

All of this implies that the chunker is a bottleneck in the cloud architecture and would not scale to thousands of nodes or huge files, losing the benefits of aggregate throughput and scalability.

3.3.2 Solution - Byte-range regular-expression partitioner

Having an embarrassingly parallel framework, it would be best to implement a byte-range partitioner or structured data format with random access, with the ability to read in parallel from the cloud storage different chunks.

Byte-range partitioners are one of the fastest ways to split work among workers. In this case, a certain amount of extra data is appended at the end of each range but it is small enough not to have any performance penalty. Also, the partitioner is generalized using regular expressions allowing adaptation to any other unstructured format that can be parsed, to avoid duplicates and to avoid invalid data. This healing using regular expressions happens entirely on the map/worker phase, which means that this type of partitioner can be made 100% parallel.

The algorithm for the partitioner is:

1. Generate a set of evenly sized byte ranges so that the file is entirely split by them. The last byte range (start and end offsets) may accommodate and have an inferior size than the others.

2. Append a predefined number of bytes at the end of each range (bytes that overlap with

the start of the next range). The last range cannot do this, nor it is needed. See figure 7.

3. Invoke a function call for each byte range.

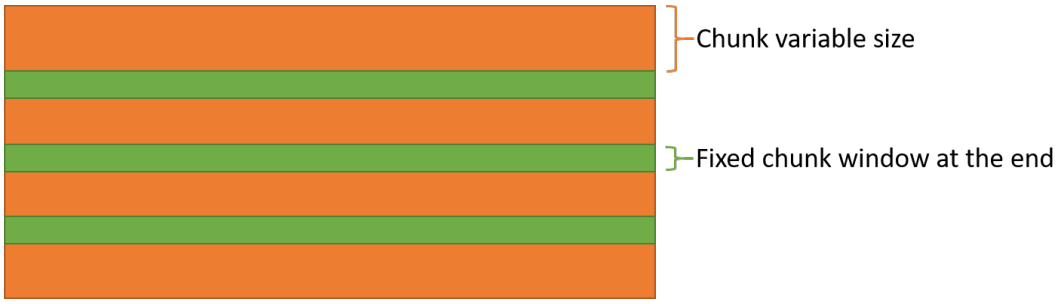

Figure 7: Chunk window/threshold. Byte range partitioner.

The 1st and 2nd steps can be seen in code section 4 for a local OCaml implementation replicating Lithops byte-range creation.

After the partitioner invokes the function call, the following happens:

1. The Lithops wrapper map implementation code reads the byte range entirely to memory and pipes it into the standard input of the OCaml binary.

2. The OCaml code seeks the first match of the provided regular expression for that format. When the first match is found that position is saved as the start of the real chunk, discarding anything before it. See code 2.

3. The OCaml code seeks the first match of the same regular expression but this time at the end of the chunk, just before the extra window at the end. The matched position is the end of the real chunk, it may be inside the extra window at the end. See code 2.

4. Now the chunk is already healed, entirely in memory and the local chunker may distribute the work among the worker processes.

```
let heal_chunk_with_regexp
    ((buf : string), (chunk_size : int), (element_delimiter : Str.regexp)) =
  (* Lithops adds the last byte from the previous chunk in the first byte in the buffer *)
  (* except for the first chunk (which has no previous chunk) *)
  (* However, right now, matching single chars is NOT supported. *)
  let first_elem_start =
    try Str.search_forward element_delimiter buf 0
    with Not_found ->
      Misc.teprintf "Errored string start:%s...\n%!" (Str.first_chars buf 250);
      invalid_arg "Impossible to heal the function. First element not found."
  in
  let string_buf_size = String.length buf in
  if string_buf_size >= chunk_size then
    let last_elem_end =
      try Str.search_forward element_delimiter buf chunk_size
      with Not_found ->
        Misc.teprintf "Errored string end + 100 threshold chars:...%s\n%!"
          (String.sub buf first_elem_start
             (string_buf_size - first_elem_start + 100));
        invalid_arg "Impossible to heal the function. Last element not found."
    in
    String.sub buf first_elem_start (last_elem_end - first_elem_start)
  else
    (* If the last chunk is smaller than the others just return until the end *)
    String.sub buf first_elem_start (string_buf_size - first_elem_start)
```

Code 2: OCaml chunk healing using regular expressions

The regular expression acts as a fundamental element / unit of information detector rather than a separator and should match the entire element. This way, sequence integrity (no duplicates or read errors) among subsequent ranges is ensured (thanks to the added window at the end of the chunk). Still, it is paramount that the regular expression is correct.

Taking the start of an element using a regular expression ensures that the start of the element is not split in half because the regular expression should match the next element in case that happened. If this was the case, the previous chunk would have taken this element using the added threshold at the end, and the current chunk would not have to process the same element (no duplication, random cutting protection). One limitation of this procedure is when the chunk size is smaller than the element size. For example, a 1 KB chunk size of a 1 GB element would never work with this partitioner. Nonetheless, other partitioners can be implemented for formats where element size is random or not guaranteed to be of an estimated size.

