Xavier Roca I Canals

PORTING GENOMICS PIPELINES TO THE CLOUD

SERVERLESS COMPUTING AS AN AVENUE FOR SCALABLE VARIANT CALLING

FINAL DEGREE PROJECT

Directed by Pedro Antonio García López

Co-directed by German Telmo Eizaguirre Suarez

Computer Engineering

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2022

Resum.

En l'actualitat, l'anàlisi de grans volums de dades a nivell local s'ha tornat poc pràctic a causa dels límits de computació i emmagatzematge d'ordinadors personals i clústers propietaris. Com a resposta, el concepte de cloud computing i el paradigma *serverless* prenen importància, gràcies a l'escalabilitat, elasticitat i el mode de facturació de pagament per ús de recursos que ofereixen. Les arquitectures *serverless* suprimeixen la gestió de la infraestructura per part de l'usuari, millorant l'accessibilitat a arquitectures distribuïdes. Per facilitar l'anàlisi i el desenvolupament d'aplicacions cloud, emergeixen nous *frameworks* com Lithops, basats principalment en el model *serverless FaaS* (*Function as a Service*). Adapten el model MapReduce al *FaaS*, executant mappers i reducers en paral·lel sobre les denominades *cloud functions*. Així i tot, les solucions que ens ofereix Lithops no sempre compleixen amb els requisits del Big Data. En aquest treball presentem la migració al cloud d'un pipeline local per a l'anàlisi de dades de genòmica, més en concret, un procés de variant calling, a partir de la utilització de Lithops. Mostrem el desenvolupament i la implementació d'una arquitectura distribuïda basada en el model MapReduce per a la paral·lelització de la fase final del pipeline. Avaluem la solució segons la seva escalabilitat i elasticitat. Comparem opcions amb diferents proveïdors de cloud per apropar-nos al rendiment més gran amb el mínim cost. Finalment, aportem dues solucions combinant sistemes d'emmagatzematge en el núvol i funcions cloud: una opció genèrica i una de major rendiment integrant serveis específics d'Amazon Web Services (S3 Select).

Resumen.

En la actualidad, el análisis de grandes volúmenes de datos a nivel local se ha vuelto poco práctico debido a los límites de computación y almacenamiento de ordenadores personales y clústeres propietarios. Como respuesta, el concepto de cloud computing y el paradigma *serverless* toman importancia, gracias a la escalabilidad, elasticidad y el modo de facturación de pago por uso de recursos que ofrecen. Las arquitecturas *serverless* suprimen la gestión de la infraestructura por parte del usuario, mejorando la accesibilidad a arquitecturas distribuidas. Para facilitar el análisis y el desarrollo de aplicaciones cloud, emergen nuevos *frameworks* como Lithops, basados principalmente en el modelo *serverless FaaS* (*Function as a Service*). Adaptan el modelo MapReduce al *FaaS*, ejecutando mappers y reducers en paralelo sobre las denominadas *cloud functions*. Aun así, las soluciones que nos ofrece Lithops no siempre cumplen con los requisitos del Big Data. En este trabajo presentamos la migración al cloud de un pipeline local para el análisis de datos de genómica, concretamente, un proceso de variant calling, a partir de la utilización de Lithops. Mostramos el desarrollo e implementación de una arquitectura distribuida basada en el modelo MapReduce para la paralelización de la fase final del pipeline. Evaluamos la solución en base a su escalabilidad y elasticidad. Comparamos opciones con diferentes proveedores de cloud para acercarnos al mayor rendimiento con mínimo coste. Finalmente, aportamos dos soluciones combinando sistemas de almacenamiento en la nube y funciones

cloud: una opción genérica y una de mayor rendimiento integrando servicios específicos de Amazon Web Services (S3 Select).

Abstract.

Nowadays, analyzing large volumes of data locally has become impractical due to the computational and storage limits of personal computers and proprietary clusters. In response, the concept of cloud computing and the *serverless* paradigm are gaining importance, thanks to the scalability, elasticity and pay-for-use billing model they offer. *Serverless* architectures eliminate user management of the infrastructure, improving the accessibility of distributed architectures. To facilitate the analysis and development of cloud applications, new *frameworks* such as Lithops are emerging, based mainly on the *serverless FaaS* (*Function as a Service*) model. The MapReduce model is adapted to *FaaS* by running mappers and reducers in parallel on the so-called *cloud functions*. Even so, the solutions offered by Lithops do not always meet the requirements of Big Data. In this project we present the migration to the cloud of a local pipeline for genomics data analysis, more specifically a variant calling process, based on the use of Lithops. We present the development and implementation of a distributed architecture based on the MapReduce model for the parallelization of the final phase of the pipeline. We evaluate the solution based on its scalability and elasticity. We compared options with different cloud providers to approach the highest performance with minimum cost. Finally, we provided two solutions combining cloud storage systems and cloud functions: a generic option and a higher performance option integrating specific services from Amazon Web Services (S3 Select).

Index

Table index

Equation index

Figure index

Code index

1 Introduction and Problem Statement

The rise of different technologies and processes for genomics data analysis comes in hand with advances in performance and resolution of data extraction methods. This has improved the throughput for obtaining genetic information but at the same time has complicated the applications and architectures capable of processing these large volumes of data.

In this project we will explain how the cloud offers multiple services capable of hosting genomics processes, with emphasis on a variant caller pipeline brought to a cloud environment.

Initially, this pipeline was executed in a local environment. The size of data varies across the pipeline, significantly surpassing its input volume at its middle steps. As a consequence, execution time increased superlinearly as the data increased and unsustainably for large input scales. The lack of computational power of personal computers or proprietary clusters limits the ability to analyze large volumes of data. Therefore, a solution in a local environment offers very limited scalability.

The advantages offered by the cloud places it as an ideal option for applications with intensive and highly variable computational and storage consumption.

1.1 Project Motivation

This final degree thesis was developed under a research grant at the CloudLab research group (Universitat Rovira i Virgili). The whole work is part of the CloudButton european project[1]. CloudButton has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825184. CloudButton's main initiative is to take advantage of *serverless* technologies to simplify the life cycle and programmability of big data analytics.

The present project came from a collaboration between CloudLab and the James Hutton Institute from United Kingdom, as a bioinformatics project for the analysis of large volumes of genomics data. This project is based on a variant calling pipeline running locally that had to be ported to a cloud environment. Variant calling is the process by which we identify variants from genomics sequence data. We align sequence to a reference genome to identify differences between both[2].

1.2 Project Goals

In this thesis we will focus on how the last stage of the pipeline that was exported to the cloud. Specifically, we will see a sequential data reduction stage adapted to a parallel context in the cloud. The `Lithops`' *serverless* framework and cloud functions were the foundations of this work.

Essentially, the main objectives of this project were defined as follows:

- Porting the current variant caller pipeline from a local environment to a cloud environment.
- Being able to process large volumes of genomics data with the migrated pipeline.
- Decreasing the pipeline execution time using services offered by cloud providers.
- Performing this task on the basis of the best performance vs cost ratio.

This pipeline consisted of different phases which were assigned to different CloudLab colleagues. Our task in this project was based on the reduce phase. Therefore, the personal objectives we were assigned were as follows:

- Parallelizing the sequential reduction phase.
- Creating a cloud architecture from the MapReduce model capable of supporting the objectives above.
- Optimizing the use of cloud resources to obtain the best performance at minimum cost.

From these objectives, we should be able to demonstrate the following benefits of using the cloud for these problems:

- The ability to easily port compute- and storage-demanding local applications to the cloud.
- The ability to process large volumes of data in a short time, offering a significant reduction of the execution time compared to local applications.
- The capacity to automatically and easily manage resources through cloud services.
- An optimization of the resources used, coming from the elasticity and scalability offered by cloud computing.
- A fine-grained cost management, thanks to the pay-per-use billing model.

Finally, we should be able to demonstrate that using cloud services is and will be an avenue to run large pipelines for genomics data analysis.

The modifications of the original genomics pipeline, and the design and implementation of two distributed reducers developed in different cloud providers (IBM Cloud[3] and Amazon Web Services[4]) using `Lithops` will be explained.

2 Background and Technological Context

2.1 Cloud Computing for Data Analysis

In recent years, data to be analyzed has increased exponentially in such a way that performing analytics locally has become impossible due to lack of computing and storage. In addition, the size of this data can be very variable, so having a fixed cluster size is not the best option in most cases anymore. Therefore, arises the need to develop a more elastic solution that adapts and scales according to the volume of data.

The concept of cloud computing emerges as a solution of this problem. Basically, cloud computing, also known as cloud, is the use of distributed computing resources such as servers or databases over the internet. The benefits offered by the cloud are multiple, including simplification of operations and services, scalability and pay-per-use billing model.

Using cloud, users should not worry about for provisioning services whose impact are unaware of, wasting resources or not having the opportunity to offer more services in an increase of clients and needs. Thanks to the pay-for-use model, the cloud offers great resource elasticity[5]. Concerning large volumes of data or Big Data, cloud computing allows to the customer to scale workloads according to the input size.

2.2 *Serverless* Paradigm

Despite the great benefits that cloud computing offers, users still have to deal with the management of these resources, which frequently forces them to configure and choose among embarrassing parameters to achieve the architecture that best suits their needs. This results in users not being able to exploit and take full advantage of the services offered. This is the reason why the concept of *serverless* computing arose.

Serverless is a cloud computing paradigm in which the cloud provider is the responsible of the infrastructure management task such as scalability or scheduling, allowing to the developers to focus only on the code. In this way, the pay-per-use billing model is exploited.

As a consequence, new *serverless* services appear apart from the resources offered by the classic *IaaS*[1] model of virtual machine instances. One of these services is known as *FaaS*[2]. *FaaS* is a type of cloud computing service that allows the user to execute brief code, wrapped in ”cloud functions”, without the need of an explicit infrastructure, and normally called through events[6]. These functions have the main characteristic of being stateless, that is, each function is independent of any other and their storage system is ephemeral, so there is no direct communication between functions. Communication and scheduling of cloud functions must be performed through external, decoupled storage and orchestration services.

Using cloud functions, the user is only responsible for specifying the allocated memory to be used and the maximum execution time of the functions, and consequently he will be charged according to these parameters.

[1] Infrastructure as a Service
[2] Functions as a Service

In conclusion, there are different aspects that differentiate *serverless* computing from serverful computing. First one, the *stateless* nature requires external storage to store both input and output data from the cloud functions. This feature allows decoupling computational capacity from storage capacity. Thus, it offers elasticity and scalability by allocating resources from different services. The second one is based on the fact that the user is no longer in charge of managing the resources, which is done automatically by the cloud providers. Finally, the billing is done once the execution of the application is finished based on the resources used –mainly allocated memory and execution time, although the billing criteria varies between *serverless* services and cloud providers.

2.3 Lithops

Under this new concept of *serverless* computing, new frameworks have emerged for the use of these resources and services in multiple applications.

`Lithops`[7] is a python multi-cloud *serverless* computing framework that allows the user to run his local python code in *serverless* computing platforms without requiring further cloud knowledge. In short, `lithops` allows the user to easily execute parallel python code in the cloud. Using the *FaaS* model, in the standard version of `lithops`, a thread is executed in each cloud function. It is important to emphasize that `Lithops` is a software project developed under the CloudButton project and highly used in different scientific fields, such as bioinformatics (genomics, metabolomics) and in geospatial data (LiDAR, satellital).

The main characteristics of this framework are the following:

- Easy to use, ability to run multiple *serverless* functions in parallel without any management efforts.
- Easy to learn, availability of documentation.
- Cross-platform, ability to run the same code across different *serverless* platforms without the need to modify the code.
- Error friendly, ability to detect errors and to evaluate the execution locally.
- Dynamic scaling, allows to exploit the maximum computational capacity offered by cloud providers based on on-demand usage.

`Lithops` has high-level APIs for computation and storage, based on cloud-based implementations of the classical local `futures` and `multiprocessing` libraries that Python offers for task parallelization.

Within the world of large-scale data analysis using cloud computing, the MapReduce model stands out as the most popular model used for this purpose. Lithops facilitates the execution of MapReduce functions through *serverless* functions.

2.4 MapReduce Model

MapReduce is a programming model for processing large inputs in parallel taking an advantage of distributed systems. Consists of two phases:

- In the map phase multiple map functions are called to process some key/value pairs and generate a new set of intermediate key/value pairs. Each mapper is assigned a partition of the total input data.
- In the reduce phase a reduce function is called to merge, aggregate or transform all the intermediate values sharing same key. Each reducer takes its corresponding key/values from each mapper's output.

The input for this model tends to be large scale data. In most cases, this big data is pre-processed and partitioned into multiple chunks/parts. Each chunk will be assigned to a mapper that will process the chunk and will return a (set of) key/object(s). M mappers can be invoked in parallel depending on the number of chunks or the user's needs. This happens due to the non-dependence between these parallel functions, as this first map stage is embarrassingly parallel.

There is an implicit phase between the map and reduce phase. This phase is known as Shuffle Phase and consists of the transfer of the mappers intermediate output to the reducers. This phase helps reducers to easily distinguish when a new reduce function should start, as reducers can be triggered as map outputs are ready.

Finally, in the reduce phase R reducers will be invoked and will run its reduce function in order to process the intermediate values/objects and return an output.

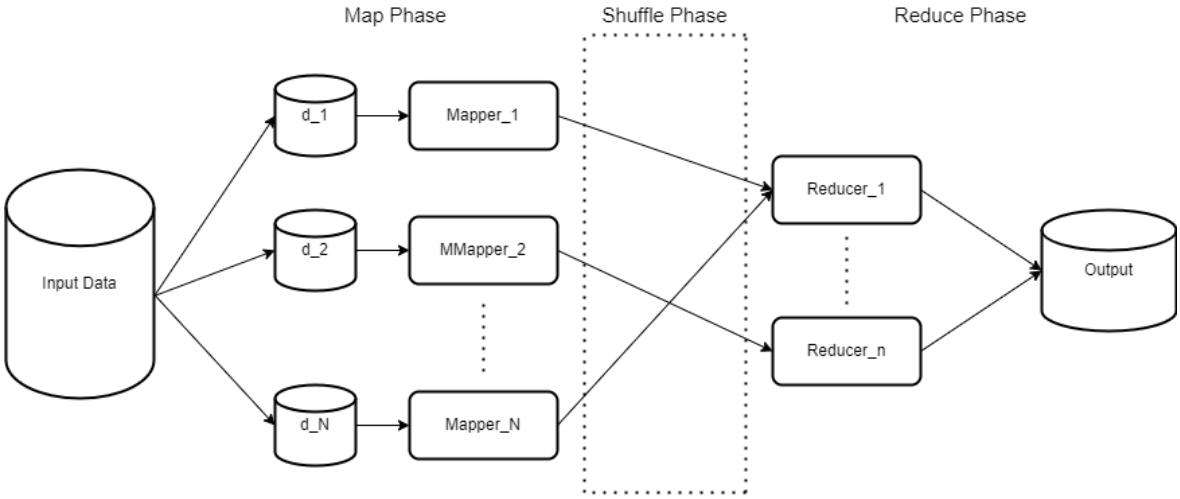

Figure 1: Map-Reduce architecture

An easy example to explain this model is the word count. Imagine we have large files with many words and we want to count all the words of each file. The MapReduce model can help us to solve this problem.

First of all, we could decide what our map and reduce functions will do:

- Map function: Count all words of the input file/chunk. Return a {word; appearances} dictionary.
- Reduce function: Sum the number of times a word appears.

The following is the architecture of a word count MapReduce:

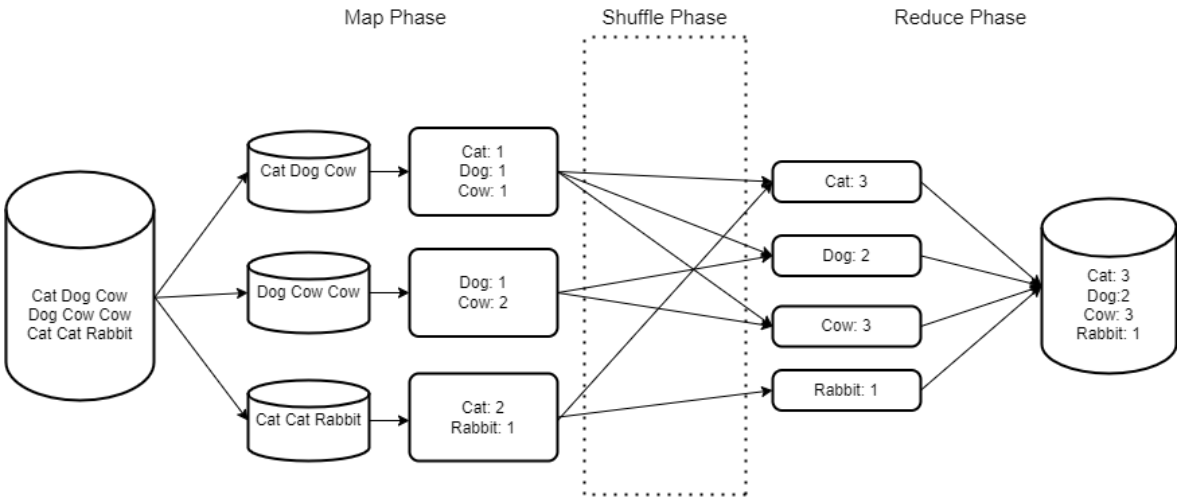

Figure 2: Word count MapReduce example

We can see that we have three lines with three words, in this case animals, as an input. Each line is assigned to a mapper that will count the number of times each word appears. In the shuffle phase, each word is assigned to a reducer. Basically, it will again aggregate the values according to its key. For example, in the case of Cat:[1,0,2], the reducers will sum all this values to get the total number (Cat: 3). Finally, all the outputs from the reducers can be merged into a single one or be written as separate partitions. Although this example is a willfully uncomplicated and non-demanding use-case, mainly for explanatory reasons, this architecture can be exploited in far more ambitious (in complexity and scale) problems.

The MapReduce model can scale up to process highly variable data volume, distributing the computation among the available computational resources. It can help the users to process and analyze large amounts of input across distributed systems. Thanks also to cloud computing, we can make the MapReduce model scalable by taking advantage of the decoupled computing and data storage it offers. In this way the workflow scalability and elasticty is guaranteed.

2.5 Lithops MapReduce Problem

As recently commented, Lithops integrates a vanilla MapReduce function within its API. The problem with this function is that is not oriented to process large volumes of data, as it calls M mappers but a single reducer. This implementation does not handle the scaling to bigger volume of data, as one single reducer is responsible of processing all the intermediate data.

Therefore, this implementation of MapReduce offers a bottleneck in one of the main phases of the MapReduce model. This results in a very poor performance that makes the scaling of architectures difficult. However, we can use Lithop's serverless function invocation APIs to create custom adaptations of the MapReduce model.

Further elaborating on the objectives listed in the previous section, in this project we will see the design and implementation of an ad-hoc *serverless* architecture through lithops to develop a scalable variant calling.

3 Related Work

Advances in genomic sequencing have reduced the cost and time to obtain genetic information, and in front of the increasing volume of data, the advent of *serverless* computing provides users with the computational resources needed for its analysis. Thus, cloud migration of these applications has become popular in recent years.

Researchers from the biology department of the University of Washington already migrated a variant calling workflow to *FaaS* cloud service. They reported that the provisioning and parallelization of theoretically unlimited resources could offer a certain percentage of error in practice, when the data size is of a very large scale. This is due to the limits offered by some of the cloud providers' services. Even so, this data could still be processed through a refined implementation of its setup/pipeline. Even with this problem and the *stateless* nature of the cloud functions, they conclude that *serverless* paradigm is and will be a great avenue for the development of genomics workflows[8].

Likewise, a study conducted at the Silesian University of Technology computed several case studies of omics through *serverless* computing, and accounted the advantages that each *serverless* service from different cloud providers can offer. They revealed great benefits in processing time, cost-effectiveness, scale of parallelism and improved privacy[9].

Finally, there exists evidence on how genomics database data has increased over the years with the advent of cloud computing, and that the MapReduce model offers a way to develop solutions for any type of data sequence analysis, such as variant calling. Thanks to this model multiple samples can be processed at the same time allowing a large scale reduction of the execution time[10].

4 The *Serverless* Variant Caller Pipeline

The main goal of the *serverless* variant caller was to port a genomics pipeline to the cloud using the `lithops` framework. As an introduction, we will see an overview of the starting version of our variant calling pipeline.

4.1 Input Files

For this pipeline two types of genomics files would be compared:

4.1.1 Fastq Files

`Fastq` files[11] are text-based files for storing biological sequences and their corresponding quality scores.

`Fastq` files uses four lines per sequence:

- First line: formed by a '@' character and a sequence identifier.
- Second line: formed by raw sequence letters.
- Third line: formed by a '+'/'-' character (corresponding to the DNA strand the sequence was read from) and in some cases the same sequence identifier.
- Fourth line: formed by the quality values of the sequence.

4.1.2 Fasta Files

`Fasta` files[12] are text-based files for representing biological sequences. It is used to reference genomes.

`Fasta` files uses two types of lines:

- First line: formed by a '>' character and a sequence description.
- Next lines: formed by lines of sequence data.

4.2 Pipeline

In this part the different phases of the current pipeline and the tools and formats that are used to complete the execution will be explained.

4.2.1 Data Partitioning

In this phase `fasta` and `fastq` files were partitioned into different numbers of chunks. This partitioning was performed on the fly in the cloud from a library previously developed by a project colleague.

4.2.2 Map Phase

In this phase the first processing of the partitions created in the previous phase took place. This processing was based on several alignments on genome sequences and references (`fasta` and `fastq` files). SAM tools[3][13] and GEM Mapper[4][14] were used for these alignments in order to create the `mpileup` files (intermediate files).

4.2.3 Mpileup File

The `mpileup`[15] format is formed by Tab-separated lines where each line represents the pileup of reads at a single genomic position.

`Mpileup` files consist of the following columns:

- Chromosome name.
- Position on the chromosome (Index).
- Reference base at this position.
- The remaining columns show the pileup data.

4.2.4 Reduce Phase

The last stage of the pipeline, the reduce phase, is the one we are focusing on in this work. In the reduction, the `mpileup` files (intermediate files) generated in the map phase were processed to obtain a final result to be later analyzed. For this purpose, these intermediate files were merged together in a single, big scale `mpileup` file which served as input for the execution of the SiNPle[5][16] application to retrieve an output.

Broadly speaking, as the specific genomics algorithm is not the late motiv of this thesis, the SiNPle is a variant caller software based on Bayesian modelling. For a position in the genome[6], SiNPle receives a bunch of sequencing data[7], and compares it to the reference genome[8] to detect statistically significant variants. For that, it uses a Bayesian approach that considers, among others, the distribution of nucleic acids or the sequencing error rates.

[3]SAM tools: Set of utilities for manipulating alignments in SAM format (Sequence Alignment/Map).

[4]GEM Mapper: High-performance mapping tool for aligning sequenced reads against large reference genomes.

[5]SiNPle: Fast and sensitive variant calling for deep sequencing data.

[6]Genome: the set of genetic information in a certain organism.

[7]Sequencing: extracting the genetic sequence, made of nucleic acids, from samples of a specific organism

[8]Reference genome: the assembled sequence of nucleic acids for a certain organism, representative its ”idealized” genomic sequence in research terms.

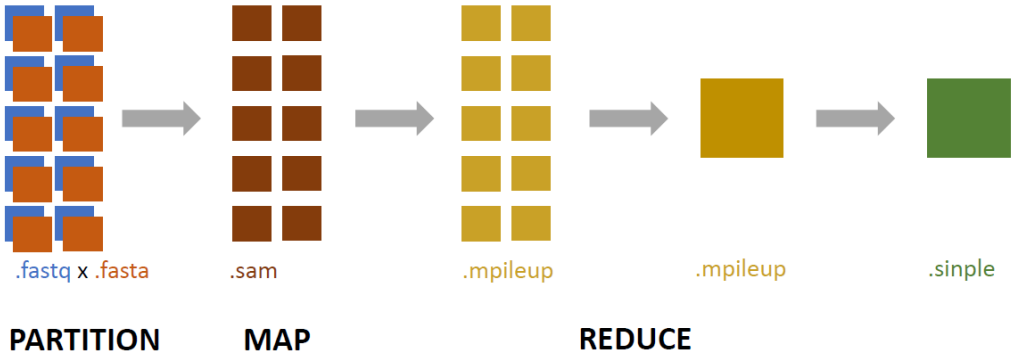

Figure 3: *Serverless* variant caller pipeline

5 Development Plan

In this section we will see the development plan of the project. The first meetings with Hutton Institute, the firsts steps, all the proposals and the phases of the project will be introduced.

Figure 4: Chronology of the project

5.1 First Meeting

In the first meeting with Hutton Institute, we had a first glance at the problem and the early working plan. We were explained about the whole initial pipeline they had, outlined in the previous section. Also an overview of the project and what goals wanted to achieve were given:

- Bringing all the pipeline to the cloud.
- Being able to process and analyze large amounts of genomics data.
- Reducing the execution time of the local pipeline through parallelisation using *serverless* & *stateless* cloud functions.
- Developing the *serverless* pipeline with the best performance vs cost ratio.

Once the objectives were set, we were instructed about all the components of the pipeline and the problems that were in each one. In my case, the reduce phase problem was assigned to me.

This task was to replace the MapReduce provided by the `lithops` framework to create a more ad hoc solution/architecture for the project. On the one hand, we had previous academic experience on cloud systems and the `lithops` framework, but we lacked familiarity with the genomics part of the problem.

In the following meetings we spent significant time on interpreting the input files, the information they have inside, the manipulation and transformation that was made to this data and finally the output file to be generated.

To initiate the first steps and fully understand the problem, a MapReduce example that counted words was made to familiarize with the map-reduce model and contextualize it in the *serverless* paradigm. In this way, we started the project with a prior knowledge of the architecture to be used and the genomics data to be analyzed.

5.2 First Proposal

From this first example, an initial solution to the problem could be proposed. This would be based on the use of the map function provided by `lithops`. In this case a new ad hoc architecture would be created based on the phases of the MapReduce model.

The phases to be considered were as follows:

1. Data partitioning: Divide the input files into different chunks. No modification was needed.
2. Map phase: Process input file and returns an intermediate result. M mappers had to be invoked through `lithops` map function.
3. Shuffle phase: Distribute the intermediate files across the reducers. It was necessary to create a new algorithm.
4. Reduce phase: Process intermediate files and returns a final result. R reducers had to be invoked through `lithops` map function.

Thanks to these modifications, functions could be invoked simultaneously for each of the map and reduce phases.

This first version consisted of a general and basic idea of how the solution should move forward. This was done in the IBM cloud in order to test this initial solution.

However, different test were carried out to check its performance. It was verified that it worked correctly with small input files and also different memory evaluations were made to check which one offered the nest performance in the reduce functions. Not only that, from the beginning we have been looking for the improvement between performance and cost comparison from different optimizations to reduce the execution time of the functions.

But we had a big problem, a computational limitation offered by the account level we had at IBM offered limited performance to run the *serverless* functions.

5.3 Migration to AWS

Once the first version was developed, the possibility of migrating the pipeline from IBM Cloud to AWS was raised in different meetings. This new cloud provider offered advantages in both cloud storage and *serverless* functions performance.