Two example regular expressions are provided for two different formats: FASTQ [29] and MAP single end reads [30]. See Code 3.

```
module ElementDetector = struct
  let fastq_re = Str.regexp "^@.*\n*.+\n\\+.*\n.+$"
  and map_se_re = Str.regexp "^[A-ZO-9]+\\.[O-9].*$"
end
```

Code 3: Partitioner regular expression healing element detectors

To avoid having Lithops as a dependency when developing locally, a OCaml partitioning scheme that mimics the Lithops partitioner with local files was also developed (byte range generation) and added to the framework toolset. It does not require any python interpreter to run. See Code 4.

```
let partition_input_file filename chunk_threshold obj_chunk_size =
  let stat = Unix.stat filename in
  let start_offset = ref 0 and ranges = ref [] in
  (* create ranges with fixed size except for the last one *)
  while !start_offset < stat.st_size - obj_chunk_size do
    let range =
      (!start_offset, !start_offset + obj_chunk_size + chunk_threshold)
    in
    ranges := range :: !ranges;
    start_offset := !start_offset + obj_chunk_size
  done;
  (* add the last exact range to cover until the end of the file *)
  ranges := (!start_offset, stat.st_size) :: !ranges;
  List.rev !ranges

let byte_range_stream_from_file filename ranges =
  let file_chan = open_in_bin filename in
  Stream.from (fun range_index ->
      if List.length ranges <= range_index then None
      else
        let start_offset, end_offset = List.nth ranges range_index in
        seek_in file_chan start_offset;
        Misc.teprintf "Range start %d, end %d\n" start_offset end_offset;
        let chunk_size = end_offset - start_offset in
        Some (really_input_string file_chan chunk_size))
```

Code 4: OCaml partitioner emulation (local)

3.4 Worker / Map

3.4.1 Problem statement

Map functions process different chunks of the same problem in parallel using Unix processes. The framework processes have no hard dependency or requirement on the OS layer other than being based on Unix, so there is no need to take into account underlying infrastructure: that is the perfect use case for serverless.

No shared state implies avoiding most of the burden required to orchestrate the parallelism and allows completely using the Function as a Service mindset, based on serverless principles.

The worker / map stateless functional architecture is rather positive for our migration to the cloud because a process is not the computational unit of a FaaS, but a function is. Unfortunately, scaling in multiple vCPUs is still necessary on cloud functions.

3.4.2 Solution - Preserving multiprocessing

One might be tempted to think that with the original architecture, the problem solution relies on substituting the Unix pipes with the cloud storage directly. That is not the case, preserving the local multiprocessing code is important to reap the benefits of vCPU scaling, which solely depends on the memory runtime chosen for the cloud function call (if running inside a virtual machine instead of cloud functions, it is also useful). The higher tiers provide three, four, five and up to six CPU cores. See Table 2.

The local chunker in the OCaml binary will be modified to ingest the whole chunk at once and it will distribute the chunk to the local worker processes. This way, Unix pipes are still used in the local execution inside the function to scale on vCPU usage and avoid being billed for idle compute.

Memory	*vCPUs*	*CPU Ceiling*
832 MB	2	0.50
1769 MB	2	1.00
3008 MB	2	1.67
3009 MB	3	1.70
5307 MB	3	2.39
5308 MB	4	2.67
7076 MB	4	2.84
7077 MB	5	3.86
8845 MB	5	4.23
8846 MB	6	4.48
10240 MB	6	4.72

Table 2: Experimental analysis of AWS lambda vCPU scaling [6].

A multi-threading function configured at 3009 MB might run 1.5x as quickly as a function designed at 3008 MB, saving 33.3 percent execution time [6]. In addition, AWS offers AWS Cost Optimizer to automate a cost-performance analysis for all the Lambda functions in an AWS account. This service evaluates functions that have run at least 50 times over the previous 14 days, and provides automatic recommendations for memory allocation [31].
If a particular use-case required a single process/worker, with the proposed architecture, the same code would still properly operate with no modifications needed.

On the new architecture, while executing the worker role, the local reducer gathers the local multiprocess worker processes outputs as if it was to reduce them, but instead, it yields the serialized data to standard output. Meanwhile, the Lithops wrapper consumes from this standard output pipe and uploads data in chunks to the cloud storage bucket.

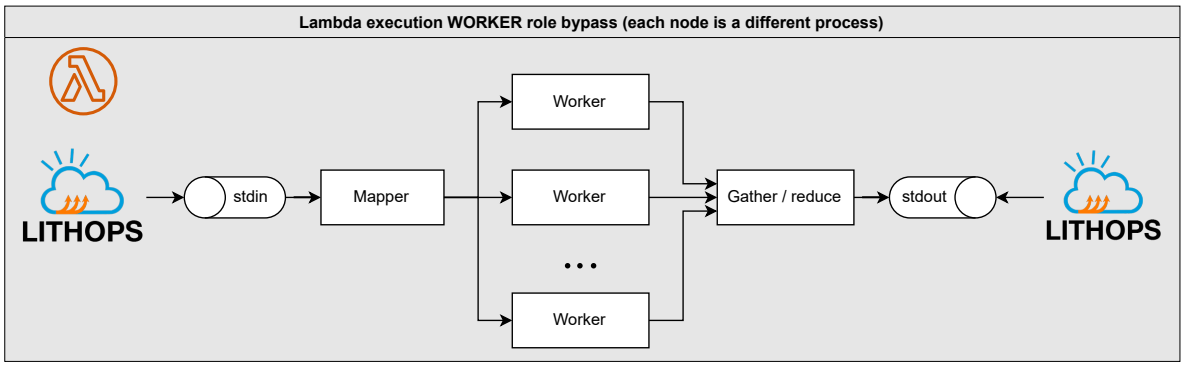

Figure 8: Worker using Lithops wrapper

The redirection of data from the reducer process to stdout is required instead of reducing directly. It is not guaranteed in all cases that a local (sample) reduction is equivalent to a global (population) reduction.

```
def run_worker(obj, storage, args):
  download_binary(storage, args)

  async def write_stdin(stdin):
    while True:
      chunk = obj.data_stream.sb.read()
      if not chunk:
        break
      stdin.write(chunk)
      await stdin.drain()
    stdin.close()
    await stdin.wait_closed()

  session_id = os.environ.get('__LITHOPS_SESSION_ID', '')
  result_key = f'intermediate-results/{session_id}'

  asyncio.run(async_subprocess_io_multipart(storage, 'worker', result_key, write_stdin), debug=True)

  return result_key
```

Code 5: Lithops wrapper map/worker code

Code section 5 shows the Lithops map function code. First of all, the compiled binary to execute is downloaded from cloud storage (the path is given as a parameter to the script). After the binary is downloaded, the writing to standard input is defined in a function and an intermediary result key is generated. The function that defines the async writing is then passed into the function that the asyncio main loop will run (see code section 1).