In order to run the first version of the pipeline in this new cloud, several changes were made to the map and reduce functions. These changes were based on the type of read and write offered by the AWS *serverless* functions.

While continuing to migrate the entire pipeline to this new cloud provider, several memory-based optimizations were performed to further reduce the demo execution time.

Even so, we still had a very basic functional version with small files. It was true that the reduce phase was parallelized, but it was not enough to process large data files, it did not allow scaling. Therefore, the bottleneck in the reduce phase appeared again.

5.4 Second Proposal

For the development of a new proposal to solve the problem that arose at the time of scaling, different meetings were held with a CloudLab colleague.

For this new version, each of the reducers processing a single chunk/set of the input files were parallelized. In this way these data could be divided into multiple reducers.

For this purpose, the formats of the intermediate files were modified and a new data extraction service offered by AWS was used.

Finally, different evaluations were carried out to analyze the performance and cost of this new enhanced version of the pipeline. These modifications allowed us to scale to a large volume of data, but at the same time we realized that in order to process the large volumes that exist in the genomics cases, further optimizations and probably the development of a new version would be necessary.

5.5 Arrangement and Coordination

The work methodology that we followed in this project was based on periodic meetings and increments, where each project colleague presented the work done and the problems that they solved and/or arose.

The contact between the managers and coordinators on the part of Hutton Institute and URV was direct through the Slack communication platform. In this platform doubts were solved and both documentation and project code were delivered.

Finally, the meetings were held weekly through the Microsoft Teams platform, which allowed to each project colleague to discuss the progress of the project.

6 Development - Vanilla Version

In this section we will review the first version of the distributed reduce phase. This was the implementation of the first proposal. We will focus on the design, the architecture, the implementation with a few snippets of code and some little experiments to prove that this solution worked for small amounts of data. Finally, the problems we encountered once the version was finished will be explained.

6.1 Design

This vanilla version was developed around the IBM cloud provider.

Basically, the following services were used:

- Cloud Object Storage[17]: Service to store objects/data in the cloud.
- IBM Cloud functions[18]: Service to run application code in *serverless* functions.

The entire MapReduce was developed in the IBM Cloud with only these two basic services.

At the meeting where this proposal was made, different goals were set:

- Removing the use of the `lithops` MapReduce function to create a new implementation from a fully parallel architecture. For now, we will focus on an ad-hoc implementation of the solution.
- Making the necessary modifications to the map and reduce function in order to communicate through object storage, due to the *serverless* functions *stateless* nature.
- Focusing on having a demo that works with little data to prove that the cloud is a good choice.
- Not thinking about scaling data, this step goes after having a demo working.

6.2 Architecture

Once we have explained the services and the goals, we will see the architecture of this ad-hoc MapReduce.

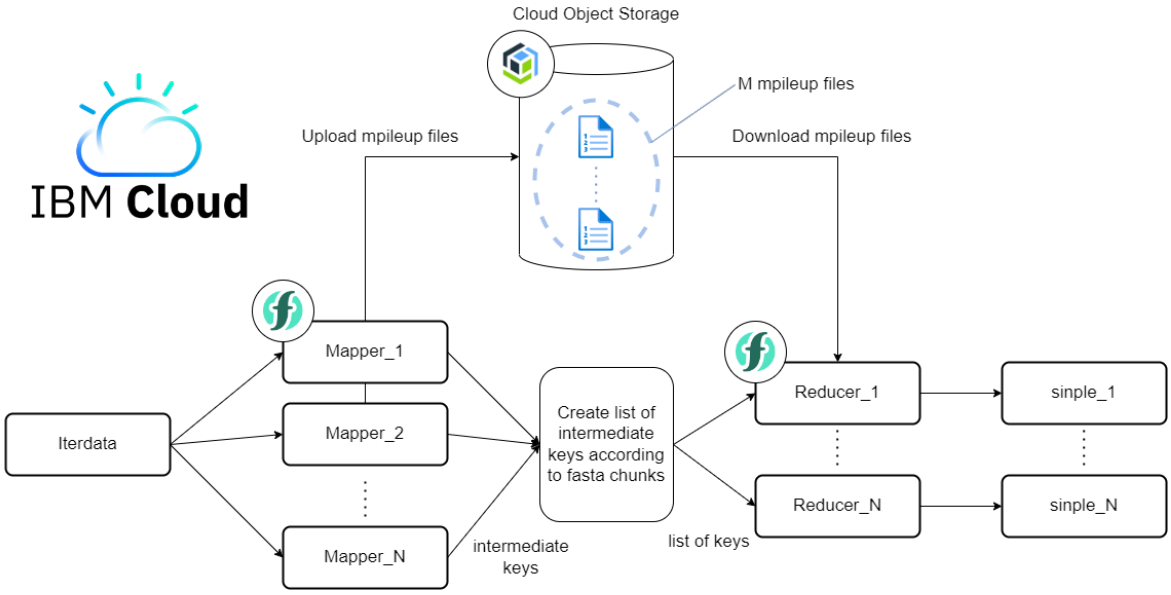

Figure 5: Vanilla version architecture

We can differentiate five main steps within this architecture:

1. Input data partition: In this step the iterdata (input data) is created for the mappers. Basically, it consists on executing functions to partition the `fasta` and `fastq` files. A dictionary will be created with the byte ranges of each `fasta` chunks and the related `fastq` chunks.

2. Map phase: $Xfastachunks * Yfastqfiles$ number of mappers will be invoked to process the input data in order to create the `mpileup` files. In this case, each `mpileup` file will upload its output to the Cloud Object Storage and will return the key with which the file was uploaded. A mapper will consist of a single IBM Cloud Function that will execute the map function code.

3. Shuffle phase: A list of lists with the intermediate keys returned by the mappers will be created. Each list will consist of all the `mpileup` files keys related to a `fasta` chunk. In total, L lists of lists of `mpileup` files keys will be made, where L is the number of `fasta` chunks.

4. Reduce phase: Each list of `mpileup` keys will be the input of one reducer. Therefore, R reducers will be invoked where R is the number of `fasta` chunks. Each reducer will process all the `mpileup` files from a single `fasta` chunk. Each reducer will consist of a single IBM Cloud Function that will execute the reduce function code. Inside the function, the `mpileup` files will be downloaded according to their keys and will be merged into a single `mpileup` file to process it and convert it to a `sinple` file.

5. Output: In this case, the outputs from the reducers will not be merged. So, If R reducers are invoked we will have S `sinple` files.

After this overview of the architecture and its steps we can focus on the implementation of this first version of the pipeline.

6.3 Implementation

As explained at the beginning of this project, the library for partitioning input files (`fasta` and `fastq` files) and creating the iterdata for mappers was already implemented by a project colleague. Not only that, the vanilla versions of the map and reduce functions were also implemented. Basically, our task was to implement a distributed architecture that would solve the bottleneck of the map reduction of `lithops`. From this clarification, the modifications of the architecture and their implementations will be explained.

6.3.1 Map Function Modification

In the original pipeline, the map function directly returned the `mpileup` object. So, for this new version this step had to be modified. The modification was very simple, it was a matter of uploading the `mpileup` file to the cloud object storage and return its key.

```
    storage.put_object(BUCKET_NAME, intermediate_key, body=mpileup_out)
    return intermediate_key
```

Code 1: Upload mpileup file to COS

These two lines were at the end of the function. `lithops` provides a storage API[19] for the user to make different actions on Cloud Object Storage. For this case, the `put_object()` function was used to upload the `mpileup` file to the storage.

The arguments for the function were the following:

- BUCKET_NAME: Name of the bucket where the object will be stored.
- intermediate_key: Name of the object.
- mpileup_out: Contents of the `mpileup` file.

Finally, the intermediate key was returned so that the reduce function could access and download the file (IBM COS object).

6.3.2 Reduce Function Modification

For the reduce function, the merging of the `mpileup` files was modified:

```
    temp_mpileup = '/tmp/file.mpileup'
    with open(temp_mpileup, 'w') as f:
        for key in results:
            mpileup = storage.get_object(BUCKET_NAME, key).decode('UTF-8')
            f.write(mpileup)
```

Code 2: Merging mpileup files

With this snippet of code the files could be downloaded using the `get_object()` function from the `lithops` storage API. The list of `mpileup` file keys was stored in the results variable. Each `mpileup` file was downloaded sequentially. Once downloaded, the contents of the file were written to the final `mpileup` file that (created in line 1 of the snippet). Afterwards the reduce functions continued with its normal reduction code – i.e. processing the `mpileup` data into `sinple`.

Finally, the `sinple` result was uploaded to the cloud in order to have a result to be subsequently analyzed.

```
1    storage.put_object(BUCKET_NAME, sinple_key, body=sinple_out)
```

Code 3: Upload sinple file to COS

The `put_object()` function was once again used to upload the results to the cloud object storage.

6.3.3 MapReduce Class

Once the map and reduce functions modifications were done and ready to be executed with the IBM Cloud Functions, a MapReduce class was created to replace `lithops`' native MapReduce function. Next we will give an overview of the implemented MapReduce wrapper.

To get a first idea, `lithops` allows to the user to call a map function through its futures API[20]. This futures API can be considered as the cloud version of the python concurrent.futures library. Basically, concurrent.futures provides a high-level interface to execute calls asynchronously. Therefore, with the help of this function several simultaneous calls in the cloud can be called.

In the original `lithops` interface, the map function can be called with the following lines of code:

```
1    fexec = lithops.FunctionExecutor()
2    fexec.map(my_map_function, iterdata)
3    fexec.get_result()
```

Code 4: Lithops map function invocation

The function executor will use the `lithops` configuration to determine its execution mode (localhost[9], *serverless*[10] or *standalone*[11]). With the executor we can call the map function being the arguments will the map function to be executed and the input data for the mappers. Finally, we can get the result using the `get_result()` function. For this project the execution mode was *serverless*.

With this basic introduction on how to invoke `lithops` map functions the MapReduce function can be introduced. This function consisted of the three steps that compose the MapReduce model.

[9] localhost: This mode allows the user to execute functions in a local machine using processes.

[10] *serverless*: This mode allows the user to execute functions using *serverless* compute services.

[11] *standalone*: This mode allows the user to execute functions using Virtual Machines in a private cluster or in the cloud.

The following code corresponds to the Map phase:

```
fexec = lithops.FunctionExecutor(runtime=self.runtime,
    runtime_memory=self.runtime_memory,
    log_level=self.log_level)
fexec.map(self.map_func, iterdata)
map_results = fexec.get_result()
```

Code 5: Map phase

As we can see, the function executor had different arguments:

- runtime: Name of the docker image to run the cloud function.
- runtime_memory: Physical memory to use to run the cloud function.
- log_level: Log level printing.

To try to clarify these arguments, a docker image is a template that contains a set of instructions for creating a container[12] that can be run on the Docker platform. The Docker platform allows to the user to separate application from the infrastructure. In this case, all packages, libraries, scripts and python version needed to run our map and reduce functions in the IBM Cloud Functions were included.

Once the executor was configured, the map function was called to invoke M mappers, the same number of items from the iterdata. This iterdata contained the `fasta` and `fastq` chunk keys and their range of indexes to get the information. Finally, a map_results variable was created to store the output from the mappers that contained all the `mpileup` file keys.

After the map phase, the next phase was the Shuffle phase:

```
keys = self.create_intermediate_keys(map_results)
intermediate_keys = self.check_size(storage,
    self.bucket,
    keys,
    self.runtime_memory)
```

Code 6: Shuffle phase

This phase consisted of two local functions:

- create_intermediate_keys: This function created the list of lists of `mpileup` files according to the `fasta` chunks.
- check_size: This function created a dictionary according to the size of each list of `mpileup` files. The aim of this function was to dynamically modify the memory of the function so that each function has the exact memory needed to be executed without errors. The reason for this function was because cloud providers charge you for functions by execution time and allocated memory. therefore, the goal was to minimize memory usage.

[12] container: software component capable of packaging the code and its dependencies so that the application can run in multiple environments

A small example of what this dictionary would look like would be the following:

```
1  {
2      "1024":[mpileup_key_1_fasta_chunk_1, ... ,
   mpileup_key_N_fasta_chunk_1],
3      "2048":[mpileup_key_1_fasta_chunk_2, ... ,
   mpileup_key_N_fasta_chunk_2]
4  }
```

Code 7: Input data for reducers

To find the ideal memory for each cloud function, a little memory experiment was made. The goal of this experiment was to find the best runtime memory to achieve the best performance and hence, the minimum execution time for the cloud functions.

The input for this experiment was different groups of `mpileup` files according to the following sizes: 300MB, 600MB, 900MB, 1200MB and 1500MB. In order to get the best runtime memory for each function the memory was increased, starting from the size of the `mpileup` files up to doubling it. For example, if the 300MB `mpileup` files were used we would test from 300MB to 600MB runtime memory. To conclude with the best runtime memory the execution time was measured for each case.

The results obtained were the following:

Total mpileup size	**100%**	**120%**	**140%**	**160%**	**180%**	**200%**
300MB	119s	102s	97s	101s	103s	99.4s
600MB	230s	205s	205s	210s	220s	218s
900MB	344s	314s	288s	292s	287s	293s
1200MB	500s	461s	444s	438s	424s	436s
1500MB	520s	470s	470s	491s	502s	512s

Table 1: Runtime memory experiment results. Column names reflect the runtime memory of the cloud functions, as percentages over the processed `mpileup` data size

In conclusion, each time the runtime memory was increased, the performance of the funcion increased. This was because the cloud functions use some of the runtime memory for the computation. Then, if the maximum size of the `mpileup` files were used as runtime memory we could limit this computation of the cloud function. It should also be emphasized that the cpu's participation goes hand in hand with the increase in memory[21]. Therefore, the more memory used, more cpu utilization.