3.5 Reducer

3.5.1 Problem statement

The reducer is the biggest bottleneck in the whole architecture. By design, it is not parallelized, launching only a single process for each program execution. This is detrimental to any form of possible scalability in the modern era, as only vertical scalability on single core (clock speed, cache) would yield better results, which is an outdated approach that stopped giving meaningful performance gains since 2005 when multi-core systems started to dominate [32]. Furthermore, it is common for the cloud providers to increase the number of available vCPUs before a whole uncapped vCPU is assigned to the instance (vCPU ceiling) [6]. Consequently, this prevents the sequential reducer to run as fast a possible, while also overprovisioning a new vCPU that is billed but unused. This scaling factor is not only applied to compute but also memory, storage and network bandwidth, creating bigger room for overprovisioning if one of the previous factors is required to increase throughput.

There is another problem, the input (download) for the worker and the reducer (a single intermediate output chunk) is always loaded in memory because we can control the chunk size and function memory (up to 10 GB), but that is not the case for the upload as the output file may be bigger than the available memory.

3.5.2 Partial solution - Multipart uploads (streaming)

We are not able to improve the reducer performance because of genericity constraints: it is out of the scope for this thesis to improve performance at the cost of losing the capacity for this reducer to work in all sorts of problems. However, as a future work, a distributed reducer architecture and or other design ideas should be applied in order to minimize the damage done to the run wall-clock time.

In the situation of the sample code use-case, a binary reduction could be performed so chunks would be reduced in pairs, speeding up the reduction process.

An approachable solution to save the output file could be using Elastic File System storage to temporarily save the file, but it requires extra configuration, uses network bandwidth, and Lithops does not support it by itself. In contrast, a multipart upload allows aggregating a single object as a set of parts. Each part is a continuous portion of the object's data and can be uploaded independently or disordered. This feature has a lot of benefits, but we are only interested in a few of them, such as the ability to begin uploading an object without knowing the final size. This allows streaming the output of the workers and the reducer to the cloud storage without configuring a temporary disk. Nevertheless, it is important to be aware of its limitations. For instance, Lithops does not support multipart upload in the way we require it.

The part size has an important constraint (among others that are not so relevant for the

problem): parts require a size of 5 MiB to 5 GiB. There is no minimum size limit on the last part uploaded. With this in mind, we must ensure that we do not use multipart upload for small files. In this case, an estimation of the expected output given a chunk size should work to approximate if the chunk is too big or too small to give results inside this range. Otherwise, for the small files, a buffer can be used and if this buffer is filled with data smaller than 5 MiB then a normal upload is used instead.

What is interesting to us in this case is streaming sequential parts. Uploading all intermediate outputs into a single file would require a coordinator if there was the possibility that multiple parts were smaller than 5MiB. Instead, saving each mapper's output to different files, aggregating these paths to a list, providing the reducer with the list and the reducer fetching each file individually in order, is the best option for simplicity. No coordinator is needed for this and functions still benefit from streaming (although to separate files) in the framework's architecture.

The library used for this task is called *aiobotocore* which exposes an async API for the *botocore* AWS library. In this case, no extra configuration is required other than installing the dependency using the *pip* package manager. A simple modification is done to the code so that Lithops hands the credentials used to the *aiobotocore* client and no extra authentication must be set up.

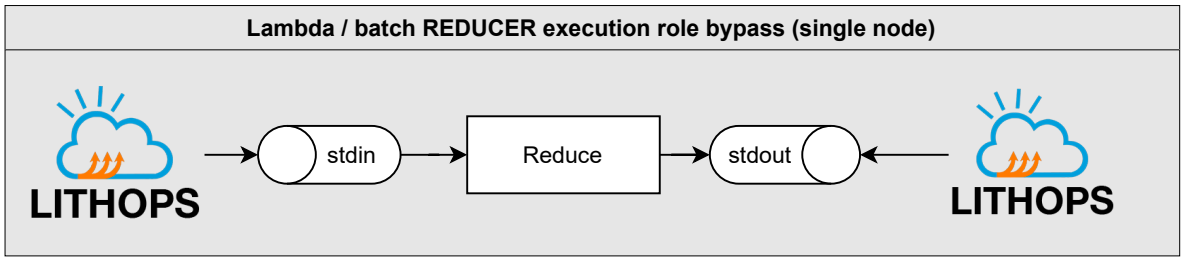

Figure 9: Reducer using Lithops wrapper

Now the Lithops wrapper gathers all the chunks saved by the workers from cloud storage, preserving order and piping them into the standard input of the OCaml binary one after the other.

```python
def run_reducer(results, args, storage):
  download_binary(storage, args)

  async def write_stdin(stdin):
    for cloudobject_path in results:
      partial_result = storage.get_object(storage.bucket, cloudobject_path)
      stdin.write(partial_result)
      await stdin.drain()
    stdin.close()
    await stdin.wait_closed()

  session_id = os.environ.get('__LITHOPS_SESSION_ID', '')
  result_key = f'results/{session_id}'

  asyncio.run(async_subprocess_io_multipart(storage, 'reducer', result_key, write_stdin), debug=True)

  # Delete map outputs, cleanup
  for cloudobject_path in results:
    storage.delete_object(storage.bucket, cloudobject_path)

  return f'lithops storage get {storage.bucket} {result_key} -o RESULTFILENAME'
```

Code 6: Lithops wrapper reduce code

Code section 6 shows the Lithops reducer function code. First of all, the compiled binary to execute is downloaded from cloud storage (the path is given as a parameter to the script). After the binary is downloaded, writing to standard input is defined in a function and a final result key is generated. The function that defines the async writing is then passed into the function that the asyncio main loop will run (see code section 1). The code flow resembles that of the map function but the standard input writing is more complex because it has to read all of the chunks (instead of a single one) from cloud storage and write them into the binary.