It can be seen in the table that the improvement from 140% to 200% did not offer a significant improvement. Therefore, the conclusion was that using a runtime memory of 140% respect to the total size of the `mpileup` files was enough to get the best or an optimal performance.

Therefore, the `check_size()` function was modified to calculate the runtime memory based on the results obtained. In this case, the runtime memory was 140% of the size of the `mpileup` files.

The following snippet of code corresponds to the reduce phase:

```
    if(len(intermediate_keys) > 0):
        storage = Storage()
        for key, value in intermediate_keys.items():
            fexec = lithops.FunctionExecutor(runtime=self.runtime,
                        runtime_memory=key,
                        log_level=self.log_level)
            fexec.map(self.reduce_func, value)
            all_results.append(fexec)

        for fexec in all_results:
            fexec.get_result()
```

Code 8: Reduce phase(1)

The code checked that the dictionary of lists of `mpileup` files keys was not empty. Then, it went through all the items in this dictionary to take the key and the value. The key corresponded to the `runtime_memory` value and the value was the list of keys. With the `runtime_memory` value the configuration of the executor could be changed dynamically to invoke cloud functions with different memory. Then, the list of keys was used as input for the reduce function. Finally, the results were obtained with the `get_results()` function.

Once all the components were implemented, the MapReduce function could be called in this way:

```
    mapreduce = MapReduce(map_alignment,
                          reduce_positions,
                          runtime_id,
                          runtime_mem,
                          'DEBUG',
                          BUCKET_NAME)
    map_time, creating_keys_time, reduce_time = mapreduce(iterdata)
```

Code 9: MapReduce function invocation

The MapReduce class received the following arguments:

- map_alignment: Name of the map function.
- reduce_positions: Name of the reduce function.
- runtime_id: Name of the runtime.
- runtime_mem: Memory to use to run the cloud function.
- 'DEBUG': Log level.
- BUCKET_NAME: Name of the bucket where the objects will be stored.
- iterdata: Input data for the mappers.

We have seen the full implementation of the ad-hoc MapReduce vanilla version. In short, the map and reduce functions were modified to upload and download the files needed for the execution, because cloud functions do not have direct communication with each other. A MapReduce class was also created to wrap our MapReduce function implementation.

6.4 Evaluation

In this section we will take a look at all the experiments and optimizations that were performed on this vanilla version in order to achieve best performance and seek possible bugs or bottlenecks.

The first test was very simple. Basically, it consisted on checking that the whole pipeline was working with the new MapReduce function. This experiment was based on running the pipeline with the first five `fasta` chunks and checking that the `sinple` outputs were correct.

These aspects were taken into account for this experiment:

- Run the pipeline with [1,2,5] `fasta` chunks.
- For each number of `fasta` chunks ten runs were be performed.
- The size of the `mpileup` files within each reducer depended on each `fasta` chunk.
- Checked that a reducer is invoked for each `fasta` chunk.
- Checked that one output per `fasta` chunk is returned.
- Compared the execution time for each test.

Once the goals were set out we proceeded to perform this first test. The initial input was a 26MB `fasta` file divided into 99 chunks and a 685MB `fastq` file divided into 104 chunks. For this experiment only five chunks were used, so the first five chunks from the `fasta` chunks were taken. Each `fasta` chunk was processed with each `fastq`. Therefore, there were 104 `mpileup` files for each `fasta` chunk.

With the following table we can see more in detail the expected inputs and outputs files for the test.

	Map Phase		Reduce Phase	
# Fasta chunks	Input	Output	Input	Output
1	105 (fasta & fastq)	104 mpileup	104 mpileup	1 sinple
2	210 (fasta & fastq)	208 mpileup	208 mpileup	2 sinple
5	525 (fasta & fastq)	520 mpileup	520 mpileup	5 sinple

Table 2: Input and output files expected for the first evaluation test

Once the experiment is introduced, we will see the results obtained:

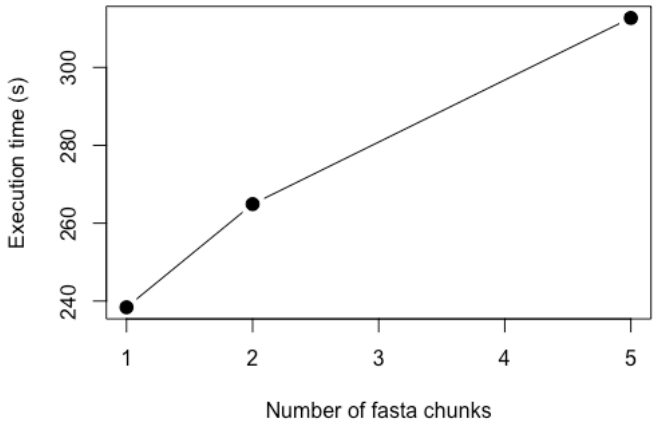

Figure 6: First experiment results (vanilla version)

Some of the results were satisfactory. The result files were the expected ones and the information inside them was correct. On the other hand, as the number of `fasta` chunks increased, so did the execution time, as can be seen in the plot. These execution time results were not exceptional. `Fasta` chunks were being processed in parallel so in theory the execution time had to be about the same. This could be caused by two issues, either the size of the `mpileup` files related to each `fasta` chunk was increasing or the computing of the IBM Cloud function was not working as expected. Despite its poor performance, the entire pipeline was working and getting the right results.

From here, the cloud functions could reach the timeout if the pipeline ran with all the `fasta` chunks. This would be a big problem, as some reducers might return a result and others would return an error. So, the reduce function had to be optimized to try to reduce execution time and prevent possible timeout errors.

There were few things to improve within the reduce function as it consisted of two simple steps, downloading and merging all the `mpileup` files into a single one and processing it with a script. We noticed that the way these files were downloaded could be modified. Basically, the files were downloaded sequentially, one after the other, and this download could be parallelized in order to reduce the execution time.

For this parallelization the `concurrent.futures` library was used by calling the `ThreadPoolExecutor` function that allowed functions to be executed asynchronously with threads.

The snippet of code for this optimization was the following:

```
1    with concurrent.futures.ThreadPoolExecutor(max_workers=20) as
     executor:
2        future = executor.map(getObject, results)
```

Code 10: Optimization of object downloading using concurrent.futures

This snippet called an executor with a maximum of twenty workers (threads), each of them running map functions. In this case, the `getObject()` function was called with the argument of the `result` variable. This variable contained the list of the `mpileup` files keys to be downloaded.

The implementation of the getObject() function was the following:

```
    def getObject(key):
        storage = Storage()
        with open('/tmp/file.mpileup', 'a') as f:
            mpileup = storage.get_object(BUCKET_NAME, key).decode('UTF-8')
            f.write(mpileup)
            del mpileup
```

Code 11: getObject() function

Basically, what this function did was call the `get_object()` from `lithops` storage API to download the `mpileup` file according to its key and write it to the final `mpileup` file. Therefore, this function was executed X times according to the number of input `mpileup` files for the reduce function in parallel with the use of 20 threads.

Once this optimization was implemented, the same experiment that was performed for the first version was also carried out for this optimization. Ten executions were performed for each number of `fasta` chunks [1,2,5] with the same input as the first experiment.

These were the results obtained:

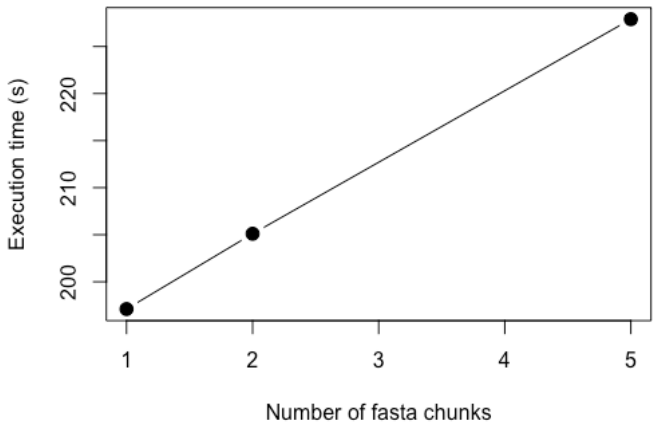

Figure 7: Second experiment results (Optimized vanilla version)

The results obtained continued to show very poor performance. Even so, the goal of this optimization was achieved as the execution time was greatly reduced.

These two plots separately were difficult to compare. Therefore, a final experiment was performed to check the performance of the entire pipeline using the two versions of this.

The goal was clear, the execution time was compared in a single plot to have an overview of the improvement offered by the optimized version. The input for each version was the same `fasta` and `fastq` file with the same amount of chunks for each one. For each version ten replicas with different number of `fasta` chunks [1,2,5,10,20,40,80,99] were executed. We compared the execution time for each case.

In this plot we can see the results obtained:

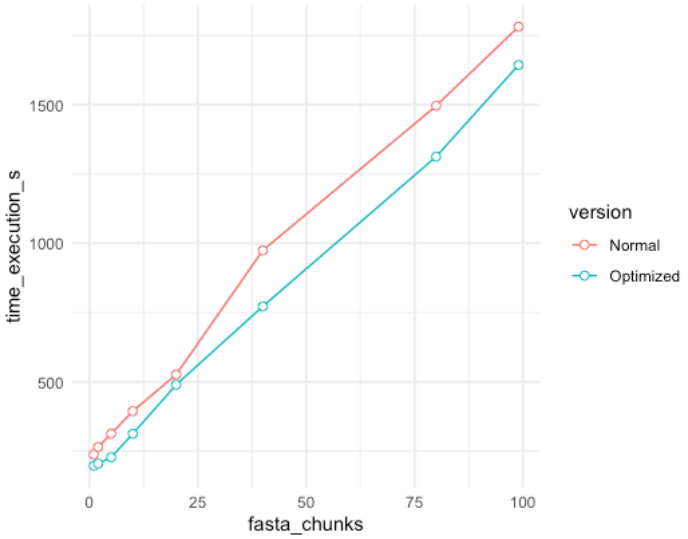

Figure 8: Normal vs Optimized vanilla version

And with this table, the results can be observed in more detail:

# Fasta chunks	Execution Time (s) Normal	Optimized
1	238,4	197,1
2	264,9	205,1
5	312,7	227,9
10	394,4	312,6
20	526,3	489,7
40	974,7	772,7
80	1496,3	1313,1
99	1781,7	1643,5

Table 3: Normal vs Optimized vanilla version experiment results

The results obtained showed that the optimized version offered a small improvement in the execution time of the demo. This improvement was of about 10-20% compared to the first version. In conclusion, the parallel download of the `mpileup` files in the reduce function offered an improvement in performance by reducing the total execution time of the pipeline. This way, the timeout error that could arise in each reducer was prevented.

In the results it could also be observed that from twenty `fasta` chunks the execution time was almost twice as the previous one. This high execution time was due to the fact that cloud

providers typically have a limit of functions that can run simultaneously (one thousand functions for IBM Cloud). From twenty `fasta` chunks, more than two thousand map functions were invoked at the same time. Therefore, once one thousand functions were run, the rest had to wait until the rest finished.

Once an optimized version of the demo was created and tested with a small input data, the execution time of the reduce function was the main point to investigate.

The execution time increase along with the number of `fasta` chunks did not make sense. Thanks to the map function of `lithops` that allowed to executed functions simultaneously, the goal to achieve was that the execution time was practically the same for each reducer, but in no case it had to increase.

In order to verify what was happening within the reduce function, an experiment was performed. The experiment was based on studying the different parts of the reduce function separately to obtain the execution time of each of them.

The function was divided into the following parts:

1. Download `mpileup` files (I/O time): Measure the time that the reduce function took to download all the `mpileup` files.
2. Sort and `sinple` script (computation time): Measure the time that the reduce function took to execute the following command lines:
    - Sort
    - Awk
    - `sinple` application

To perform this experiment, [1,5,10] reducers were invoked processing the same `mpileup` files (300MB in total) to compare which of the four parts was causing this problem. Also, each case was executed ten times in order to obtain reliable execution times.

These were the results obtained from the experiment:

	Execution Time (s)			
# Reducers	getObject	Sort	Awk	sinple
1	21,36	54	9	8
5	26,4	126	17	19
10	31	261	35	28

Table 4: Reduce function experiment results - 300MB input (IBM)

With these results, we got a broader view into the execution time problems. To get an idea, the results had to be the same for each case. On the other hand, as can be seen, all parts suffered from an increase in execution time. With this separation, the results were seen in more detail. Specifically, the computation part, specifically that in which the `mpileup` file was sorted, was the one that got a bigger increase in execution time compared to the others.

One of the reasons for this problem was because of the type of account we had. This account was a student account which had the IBM Cloud free tier. This level not only had minimal cloud services, also the quality and performance of these services was not very good compared to the payment levels. In conclusion, as more functions were invoked during the execution of the demo, these lost performance because this free tier offered limited computation for the cloud functions.

Therefore, the problem was the free tier of the cloud provider. An increase of the tier of the account had to be proposed which led to the payment for better cloud services. This tier improvement was necessary in order to achieve the goal of the project to be able to create a *serverless* pipeline that could process large amounts of data. Performance issues with the free tier already appeared already with very small data. The solution would clearly not be scalable for bigger input sizes. Thus, continuing with this tier would not allow the correct progress and development of the project.

7 Migration to Amazon Web Services

In this section we will introduce the whole process of migrating the pipeline from IBM Cloud to Amazon Web Services focusing on the reason why it was done, the necessary modifications in the implementation and the new experiments of the vanilla version in Amazon AWS.

At one of the meetings, the idea of choosing a cloud provider was put on the table in order to create a common account to control the overall cost of the experiments that each member of the project was going to perform. The goal was to compare which cloud provider could offer the better performance by running the pipeline and the experiments on a large scale. The use of AWS was introduced by one of the project coordinators with previous experience in using different clouds.

7.1 Amazon Web Services

For this cloud provider we were going to use the same services as in IBM Cloud: cloud storage (AWS S3[22]) and *serverless* functions (AWS Lambda[23]). Therefore, the performance that each cloud offered in each of these services were compared. This was done thanks to the benchmarks integrated evaluated in `lithops` framework.

7.2 *Serverless* Functions Performance Comparison

`lithops` benchmarks on *serverless* functions were observed. The test was based on the calculation of FLOPS, floating point operations per second. For each cloud provider, the execution time of one thousand parallel functions processing Giga-FLOPS was compared.

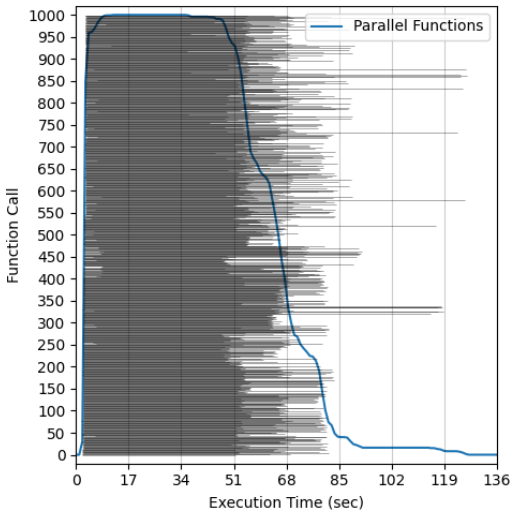

Figure 9: Execution time (s), CPU-bound operations - IBM Cloud Functions[13]

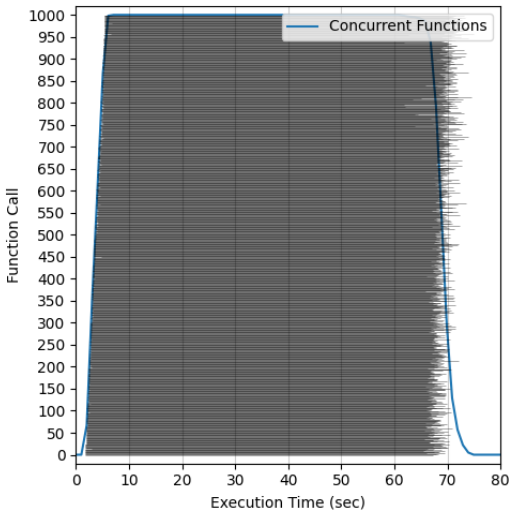

Figure 10: Execution time (s), CPU-bound operations - AWS Lambda Functions[14]

These plots showed that AWS Lambda functions offered better stability and better performance on execution time. A large level of real parallelism was observe, and lower variability

[13] extracted from https://github.com/lithops-cloud/applications/tree/master/benchmarks/flops/ibm_cf
[14] extracted from https://github.com/lithops-cloud/applications/tree/master/benchmarks/flops/aws_lambda

in execution time, for the same operations and the same number of functions.

7.3 Cloud Storage Performance Comparison

Storage services were also compared. For this test, 1MB blocks of random data were generated through the `numpy` library. One thousand functions were invoked for reading and writing 512MB objects formed by the previously generated blocks. The execution time of the was compared for each cloud provider.

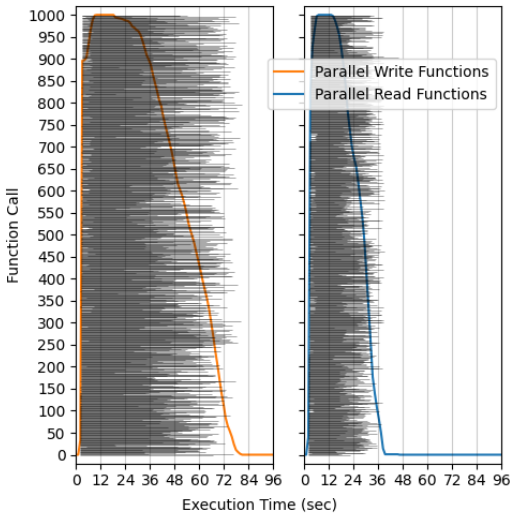

Figure 11: Execution time (s), I/O-bound operations - IBM Cloud Object Storage[15]

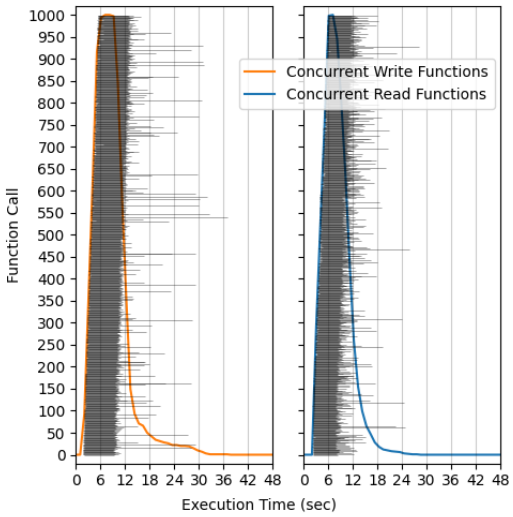

Figure 12: Execution time (s), I/O-bound operations - AWS S3[16]

As can be seen, read and write operations using S3, offered much higher performance than in the IBM Cloud.

7.4 Pipeline Modifications

The idea of staying in IBM Cloud didn't make sense with the great performance offered by AWS. Therefore the migration of the demo to this last cloud was decided.

The first modifications were made to the Docker image, where the specific libraries to run AWS services had to be changed. We had to correct multiple errors caused by the different functioning of the Lambda functions.

One of the biggest problems was that Lambdas functions have a read-only file system. So the option to write to a single file all the contents of the `mpileup` files received by a reducer was not possible. We identified that the only option for writing in the Lambda functions was using the temporary directory, also known as tmp. The execution method was simple, every time a file had to be written, the current directory of the function had to be changed to the

[15] extracted from https://github.com/lithops-cloud/applications/tree/master/benchmarks/object_storage/ibm_cos
[16] extracted from https://github.com/lithops-cloud/applications/tree/master/benchmarks/object_storage/aws_s3

tmp directory.

These changes in the implementation were added to change the directory:

```
wd = os.getcwd()
os.chdir("/tmp")
```

Code 12: Change to tmp directory

In this code snippet we can see the use of the `os` library, which allows to execute operations of the operating system. In this way, the current directory could be obtained, change it and once the operations in the tmp directory have been performed, change back to the directory were we were before.

This small change solved all the problems that had arisen before. The only drawback was that this directory had a maximum capacity of 512MB. The increase in this capacity went hand in hand with additional cost. Therefore, we predicted that the free capacity limit of this directory was going to be a future bottleneck when we wanted to scale the input data.

Once the vanilla version was up and running for AWS the following experiments to test its performance were performed.

7.5 Sort Problem

As we have seen in previous sections, one of the problems of the IBM Cloud functions was the loss of computation as the number of chunks to be processed increased. This problem was seen in the performance of the sort command in the bash script. To check if this problem was still occurring, the following experiment was carried out.

This consisted of splitting the reduce function to compare the execution time of each part. In this case, it was divided in only two parts, downloading the `mpileup` files and executing the script. Different numbers of reducers [1,5,10,20] were run with an input of 45MB. Each execution was performed ten times.

The results obtained were as follows:

	Execution Time (s)	
# Reducers	getObject	Script
1	4,5	12,1
5	4,3	12,3
10	4,6	12,2
20	4,5	12,3

Table 5: Reduce function experiment results - 45MB input (AWS)

As can be seen in the table, the results obtained were as expected, functions having the same execution time with the same input regardless of the number of parallel functions. Therefore, we were able to conclude that the problem appeared at IBM Cloud was due to the loss of computational power offered by the free tier account. With AWS not only we

solved it but also improved the performance of these functions.

When testing an input larger than 300MB per function –the maximum input tested in IBM– a new problem appeared: the capacity of the tmp directory had reached its limit.

7.6 Temporary Directory Problem

At first, the reason for this problem was not understood since the initial input of the reducer did not reach this limit. Therefore, there was some operation inside the reduce function that was writing to the tmp directory without our knowledge. We deduced the only cause that could do that had to be the script. Through unit testing, we resolved that the sort command was the cause of the problem.

```
sort --parallel 3 -T . -k1,1 -k2,2n
```

Code 13: Sort bash command

What this sort did was to sort the input data by separating it into temporary files stored in the directory specified with the `-T` argument. In this case the files were saved in the current directory (tmp) and then not deleted. The tmp directory held the `mpileup` file and the temporary files created by this command.

This error was solved in this way:

```
sort --parallel 3 --buffer-size="\$buffersize"M -T /nonexistant/dir -k1,1 -k2,2n
```

Code 14: Sort bash command solved

The directory in which the files were stored had to be changed to a non-existent directory, so that instead of writing to tmp, it was written to memory. In addition, a new argument `--bufer-size` had to be added to introduce the memory that the script would use with respect to the total memory of the function to perform the sorting. To check the correct functioning of the script, the last experiment was repeated, but now using a 300MB input.

The results obtained were as follows:

	Execution Time (s)	
# Reducers	getObject	Script
1	7,2	27,53
5	6,9	27,41
10	7,1	27,24
20	7,2	27,52

Table 6: Reduce function experiment results - 300MB input (AWS)

Finally, with these functional results we could say that the pipeline was migrated and running on the new cloud provider AWS.

7.7 Optimizations

Once the demo was up and running, some experiments were carried out to try to optimise this version on AWS.

7.7.1 Optimisation of the Sort Command

In this case, a new way of sorting the `mpileup` file was proposed to improve the results of the bash-integrated sort command. An alternative option that was implemented through the `Pandas` library, a powerful python framework for data analysis and manipulation.

An example of how to sort with `Pandas` is the following:

```
import pandas as pd

data = #mpileup file contents
df = pd.DataFrame(data)
df.sort_values(by=['Index'], inplace=True)
```

Code 15: Example of ordering with Pandas

The `mpileup` file contents were stored in memory and a DataFrame was created from it. A DataFrame is a two-dimensional data structure with rows and columns. The `mpileup` file is based on a TSV[17] format and is therefore perfectly adapted to the `Pandas` library as it is designed to analyze structured data. From this DataFrame the stored information can be analyzed or manipulated. In this case, the `mpileup` file content was sorted by the Index column using the sort_values() function.

Once this new implementation was developed, a new experiment was performed. This experiment consisted of taking the execution time using `Pandas` sort method vs the bash-integrated sort command. The execution was based on running ten times different number of reducers [1,5,10, 20] with the different `mpileup` file sizes for each case [50MB, 150MB, 300MB].

The results obtained were as follows:

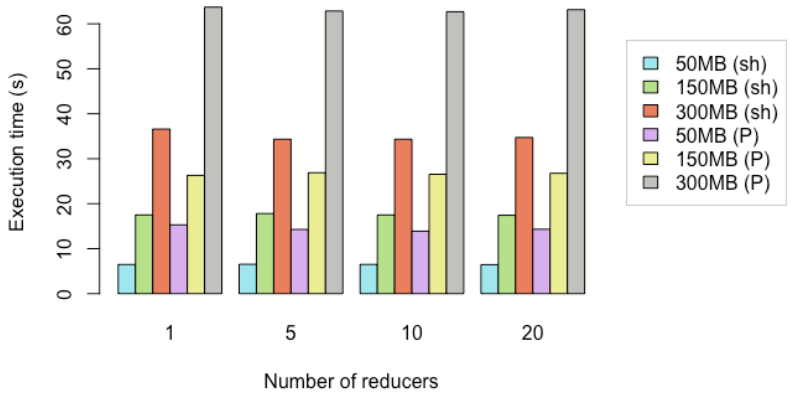

Figure 13: Sort command (sh) vs Pandas sort experiment results (Optimized vanilla version)

[17] TSV: Tabulator Separated Values, file format for storing data in tabular structures

The results were not as expected. The `Pandas` sort method involved a longer execution time. Therefore, this optimisation was discarded and the previous sort method was chosen.

7.7.2 Memory Evaluation

The last experiment with the vanilla version consisted of an evaluation of runtime memory for the reduce function.

This experiment was based on the following points:

- Setting a safe zone to avoid memory errors while running the script.
- Setting a runtime memory of 2048MB and testing with different inputs sizes to check the number of memory errors that appear.
- Finding the smallest runtime memory to be able to process 500MB (maximum input data volume) without returning memory errors.
- Performing the same execution with different runtime memories to see the performance improvement it offers.

7.7.2.1 First Safe Zone Evaluation

To perform this first test the percentage of memory used by `--buffer-size` argument of the sort command of the script was modified. The test percentages were [30%, 40%, 50%, 60%, 70%, 80%] and for each case ten functions were invoked with the same input 300MB and with the same runtime memory of 2048MB.

The results obtained were as follows:

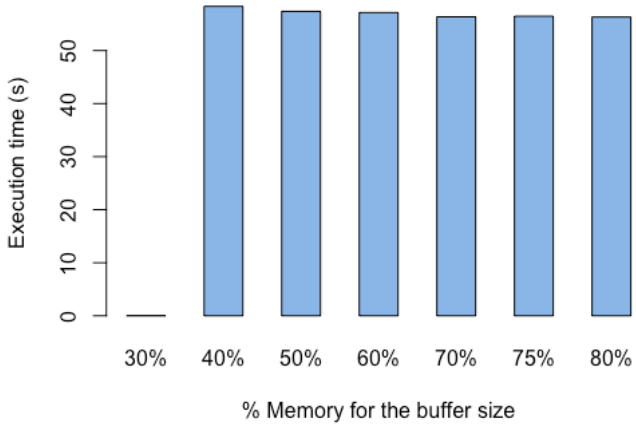

Figure 14: First safe zone experiment results (Optimized vanilla version)

These results did not provide a clear conclusion. Basically, the safe zone was between 40% and 80%. The only percentage that returned an error was 30%. This range was too wide and too imprecise to give a more concrete solution.