The output from the reducer is uploaded to a single file. However, the OCaml reducer binary is free to use any available cloud storage libraries to write an arbitrary number of files to cloud storage directly by itself (instead of using the standard output only).

3.6 Program execution flow

Instead of the chunker directly reading from files using OCaml standard library, it should receive its input via standard input as a string and scatter it to its child processes.

The redirection of the local reducer input to standard output when running in a map function comes at a cost: a code refactoring/constraint. The wrapped binaries must decide when to take the reducer code path that redirects the input to standard output and when to reduce the input directly (do the task of the actual reducer implementation).

A cheap and quick way to differentiate when to execute a worker or reducer is using

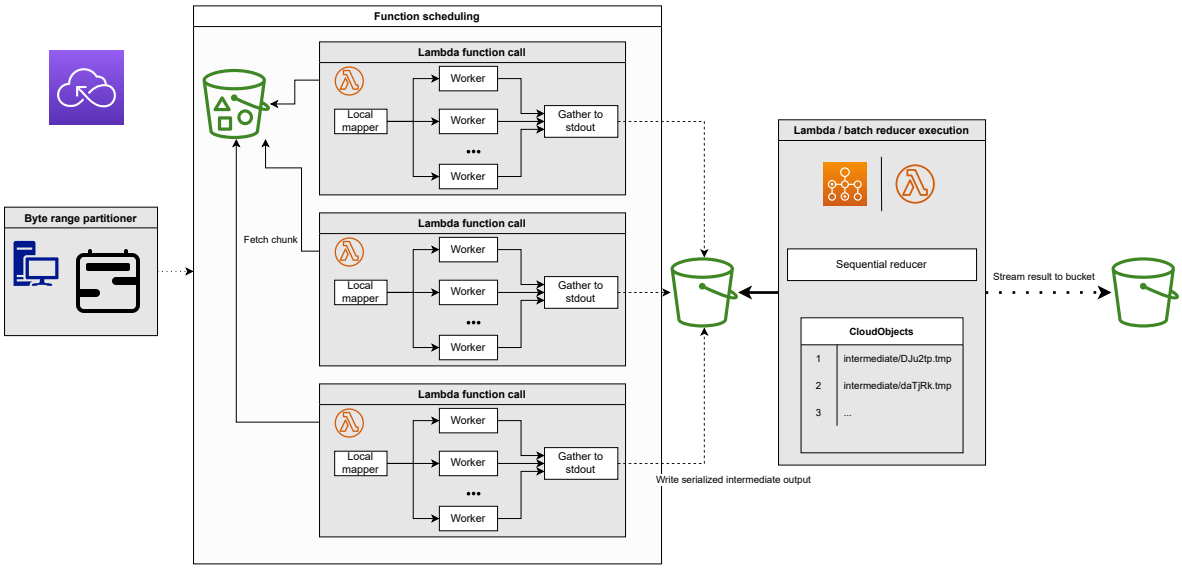

Figure 10: Cloud framework architecture

environment variables, set by the Lithops map and reduce implementation functions to differentiate code paths to run depending on where the same binary should take the role as a reducer or a worker. See code section 7 for an example of the worker map function, highlighted in boldface.

Environment variables can be queried from any binary independently of the programming language used, providing language transparency in the mechanism that the wrapper uses to communicate which code path is to be run (worker or reducer). See code section 8 and code section 9 for the OCaml handling code of environment variables.

```
1 p = await asyncio.create_subprocess_shell(cmd,
2                                           stdin=asyncio.subprocess.PIPE,
3                                           stdout=asyncio.subprocess.PIPE,
4                                           stderr=asyncio.subprocess.PIPE,
5                                           env=dict(
6                                               os.environ, DIFF='worker'),
7                                           cwd="/tmp")
```

Code 7: Differentiating environment variable set from Lithops functions. Snippet from code section 1.

```
if Sys.getenv_opt "DIFF" <> None then (
  (* Worker role: output the received chunks from the workers directly to stdout *)
  output_value o_2_stdout data;
  flush o_2_stdout)
else
  (* Worker role not set: reduce directly *)
  reduce data;
```

Code 8: Reducer function input redirection path inside the framework to stdout depending on the environment variable.

```
match Sys.getenv_opt "DIFF" with
 | Some "reducer" -> (
  (* Reducer is single threaded *)
  let w_2_o_pipe_in = ref Unix.stdin in
  let w_2_o = Unix.in_channel_of_descr !w_2_o_pipe_in in
  try
  while true do
    let data = (input_value w_2_o : 'b) in
      reducer data
  done
  with End_of_file -> ())
 | None | Some _ -> (* start chunker and workers *)
```

Code 9: Single threaded reducer execution outside the OCaml framework functions depending on the environment variable.