7.7.2.2 Second Safe Zone Evaluation

In order to try to reduce the safe zone range, another experiment was carried out based on the execution of the reduce function by increasing the input data. This input was increased from 35Mb by 25MB from 300MB up to 500MB. The runtime memory used was 2048MB and was tested for these two percentages [50%, 75%]. Finally, the number of errors obtained in each execution was compared.

The results obtained were as follows:

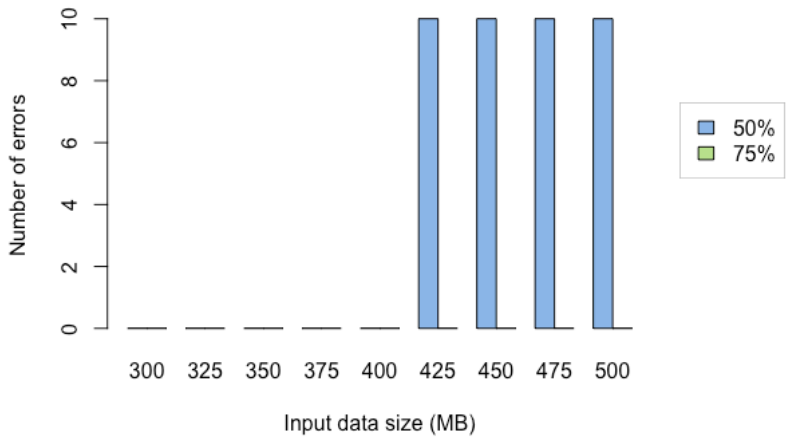

Figure 15: Second safe zone experiment results (Optimized vanilla version)

As can be seen, there were no errors with an input between 300MB and 400MB regardless of the percentage of memory used. But, from 425MB upwards, it was clear that using 50% memory was not enough to sort that volume of data. Finally, we were able to conclude that using 75% of the memory would not give any memory errors and would therefore be the safe zone.

7.7.2.3 Minimum Runtime Memory Evaluation

This test was performed to check what was the minimum runtime memory needed to be able to run the reduce function without getting any memory errors. The input data tested were the maximum allowed for a single reducer, in this case 500MB. The memory percentage for the buffer size was what we called the safe zone [75%]. We started with a runtime memory of 2048MB and decreased from 256MB.

This experiment was quite fast, as on the first attempt with 1792MB it returned an error. In conclusion, the minimum memory required to run the reduction function was 2048MB.

7.7.2.4 Performance Evaluation

In this case, the performance obtained by increasing the minimum execution memory by 512MB was evaluated. This value was increased to a maximum of 4604MB. Ten iterations were performed for each case. The input data was 500MB and the percentage for the buffer size was 75%. To evaluate this improvement, the speedup of the execution time against the

minimum memory configuration was compared.

$$Speedup = Executiontime(2048MB)/Executiontime(Increasedmemory) \quad (1)$$

The results obtained were as follows:

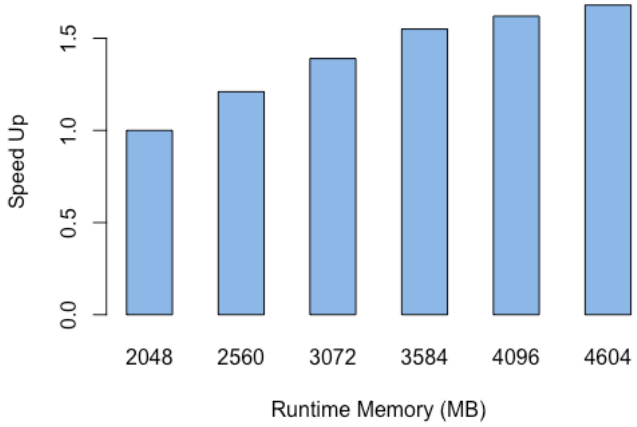

Figure 16: Performance experiment results (Optimized vanilla version)

As the execution memory increased, the speed up also increased. It offered a 50%-60% improvement from 3584MB over the initial execution time. From 4096MB the improvement was practically minimal. Therefore, the runtime memory that gave us the best performance was 4096MB.

8 Development - Definite Version

In this section we will report the enhanced version of the distributed reduction stage, which is integrated the last implementation of the pipeline. The design, the architecture, the implementation with code snippets and some brief experiments will be reviewed.

8.1 Design

This version is developed using the AWS cloud provider. Basically, the following services were used:

- S3: Service to store objects/data in the cloud.
- Lambda functions: Service to run application code in *serverless* functions.

The main objective of this version was to improve the vanilla architecture to obtain a scalable pipeline that could process large volumes of data. To his end, the following proposal was discussed at the meetings:

- Invoking multiple reducers to process a single `fasta` chunk.
- Splitting the contents of the `mpileup` files in each of these reducers.
- Merging all partial results to create a final result.

The straightforward solution was to split the `mpileup` files with a byte ranges and assign a range to each reducer. Each line of the `mpileup` file consisted of a different number of bytes. Therefore, there was a complexity in splitting the files without losing or cutting any rows.

Besides, a limitation appeared due to the functioning of the `SiNPle` application algorithm. Each `mpileup` file was made up of indices, and all the rows for the same index had to processed jointly (something close to a non-associative reduceByKey).

As the map stage was implemented and the project runners did not contemplate the possibility of modifying it, we had to implement an intermediate ”sampling step” on the mapper results to distribute the indexes across reducers. We needed a method to assign indexes to reducers in a load-balanced manner –that is, each reducer processing a similar number of rows. Once the problem was defined, the proposal was modified:

- Getting all the indexes of each `mpileup` file.
- Creating ranges of indexes to be processed by the reducers.
- Merging all partial results together one after another.

Aside from this solution, we still had the problem of reading bytes. In this case, the exact position of the indexes had to be obtained and not only that, the size of the row had to be controlled. This problem required some changes in the map function with a lot of complexity that could have been time consuming.

In a meeting with one of the project coordinators, the idea of using S3 Select emerged, which would allow us to query the objects stored in S3. But to do so, the format of the `mpileup` files had to be changed to CSV or parquet.

8.2 S3 Select

S3 Select is a AWS service that enables applications to get only the data you need from an structured object on S3 by using SQL queries. Queries are run in the server side, releasing the user from SQL implementations, and are billed based on the amount of data processed. This service has some limitations, for example, the format of the file in which the queries are performed and the type of queries, which only allow filter operations.

8.3 File Format

As our definite implementation supports both the CSV and the Parquet format, we will have a quick review of both options and their main features.

8.3.1 CSV Format

A Comma Separated Values (CSV) file is a plain text file that contain registers separated by newlines and fields separated by commas, in a classic table-like structure. It is one of the most popular formats worldwide as it is one of the most widely used formats in office applications such as Microsoft Excel and others.

8.3.2 Parquet Format

A parquet file is a column-oriented data file to guarantee an efficient data storage and manipulation.These data are compressed and encoded to avoid large accumulations of data in local or cloud storage.

To understand how it works, we will see the following example:

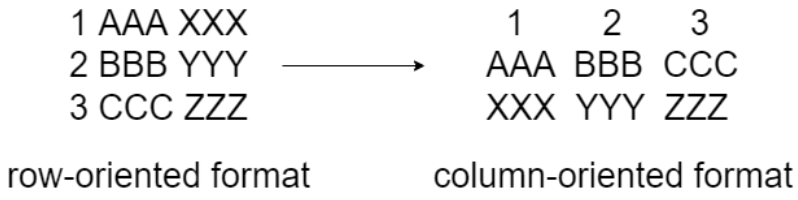

Figure 17: Row-oriented format vs Column-oriented format

This column-oriented format allows to obtain all the information of a column in a single request, as column data is disposed adjacently. For the case of the `mpileup` file, this format eased the extraction of the index column.

8.4 Architecture

Once the formats used have been explained, the architecture developed from the proposal seen above will be explained in more detail.

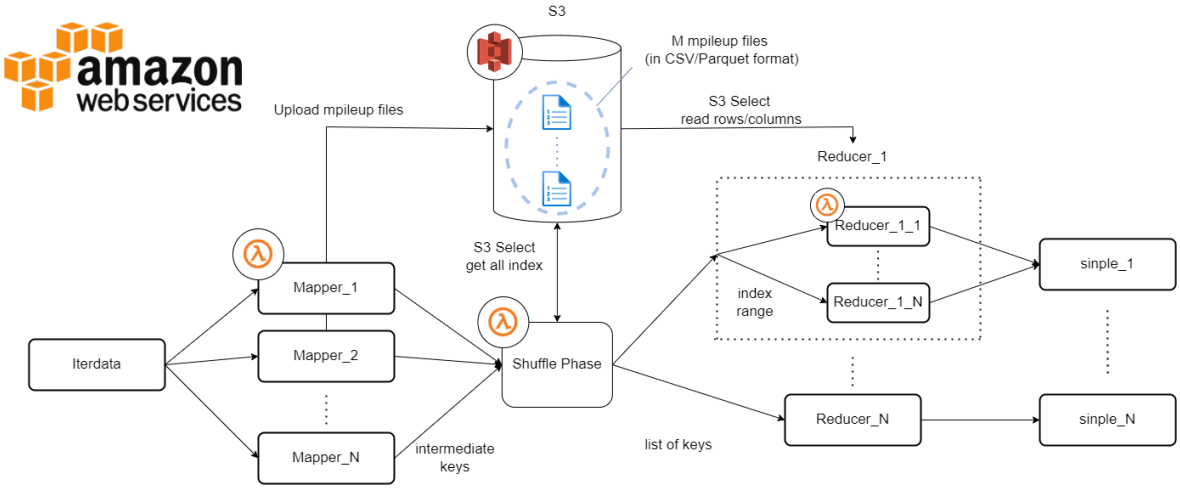

Figure 18: Enhanced/Current version architecture

We can differentiate five main steps within this architecture:

1. Input data partition: The iterdata (input data) is created for the mappers.

2. Map phase: (X`fasta`$chunks * Y$`fastq`$files$) number of mappers will be invoked to process the input data in order to create the `mpileup` files. A mapper will consist of a single Lambda function that will execute the map function code.

3. Shuffle phase: In this phase, the data input necessary to process each `fasta` chunk will be created. In order to invoke multiple reducers for a single chunk, S3 Select will be used to obtain all the indices in the `mpileup` file and assign them to each reducer.

4. Reduce phase: R reducers will be invoked where R will be the necessary number of reducers capable of processing all the `mpileup` files. A reducer will consist of a single Lambda function that will execute the reduce function code. To get the information from each `mpileup` file, S3 Select will be used according to extract the rows corresponding to the previously assigned indexes.

5. Output: In this case a merge of all partial results assigned to a single `fasta` chunk will be made. However, the final results of each chunk will not be merged.

After this overview of the architecture and its steps we can focus on the implementation of the current version of the pipeline.

8.5 Implementation

For the implementation of the current version, changes are made in the map function, in the reduce function and in the MapReduce class previously implemented and modified.

8.5.1 Map Function Modification

The modification of this function is based on changing the `mpileup` file format to parquet or CSV. This can be easily done thanks to the `Pandas` library explained above. The content of the `mpileup` file must be converted into a DataFrame and then to the chosen format. An example of how to convert to CSV or Parquet would be the following:

```
import pandas as pd

#Convert mpileup to DataFrame
data = #mpileup file contents
df = pd.DataFrame(content)

#DataFrame to Parquet
df.to_parquet(key)

#DataFrame to CSV
df.to_csv(key, index=False, header=False)
```

Code 16: Example to convert a dataframe to CSV or parquet with pandas

Once the file format required is created, the following lines are used to upload it to S3:

```
with open(key, "rb") as file:
    storage.put_object(BUCKET_NAME, intermediate_key, body=file)
return intermediate_key
```

Code 17: Upload CSV or parquet file to S3

Where `bucket_name` is the name of the storage in S3 where the file is saved, `intermediate_key` is the name in which the file is saved and finally `body` is the content of the file to be uploaded to S3.

These are the small modifications necessary for the map function for the correct functioning of the pipeline.

8.5.2 Reduce Function Modification

The main objectives of the new reduce function are as follows:

- Extracting all rows from the `mpileup` files in parquet or CSV format from the assigned index range with S3 Select.
- Converting extracted data to `mpileup` format.
- Running the SiNPle application with the `mpileup` file created as input.

The way `mpileup` files are extracted has been modified. In this case, the file is no longer downloaded but the necessary information is obtained from a range of indexes. S3 Select is used for this purpose. In the following code snippet we will see how the information can be taken through S3 Select:

```
s3 = boto3.client('s3')

 if file_format == "csv":
     expression = "SELECT * FROM s3object s WHERE cast(s._2 as int)
                           BETWEEN %s AND %s" % (range['start'],
                           range['end'])
     input_serialization = {'CSV': {}, 'CompressionType': 'NONE'}

 elif file_format == "parquet":
     expression = "SELECT * FROM s3object s WHERE s.\"1\"
                           BETWEEN %s AND %s" % (range['start'],
                           range['end'])
     input_serialization = {'Parquet': {}, 'CompressionType': 'NONE'}

 for k in key:

     resp = s3.select_object_content(
         Bucket=BUCKET_NAME,
         Key=k,
         ExpressionType='SQL',
         Expression=expression,
         InputSerialization = input_serialization,
         OutputSerialization = {'CSV': {"FieldDelimiter" : "\t"}}
     )

     data = ""
     for event in resp['Payload']:
         if 'Records' in event:
             records = event['Records']['Payload'].decode("UTF-8")
             data = data + records
```

Code 18: Using S3 Select to extract data from mpileup files converted to CSV or parquet format

As we can see, the SQL expression changes according to the file format, that is due to the structure in which the data is stored. In this case, the data is read from storage sequentially leaving behind the optimization performed in the vanilla version. In the `resp` variable the response of the S3 Select service is obtained. The result of the query is included in the `PayLoad` field, containing the `mpileup` rows to process.

Arguments to highlight can be the following:

- ExpressionType: Type of S3 Select query expression. In this case SQL.
- Expression: SQL query.
- InputSerialization: Input file format.
- OutputSerialization: Output file format.

The following lines of code are used to reconvert the S3 Select response into a `mpileup` file:

```
 with open(mpileup_file, 'a') as f:
         f.write(data)
```

Code 19: Saving S3 Select response into a mpileup file

This step is simple, the response saved in the data variable is written in the selected `mpileup` file.

To avoid the accumulation of partial files and to add a new function where a merge of these files will have to be done, the AWS multipart upload service is used. Multipart upload allows the user to upload a file by uploading parts of that file. This way we can upload the parts related to a single `fasta` file without the need to add a new function for it.

Multipart upload is based on three phases:

1. Creation of the multipart upload.
2. Parts upload.
3. Completion of the multipart upload.

For the reduce function, the parts upload is implemented and the snippet of the code is as follows:

```
    s3 = boto3.client('s3')
    part = s3.upload_part(
        Body = sinple_out,
        Bucket = BUCKET_NAME,
        Key = mpu_key,
        UploadId = mpu_id,
        PartNumber = n_part
    )
```

Code 20: Uploading a part to S3 using Multipart Upload

To upload a part, the S3 function `upload_part()` containing the following arguments is called:

- Body: File contents. In this case, `sinple` result.
- Bucket: Name of the bucket where the file will be stored.
- Key: File name.
- UploadId: Id of the multipart upload to perform the upload.
- PartNumber: Part number to be uploaded.

These are all modifications to the reduce function. Finally, we will explain the changes in the logic of the MapReduce class to be able to invoke these functions correctly.

8.5.3 MapReduce Class Modifications

The component that has undergone the most changes is the MapReduce class. Basically, a large part of the logic has to be changed in order to be able to invoke multiple reducers for each chunk of the `fasta` file.

The new logic is based on a load balancer that has to fulfill the following steps:

- Extracting the index columns from all `mpileup` files stored in parquet or CSV format with S3 Select.
- Creating a dictionary of {"index": #rows}
- Assigning to each reducer a maximum number of rows to process in order to distribute a range of index to each.

In order to accomplish these steps, several functions are implemented to create the input data for each reducer. Now we will see the work performed by each of them:

- Create_intermediate_keys(): This function creates a list of multiple lists of `mpileup` files. Each list is a list of `mpileup` files related to a `fasta` chunk.
- Create_sinple_key(): This function creates a list of result keys to store the `sinple` output file for each reducer.
- Get_s3_select_indexes(): This is the main function that is in charge of using S3 Select to obtain the index columns. We will refer to it as the load-balancer function, as it is in charge of distributing the rows equally among reducers. It generates a dictionary with index: #rows from all `mpileup` files, in order to get a list of the index ranges that will be read by the reducers.

For the multipart upload part implementation:

- Create_multipart(): This function starts the multipart upload service from the `sinple` keys. Returns an id with which each of the parts can be uploaded.
- Complete_multipart(): This function completes each of the parts in order to consider the total upload of the file as effective.

Finally, a last function is invoked to collect all the information obtained from the previous functions to create the input data necessary for the invocation of each reducer. We will see an example of the data that the `create_iterdata_reducer()` function returns as input for the reducers. In this case we will see a `fasta` chunk divided into two reducers:

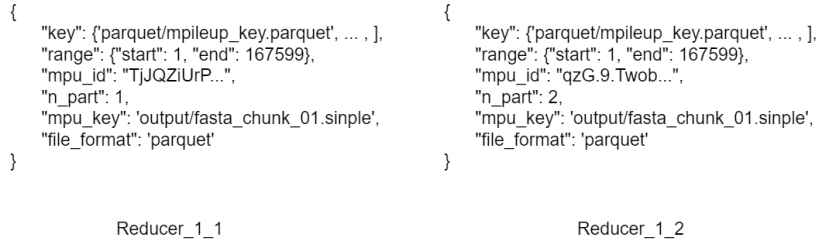

Figure 19: Iterdata example for the reducers

As can be seen there are the following common parts:

- key: List of `mpileup` files.
- mpu_key: Name of the resulting file which consisted of all uploaded parts.

- file_format: Common file format for the entire pipeline execution.

The parts that vary according to the reducer are:

- range: Index range assigned to each reducer.
- mpu_id: Unique identifier for the upload of each part.
- n_part: Part number.

Finally, the function of invoking reducers with different runtime memories is eliminated. It is replaced by the runtime memory that offers the best performance as seen in previous experiments. In this case it is 4096MB as runtime memory.

Then the reducer phase is modified in this way:

```
1    fexec = lithops.FunctionExecutor(runtime=self.runtime,
2                                     runtime_memory=self.runtime_memory_r,
3                                     log_level=self.log_level)
4    fexec.map(self.reduce_func, iterdata, timeout=self.
   func_timeout_reduce)
5    results = fexec.get_result()
```

Code 21: Reduce phase (2)

Basically, from the `lithops` executor different reduce functions can be invoked with the same runtime memory that will be in charge of processing the data previously processed as iterdata.

Having seen the whole implementation, we will get into the testing and evaluations to which the definite version is subjected.

8.6 Experiments

To test the new improved version of the pipeline, two experiments are conducted based on the performance offered by S3 Select and the additional cost obtained by using this service.

8.6.1 Performance Evaluation

In this test we evaluate the performance of reading and processing N `mpileup` files with a single reducer using the vanilla version through the Standard GET vs using R reducers with S3 Select of the enhanced version.

8.6.1.1 Reading Performance Evaluation

To start with, only the reading execution time is evaluated for both architectures. The steps are the following:

1. Measuring the total time for reading all the files from a single reducer (vanilla version).
2. Generating a load-balanced index set for each of the R reducers (so that everyone reads approximately the same amount of data).

3. Measuring the total time of reading all the files from R reducers. On each reducer will only read the rows with S3 Select according to the assigned indexes.

The reading of [10,50,100] 3GB `mpileup` files are evaluated using different numbers of reducer [1,2,4,8]. For each case ten iterations are performed with a runtime memory of 4096MB. For a single reducer, `mpileup` files are used. For the multi-reducer architectures, files are converted to parquet format to use S3 Select.

The results obtained are as follows:

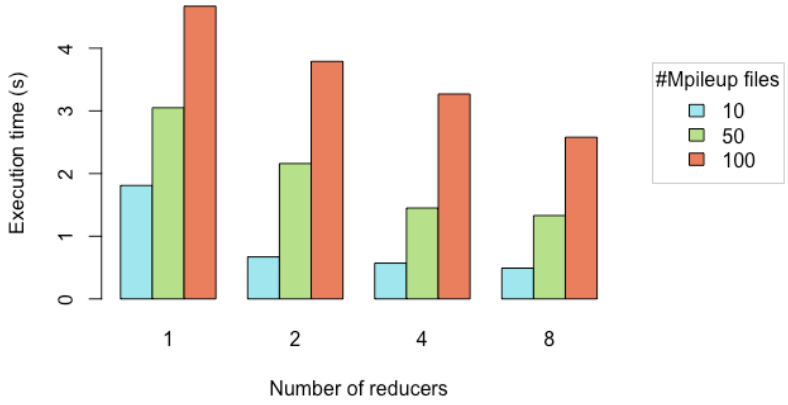

Figure 20: Reading performance evaluation results

As a conclusion we can say that using S3 Select offers better read performance than using the Standard GET. These results are favorable for S3 Select because the files were split into [2,4,8] reducers so each reducer reads X indexes instead of the entire file. Therefore, the results are as expected.

8.6.1.2 Reading & Processing Performance Evaluation

For this experiment the same steps discussed in the previous evaluation are followed but the processing time when executing the SiNPle application to obtain a sinple result is added.

The characteristics are the same from the previous experiment but the CSV format is added to have a better comparison between CSV and parquet format.

The results obtained for parquet format are as follows:

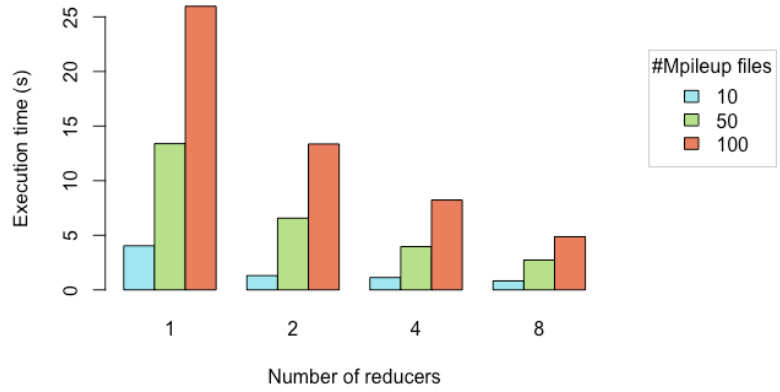

Figure 21: Reading & processing performance evaluation results (Parquet)

Here we can see the big difference between using a single reducer compared to multiple reducers. The execution time to process a single file with only one reducer is much larger than using different reducers to process the same file. For two reducers this difference is of 50% and for eight reducers it can reach 75%.

The results obtained for CSV format are as follows:

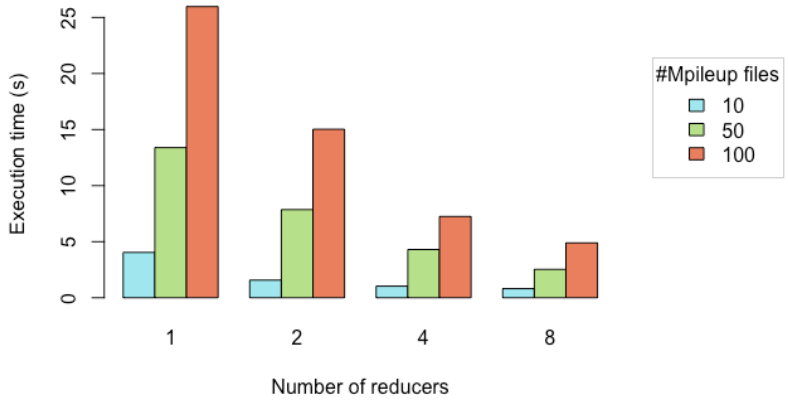

Figure 22: Reading & processing performance evaluation results (CSV)

The results obtained are very similar to those obtained with the parquet format. We can say that for both CSV and parquet format the distribution of the `mpileup` files in different reducers offers better performance than using a single reducer.

As the results are very similar, the two plots are combined to directly compare the CSV and parquet results:

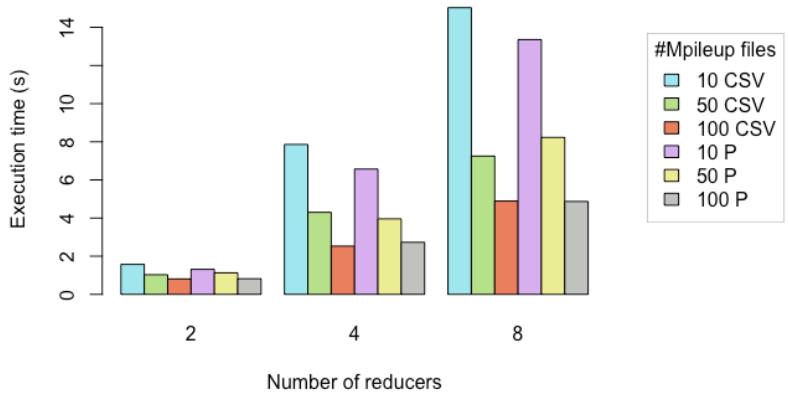

Figure 23: Reading & processing performance evaluation results (CSV vs parquet)

As a final conclusion, we can say that parquet offers a little better performance than CSV. Not only that, parquet files stored in S3 has smaller size than CSV files thanks to the compression and encoding of these. Therefore, parquet offers a better execution time and larger storage space.

8.6.2 Cost Evaluation

This experiment are based purely on calculations, any execution is performed in this case, all calculations are done manually and approximately. For this case the cost related to lambda functions' execution time is not considered, as we only include the cost of the requests.

For the single reducer option (standard GET operations), the calculation will include the following cost breakdown:

- Bandwidth cost (total GBs returned).
- Throughput cost (number of GET requests performed).

For the multiple reducer case (S3 Select), the calculation will include the following cost breakdown:

- Bandwidth cost (total GBs returned by S3 Select service).
- Throughput cost (number of GET requests performed).
- Data scanning cost (total volume of data processed).

Looking at the Amazon S3 Prices[24], the following table is created:

Cost type	Price
Bandwidth	7×10^{-4}\$ / GB
Throughput	4×10^{-4}\$ / 1000 Requests
Data scanning	2×10^{-3}\$ / GB

Table 7: Amazon S3 Prices (1)

For cost calculations, we assume that the bandwidth cost is 0$, as the cost of transferring data inside private networks (using private endpoints) is free for most cloud providers. We calculate the price of obtaining [10,50,100] `mpileup` files of 3GB each, using [1,2,4,8] reducers. In this experiment we do not exceed 1000 request, for this reason, the price of a single request will be (4×10^{-4}/1000) because we assume that the pipeline will be executed several times.