3.7 Ocaml API changes

Code section 12 displays the modified framework entrypoint function. For reference the original function code section 10 changed to code section 11 in the adaptation to a Lithops wrapper environment.

```
let process_stream_chunkwise ?(buffered_chunks_per_thread = 10)
    (f : unit -> 'a) (g : 'a -> 'b) (h : 'b -> unit) threads
```

Code 10: OCaml framework original main function declaration.

```
let process_stream_chunkwise_with_lithops ?(chunk_size_in_mb = 32)
    (filename : string option) (elem_detector : Str.regexp)
    (chunker : string -> 'a) (worker : 'a -> 'b) (reducer : 'b -> unit)
```

Code 11: OCaml framework adapted main function declaration.

```
let process_stream_chunkwise_with_lithops ?(chunk_size_in_mb = 32)
    (filename : string option) (elem_detector : Str.regexp)
    (chunker : string -> 'a) (worker : 'a -> 'b) (reducer : 'b -> unit)
    threads =
  let chunk_size = chunk_size_in_mb * 1024 * 1024 in
  (* The threshold is a lithops constant *)
  let chunk_threshold = 128 * 1024 in

  if Option.is_some filename then
    (* Lithops is not running. We want to partition a *file* simulating lithops. *)
    let filename = Option.get filename in
    let ranges = partition_input_file filename chunk_threshold chunk_size in
    let stream = byte_range_stream_from_file filename ranges in
    let healed string_buf =
      heal_chunk_with_regexp (string_buf, chunk_size, elem_detector)
    in
    let execute_par string_buf =
      process_stream_chunkwise
        (fun () -> chunker (healed string_buf))
        worker reducer threads
    in
    Stream.iter execute_par stream
  else
    (* Filename not set, read from stdin only. *)
    match Sys.getenv_opt "DIFF" with
    | Some "reducer" -> (
        (* Reducer is single threaded *)
        let w_2_o_pipe_in = ref Unix.stdin in
        let w_2_o = Unix.in_channel_of_descr !w_2_o_pipe_in in
        try
          while true do
            let data = (input_value w_2_o : 'b) in
            reducer data
          done
        with End_of_file -> ())
    | None | Some _ ->
        (* Assume receiving a single chunk *)
        let read_stdin_into_buf buffer_size =
          let input_buffer = Buffer.create buffer_size in
          try
            Buffer.add_channel input_buffer stdin buffer_size;
            input_buffer
          with End_of_file -> input_buffer
        in
        let read_stdin = read_stdin_into_buf (chunk_size + chunk_threshold) in
        let healed =
          heal_chunk_with_regexp
            (Buffer.contents read_stdin, chunk_size, elem_detector)
        in
        process_stream_chunkwise
          (fun () -> chunker healed)
          worker reducer threads
```

Code 12: OCaml framework main function.

4 Validation

The purpose of this section is to:

- Highlight streaming from memory to the cloud storage without configured temporary storage.
- Emphasize on the throughput gained from cloudifying the map phase.
- Demonstrate how the performance scales as the input file gets bigger.

The speedup is mostly expected on the Map phase as a result of the Map function completely running in parallel without communication between nodes (embarrassingly parallel), being the Reduce phase equal or worse than in a single-machine setup (sequential). Leveraging serverless functions for the map function, there is the benefit of aggregate bandwidth and scalability, whereas the reducer can just use a virtual machine or a cloud function if the file is small enough.

Tests have been prepared to generate throughput and speedup plots for the map phase, using variable volumes: 438 MB (omitted from this section, only functional testing), 4 GB and 100 GB files. The files are saved in S3 cloud storage and the Lithops coordinator runs from a laptop using wireless outside the cloud internal network. To verify integrity after processing the files, a checksum comparison is made with the single-machine code output and the cloud version output.

4.1 Experiments

As baseline for testing single-machine performance we have processed both 4 GB and 100 GB files through a *c6id.2xlarge* instance. C6i instances are powered by 3rd Generation Intel Xeon Scalable Ice Lake processors. This instance in particular has 8 vCPUs, 16 GB of memory, up to 12.5 Gbps of network bandwidth and a 474 GB NVMe SSD. The input files have been downloaded from S3 to the hard disk to speed up the reading phase. However, the outputs are directly streamed to S3.

Figures 11 and 12 show that both files get similar speedup values, even when one is x25 times bigger sized than the other. In comparison, Figure 13 displays how using serverless functions and their aggregate bandwidth can scale better for huge files thanks to a parallel architecture.

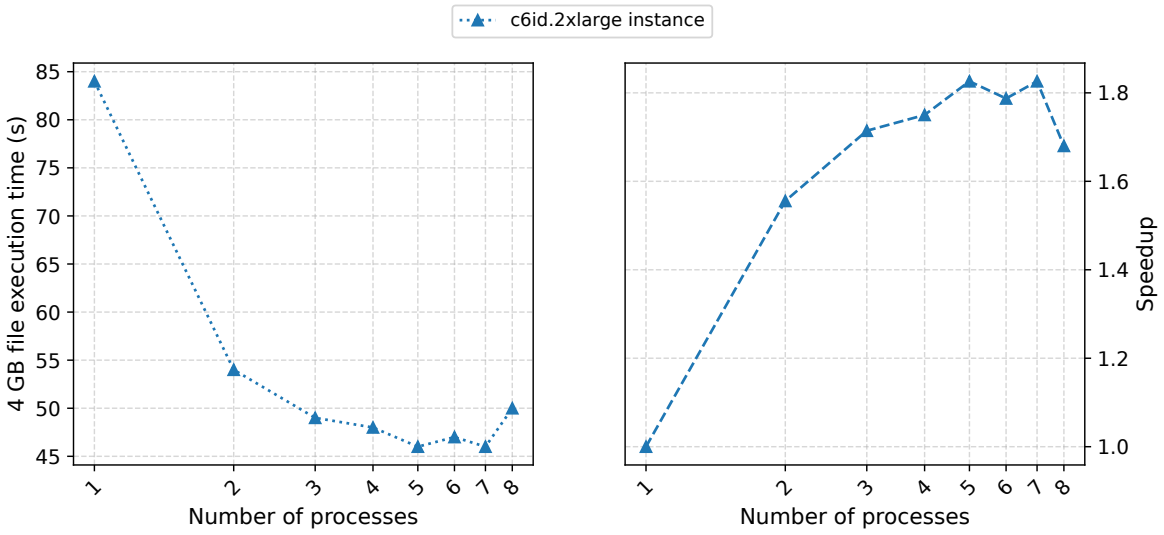

Figure 11: 4 GB file - Virtual machine execution time and speedup

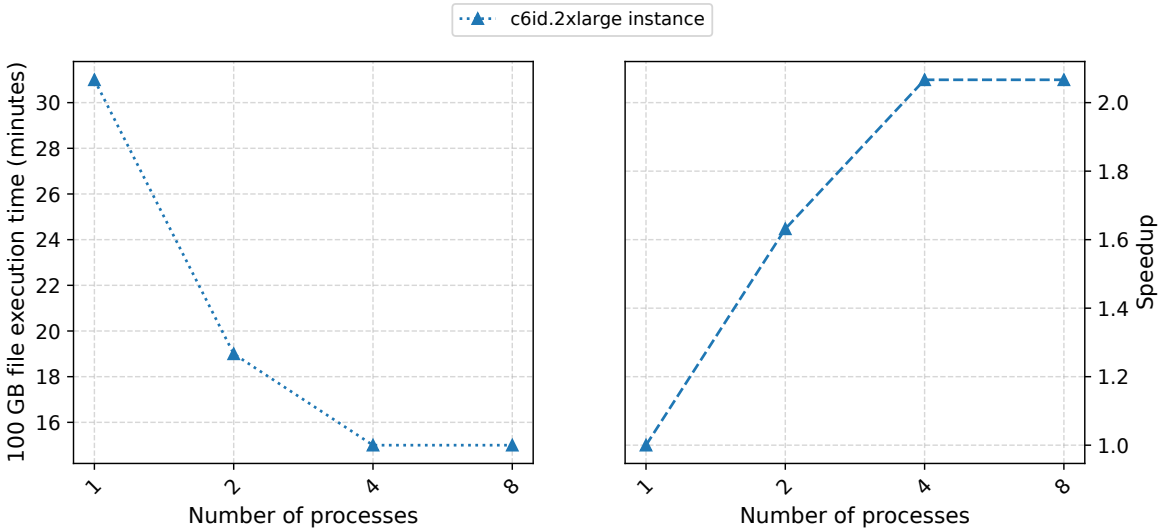

Figure 12: 100 GB file - Virtual machine execution time and speedup

Additional throughput (reading from S3 + processing + writing to S3) Figures 14 and 15 exhibit the reward in having a serverless solution that can easily scale. The higher the number of functions, the smaller the chunk size is.

$$Average\ function\ execution\ time = \frac{\sum Execution\ time\ of\ each\ function}{Number\ of\ functions\ running\ in\ parallel} \quad (1)$$

Execution time of each function is the time it takes a function to finish (lithops lambda setup + business logic execution time). Fetching status function time (orchestrating) is not taken into account.

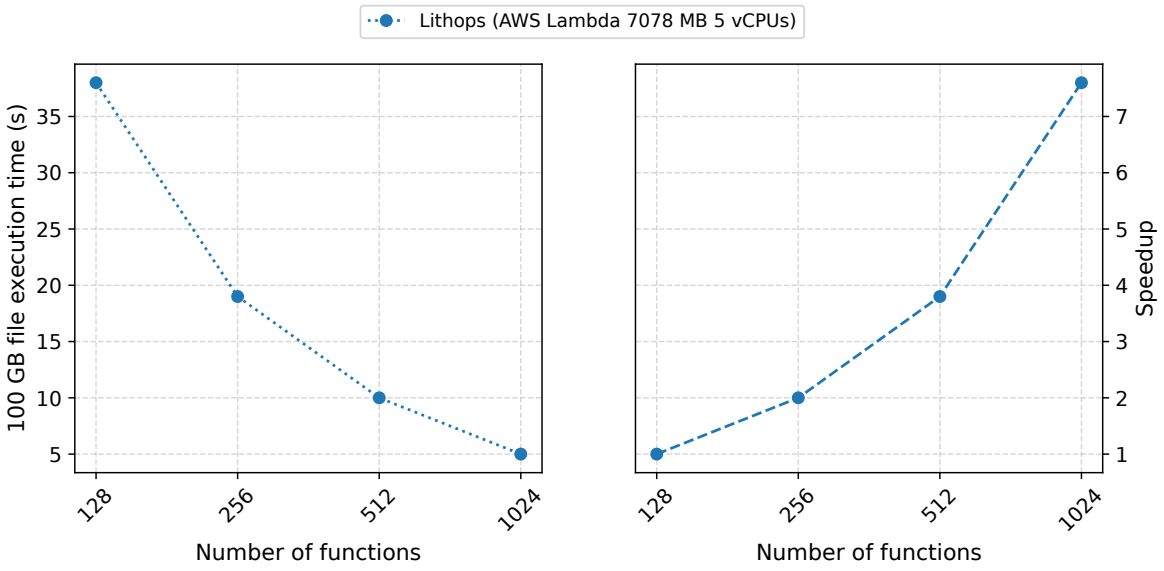

Figure 13: 100 GB file - Lambda execution time and speedup

Calculating the average execution time makes the assumption that all functions are running in parallel, all at the same time and not taking into account cold starts or delayed executions.

Thus, throughput can be defined as:

$$Throughput = \frac{Input\ file\ size}{Average\ function\ execution\ time} \tag{2}$$

It is important to consider that the number of functions and virtual machine number of processes are not directly comparable because a single function may have more than a single process itself. For instance, whereas a single virtual machine has been provisioned with 8 vCPUs, the flexibility of serverless allows to launch 1024 functions which use 5120 vCPUs and up to 7.2 TB of memory, billed only for the time used. Furthermore, tested Lambda configurations are still not the fastest and the concurrency quota can still be increased to reduce wall-clock time even more (up to 3000 concurrent functions) as long as the chunk size is at least 1 MB.

The total billed amount for the *Lambda* experiments (all configurations and plots) was approximately $5. Temporarily storing results in S3 costed less than 40 cents.

4.2 Validation key takeaways

There is an important problem of having a sequential reducer: running it inside Lambda functions for huge files is unsafe because it may trigger the 15 minute limit. If wall-clock time does not have to be minimized, the reducer can just run using AWS Batch.

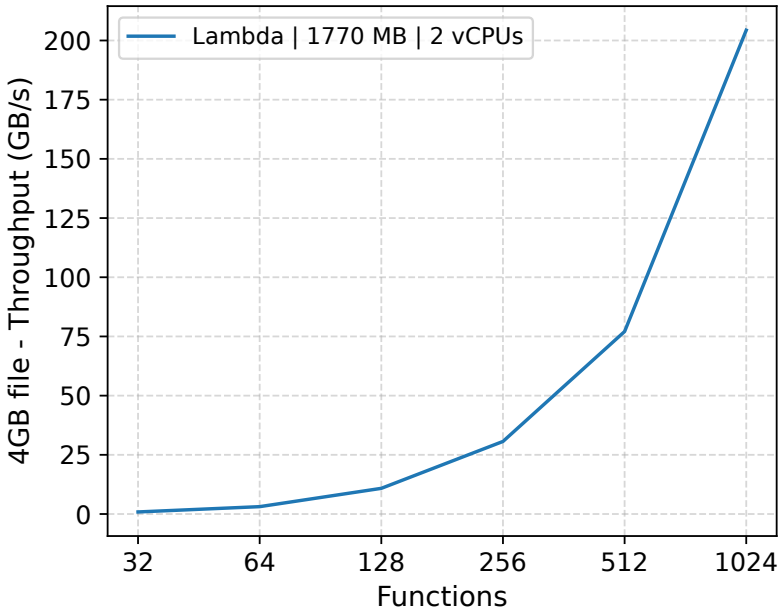

Figure 14: 4 GB file - Throughput

On the other hand, it can be seen that the serverless map phase beautifully scales. It is run in parallel benefiting from aggregate resources, reducing the uncertainty on billing and reducing execution time considerably without compromising over or underprovisioning. The biggest gains for performance come as the input file size increases, the parallelism increases the throughput to a certain point that would not be possible inside a single-machine environment, and thus qualifying our solution for big data analytics.

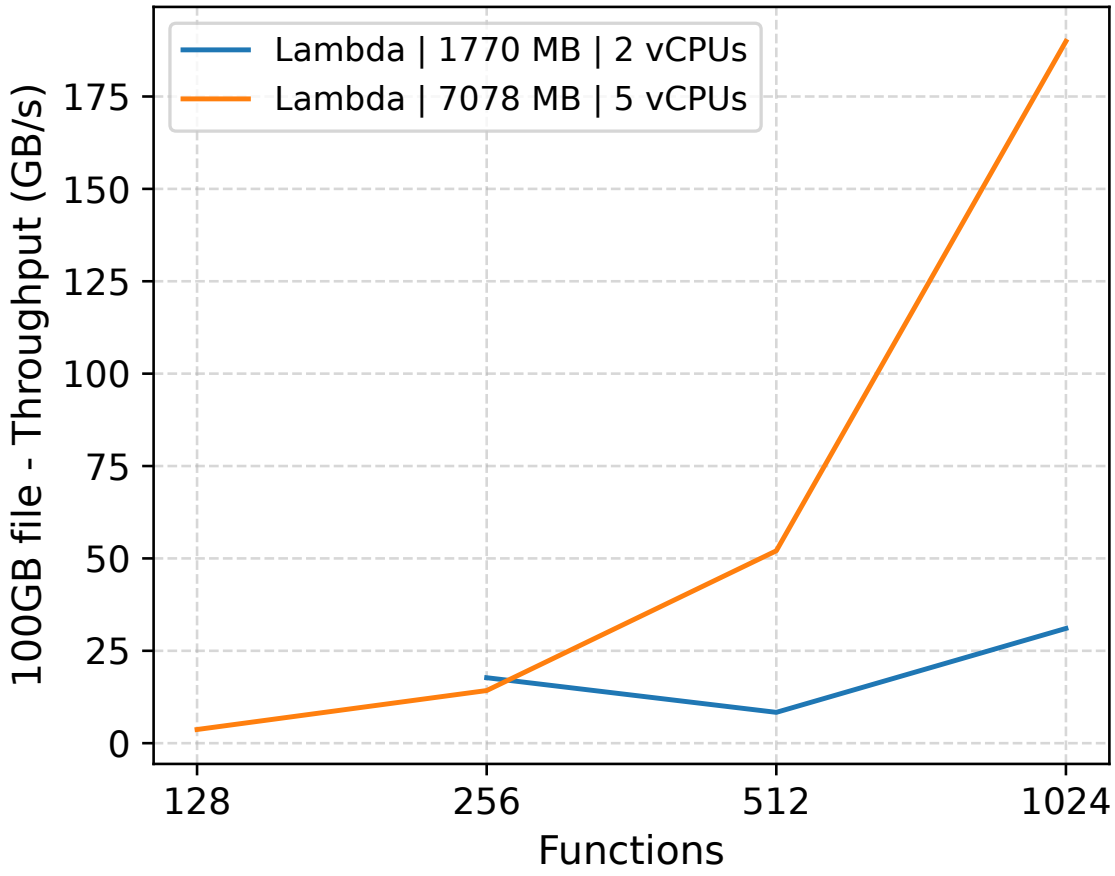

Figure 15: 100 GB file - Throughput

5 Conclusion

Our work has led us to conclude that an inter-process communication wrapper is the simplest approach towards the cloudification of an existing multiprocessing parallel framework already used in different use-cases and on polyglot programming constraints. We have devised a satisfactory methodology to avoid the overhead of rewriting business logic, treating each program like a black box instead of tweaking its internals in a lengthy process. We also showcased the essential modifications the original parallel framework has undergone to unleash the power cloud has to offer. Finally, as the experiments prove, the wrapper achieves a serverless architecture that relies on fundamental concepts of both cloud and systems programming. After the changes, the new architecture outperforms single-machine executions for huge files and the local-cloud performance gap greatly improves as bigger the input files grow. Thus, we can now declare this framework is big-data friendly.

This project has demanded me learning cloud computing and big data analytics concepts, as well as getting immersed into multidisciplinary subjects such as bioinformatics. It has nudged me to go beyond jargon, to speak and understand different disciplinary languages when working with scientists of different knowledge backgrounds. Genomics intricacies such as DNA sequencing, commonly used formats (FASTA/FASTQ, SAM/BAM, MAP), variant calling and the best practices in the field were among things I had to assimilate during the first phases of the project.

Personally, knowing that the results of this work will be used in a production environment with real impact has justified my hard work over the year. I am grateful to the CloudLab research team and the collaborators from The James Hutton Institute, Paolo Ribeca and Lucio Marcello for helping to make possible this thesis.

References

[1] Antonio Salis. Serverless computing and faas, the compss approach. https://www.mf2c-project.eu/index.html@p=4922.html, 2019.

[2] Shawn Khan AWS. Comparing your on-premises storage patterns with aws storage services. https://aws.amazon.com/blogs/storage/comparing-your-on-premises-storage-patterns-with-aws-storage-services/, 2020.

[3] Vaishaal Shankar. numpywren readme. https://github.com/Vaishaal/numpywren/blob/d635456bb497238c3ae8c428188fee52e8d52851/README.md#numpywren-1, 2017.

[4] European Nucleotide Archive. Statistics. https://www.ebi.ac.uk/ena/browser/about/statistics, 2022.

[5] Amazon Web Services. Aws lambda - developer guide - lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, 2022.

[6] Luc van Donkersgoed. Aws re:invent 2020 day 3: Optimizing lambda cost with multi-threading. https://www.sentiatechblog.com/aws-re-invent-2020-day-3-optimizing-lambda-cost-with-multi-threading, 2022.

[7] Cloudbutton Project Members. Cloudbutton project. https://cloudbutton.eu/, 2022.

[8] Aitor Arjona, Gerard Finol, and Pedro Garcia-Lopez. Transparent serverless execution of python multiprocessing applications, 2022.

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

[10] Antonio Regalado. Who coined 'cloud computing'? https://www.technologyreview.com/2011/10/31/257406/who-coined-cloud-computing/, 2011.

[11] Amazon Web Services. Announcing amazon elastic compute cloud (amazon ec2) - beta. https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/, 2006.

[12] Peter Mell and Timothy Grance. The nist definition of cloud computing, 2011-09-28 2011.

[13] Cloudflare. What is multitenancy? https://www.cloudflare.com/learning/cloud/what-is-multitenancy/, 2022.

[14] CloudZero. Single-tenant vs. multi-tenant cloud: Which should you use? https://www.cloudzero.com/blog/single-tenant-vs-multi-tenant, 2021.

[15] Paul C. Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The server is dead, long live the server: Rise of serverless computing, overview of current state and future trends in research and industry. *CoRR*, abs/1906.02888, 2019.

[16] Glenn O'Donnell Mike Gualtieri. Augment devops with noops. https://www.forrester.com/report/augment-devops-with-noops/RES59203, 2011.

[17] Red Hat. What is function-as-a-service (faas)? https://www.redhat.com/en/topics/cloud-native-apps/what-is-faas, 2020.

[18] IBM Cloud Education. What is cloud storage. https://www.ibm.com/cloud/learn/cloud-storage, 2019.

[19] Josep Sampe, Marc Sanchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro Garcia-Lopez. Outsourcing data processing jobs with lithops. *IEEE Transactions on Cloud Computing*, pages 1–1, 2021.

[20] genome.gov NIH. Genomic data science fact sheet. https://www.genome.gov/about-genomics/fact-sheets/Genomic-Data-Science, 2022.

[21] NCBI. Sequence read archive database growth.

[22] Monya Baker. Next-generation sequencing: adjusting to data overload. *Nature methods*, 7(7):495–499, 2010.

[23] Ben Langmead and Abhinav Nellore. Cloud computing for genomic data analysis and collaboration. *Nature Reviews Genetics*, 19(4):208–219, 2018.

[24] Wikipedia contributors. Black box — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Black_box&oldid=1084451807, 2022. [Online; accessed 28-May-2022].

[25] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. The serverless shell. In *Proceedings of the 22nd International Middleware Conference: Industrial Track*, pages 9–15, 2021.

[26] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319, 2017.

[27] Wikipedia contributors. Dataflow programming — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Dataflow_programming&oldid=1077268741, 2022. [Online; accessed 28-May-2022].

[28] Tanveer Ahmad, Nauman Ahmed, Johan Peltenburg, and Zaid Al-Ars. Arrowsam: In-memory genomics data processing using apache arrow. In *2020 3rd International Conference on Computer Applications & Information Security (ICCAIS)*, pages 1–6. IEEE, 2020.

[29] Peter JA Cock, Christopher J Fields, Naohisa Goto, Michael L Heuer, and Peter M Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic acids research*, 38(6):1767–1771, 2010.

[30] Santiago Marco-Sola, Michael Sammeth, Roderic Guigó, and Paolo Ribeca. The gem mapper: fast, accurate and versatile alignment by filtration. *Nature methods*, 9(12):1185–1188, 2012.

[31] Amazon Web Services. Aws lambda - operator guide - memory and computing power. `https://docs.aws.amazon.com/lambda/latest/operatorguide/profile-functions.html`, 2022.

[32] Herb Sutter et al. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.