The results obtained are as follows:

	Cost ($)		
# Reducers	10 mpileup	50 mpileup	100 mpileup
1	4×10^{-6}	2×10^{-5}	4×10^{-5}
2	5.95×10^{-5}	2.97×10^{-4}	5.95×10^{-4}
4	9.86×10^{-5}	4.93×10^{-4}	9.86×10^{-4}
8	1.76×10^{-4}	8.83×10^{-4}	1.76×10^{-3}

Table 8: Cost evaluation results (Parquet)

In conclusion, the cost of using S3 Select is higher due to two factors, the first is that it is made up of two costs and the second is that the amount of data scanned is multiplied by the number of reducers.

8.6.3 Cost Reference Tables

In this section we construct cost reference tables for future comparisons. The table is created from the evaluation of the cost of reducing 300MB using the standard get for a single reducer and using S3 Select for [1 and 2] reducers with Parquet and CSV format.

The calculation will include the following cost breakdown:

- S3 GET request charges
  - S3 GET requests from the S3 Standard storage class
- S3 Select request charges
  - Total GBs returned by S3 Select service.
  - Total volume of data processed.
- Lambda charges
  - Lambda compute charge
  - Lambda request charge

Looking at the Amazon S3 Prices, the following table is created:

Cost type	Cost ($)
S3 GET	$4 \times 10^{-4}$$ / 1000 Requests
Bandwidth	$7 \times 10^{-4}$$ / GB
Data scanning	$2 \times 10^{-3}$$ / GB
Lambda compute	1.68×10^{-5} / #req / #sec / mem allocated (GB)
Lambda request	2×10^{-4} / 1000 Requests

Table 9: Amazon S3 Prices (2)

For standard GET:

Cost type	Cost ($)
S3 GET	4.21×10^{-5}
Lambda compute	2.93×10^{-3}
Lambda request	2×10^{-7}
Total	2.97×10^{-3}

Table 10: Cost reference table - 300MB - 1 reducer (Standard GET)

For S3 Select using parquet format:

Cost type	Cost ($)
Bandwidth	2.05×10^{-4}
Data scanning	1.95×10^{-4}
Lambda compute	3.71×10^{-3}
Lambda request	2×10^{-7}
Total	4.12×10^{-3}

Table 11: Cost reference table - 300MB - 1 reducer (Parquet)

Cost type	Cost ($)
Bandwidth	2.05×10^{-4}
Data scanning	3.9×10^{-4}
Lambda compute	4.91×10^{-3}
Lambda request	4×10^{-7}
Total	5.51×10^{-3}

Table 12: Cost reference table - 300MB - 2 reducers (Parquet)

For S3 Select using CSV format:

Cost type	Cost ($)
Bandwidth	2.05×10^{-4}
Data scanning	5.86×10^{-4}
Lambda compute	3.51×10^{-3}
Lambda request	2×10^{-7}
Total	4.29×10^{-3}

Table 13: Cost reference table - 300MB - 1 reducer (CSV)

Cost type	Cost ($)
Bandwidth	2.05×10^{-4}
Data scanning	1.17×10^{-3}
Lambda compute	4.81×10^{-3}
Lambda request	4×10^{-7}
Total	6.18×10^{-3}

Table 14: Cost reference table - 300MB - 2 reducers (CSV)

As we can see, reducing using standard GET requests is cheaper than using S3 Select because there is not the additional cost that this services offers. If we compare the parquet and CSV formats using different reducers we can see how the CSV part offers a small increase in cost. This is due to the fact that the data saved in this format has a larger size compared to the data saved in parquet. This, the data to be scanned is larger.

8.7 Evaluation

In this section we will see the evaluation of the overall performance of the demo.

In this case, some modifications are added to the map function based on performing better alignment of the genomes using new services such as EC2[18][25] and Redis[19][26] to process and store the indexes and their alignments.

With EC2 we have access to more powerful compute units. AWS offers multiple instances with multiple number of cpus and more physical memory. In this way, we gain in performance and computation against the computational limits of cloud functions. An ecosystem solely dedicated to the alignment of sequences and references of genomes can be created.

With Redis a great improvement is achieved by reducing data access latency by implementing an in-memory cache instead of storing data on disk at the server side.

The total data volume is increased to see the scalability offered by the enhanced version. For this, the demo is tested with a 1GB `fasta` file divided into 18 chunks and a 1.1GB `fastq` file divided into 30 chunks generating a total of 540 `mpileup` files of 50MB each. Therefore, a total volume of 26-27GB to be reduced.

For this evaluation, a test is performed with different numbers of iterations (`mpileup` files – [5,40,540]) to be reduced. In this case we will see the total execution time and the execution times of each phase (Map, Load-balancer, Reduce).

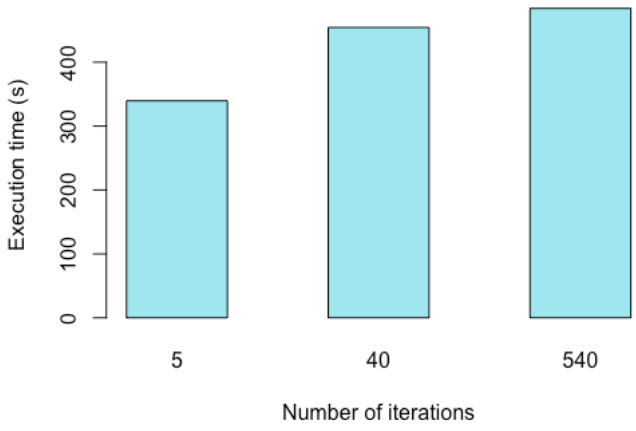

Figure 24: Total execution time - Enhanced version

[18]EC2: Amazon Elastic Compute Cloud provides scalable computing capacity from invocation of virtual servers/computing environments known as instances.

[19]Redis: Remote Dictionary Server, is an in-memory data structure storage used in distributed systems

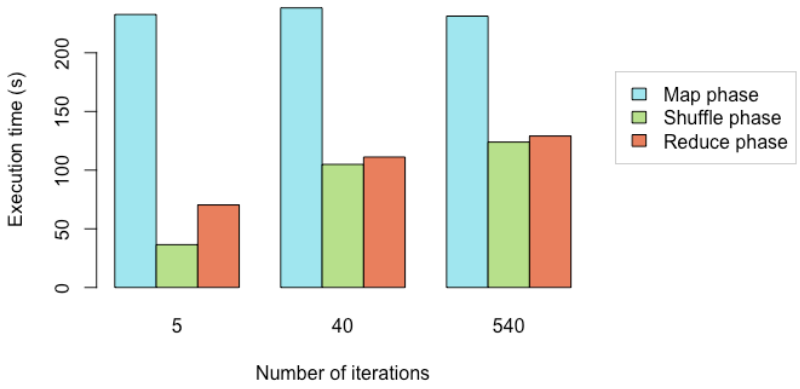

Figure 25: Execution time per phase - Enhanced version

Our results show that the execution time only increases slightly with the number of iterations. This is due to the parallelization of the functions within each phase. In this case, the map and reduce functions execution times are similar between iterations because the limit of thousand simultaneous function offered by AWS is not reached as was the case in the first IBM version, which affected performance considerably since many functions had to wait to be invoked. This number of invoked functions can be modified thanks to the creation of larger `fasta` and `fastq` chunks. Therefore, we can actively avoid this limit.

As can be seen, when the maximum number of `mpileup` per `fasta` chunk is reached, the load-balancer execution time increases. Basically this happens due to the fact that more requests are made to obtain the index column on each `mpileup` file. In addition, the more columns obtained, the larger dictionary has be created and processed.

In the following table, we will see the number of functions that were invoked to perform this evaluation:

	# Functions		
# Iterations	Map phase	Shuffle phase	Reduce phase
5	5	1	1
40	40	3	8
540	540	18	57

Table 15: Number of functions invoked for each phase - Enhanced version

As a result, we can see how M mappers are invoked according to the number of iterations, since each of them generates an `mpileup` file. For the load-balancer phase, a function is invoked for each `fasta` set to process each of these `mpileup` files related to the set. Finally, according to the load-balancer which is in charge of dividing the index ranges across the reducers, R reduce functions will be invoked.

We have been able to observe the increased scalability offered by this enhanced version compared to the first one. This version allows to reduce a volume of 26-27GB of date in 13GB assuming a higher cost for an increase in the of Lambda functions and services such

as S3 Select.

9 Conclusion

This final degree project has allowed me to enter the world of cloud computing and systems through *serverless* functions. By participating in this European project based on genomics use case, I have been able to face a real problem on this paradigm. Being day by day with cloud and genomics experts has helped me to become familiar with these services, architectures and concepts that previously caused me great doubts due to their complexity.

Now I understand in a much greater way the complexity involving big data projects. I am familiar with the difficulties that arise from designing architectures that fit the objectives of the problem and verifying that these can really offer a solution. Scalability issues, price/performance constraints and service specificities are issues that I have much more in mind after this project.

I am grateful to all the members of the CloudLab i have worked with and who have helped me throughout this project, as well as the members of the Hutton Institute. Being able to face this problem has been key in deciding my next steps in my academic career and facing my future job options.

Referring to the development process of the project, it is worth mentioning the added difficulty of communication problems with the bioinformaticians, due to the lack of specialization in cloud computing and *serverless* architectures. Even so, thanks to the multiple meetings held, we were able to agree on the steps to be followed and to continue advancing in the project.

In the process of designing and implementing the different solutions, the cost was a main limiting factor, as we had to work on a bounded cloud budget. Due to the pay-per-use billing model of the cloud, it is essential to track the active services and resources at any given moment. We were constantly looking for the most affordable option once a functional result was obtained. As a consequence, testings were iteratively planned to perform the first evaluations locally and then port them to the cloud.

It should also be added that this multiphase variant caller pipeline was modified concurrently by different CloudLab colleagues. Each of us was working on a specific phase in a modular way. Thanks to the fact that the cloud has decoupled services –i.e. specific services for each resource– it helped not to create so many dependencies between colleagues and to be able to modify this pipeline as a team.

One of the biggest problems was that the pipeline could not be scaled until the last weeks. This caused problems concerning memory management, communication between components and phase orchestration that had not been detected previously due to the low data load of the initial tests. This resulted in unplanned final meetings and some changes in the operation of various functions that would serve as patches to avoid eventual bugs.

Finally, I am proud to have delivered an architecture that allows the analysis of a large volume of genomics data, and that could be extended to its use in real projects. I encountered new technologies and services that I did not know about, and managed to offer a solution to such a specific problem as the one that was proposed.

Our final implementation of the whole pipeline, resulting of our teamwork, demonstrated that a total of 27GB could be processed in only 8 minutes. Something unreachable if this execution had been carried out on a personal machine.

10 Future Work

As we have seen in the previous section, this new version offers greater scalabilty compared to the first version. Being able to reduce a volume of data of up to 27GB. Further optimizations on the architecture should be considered to reach the data volume magnitudes of greater genomics pipelines.

During the executions different memory problems arose in the load-balancer phase. The main problem was based on the index and row dictionary that was created inside this function. This data structure made the function very demanding on memory. We should look for different ways to fix it or modify it.

The load-balancer was in charge of distributing the index ranges across the different reducers. These index ranges were created from the maximum range of rows that could be processed by each reducer. The difficulty of the problem comes from the fact that not all the indexes are found in the `mpileup` file. That is, if there are X rows/indexes in total, there are Y rows/indexes missing in each `mpileup` file. Therefore, having an exact calculation of the maximum number of rows has not a straightforward solution. The first future work would be to modify this memory-dependent load-balancer to avoid using the dictionary data structure.

One of the goals of the pipeline was also to obtain the highest performance, therefore, we also seek to reduce the execution time of each phase. If we look at the reduction phase, as a second modification would be to parallelize the `mpileup` file data download, which was already done in the first version but was modified to a sequential implementation in order to obtain the data with S3 Select. We should also evaluate the performance of the parallel read with standard GETs vs S3 Select.

It should be noted that parallelizing the reading of data with S3 Select could cause problems. The queries executed in this service are executed on servers close to S3, and possibly the computational capabilities offered by this service to extract data are limited. Therefore, there could be inconsistent reading performance as the number of parallel S3 Select requests increase, as they would compete for the same resources.

Another point to take into account is that S3 Select offers very simple SQL expressions, so the data manipulation should not be very complex. When optimizing the load-balancer with a more sophisticated algorithm, it should be considered whether S3 Select allows this manipulation. In conclusion, we doubt that S3 Select will be used in future versions due to these restrictions.

The cost of executing the pipeline is a value to be taken into account. As a final point, we should check which configurations give the highest performance at lowest cost. The cost/performance ratio was already been taken into account while developing the different versions with execution time and memory evaluations. Such evaluations should be extended to the whole pipeline.

References

[1] "CloudButton European Project," https://cloudbutton.eu/, accessed: 2022-05-26.

[2] D. C. Koboldt, "Best practices for variant calling in clinical sequencing," *Genome Medicine*, vol. 12, no. 1, p. 91, Oct 2020. [Online]. Available: https://doi.org/10.1186/s13073-020-00791-w

[3] "IBM Cloud," https://www.ibm.com/cloud, accessed: 2022-05-26.

[4] "Amazon Web Services," https://aws.amazon.com/, accessed: 2022-05-26.

[5] Armbrust, Michael, A. Fox, Armando, Griffith, Rean, Joseph, A. D, R. Katz, R. H, A. Konwinski, Andrew, G. Lee, Gunho, Patterson, D. A, Rabkin, Ariel, Stoica, and Matei, "Above the clouds: A berkeley view of cloud computing," 01 2009.

[6] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," 2019. [Online]. Available: https://arxiv.org/abs/1902.03383

[7] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona, "Toward multicloud access transparency in serverless computing," *IEEE Software*, vol. 38, no. 01, pp. 68–74, jan 2021.

[8] A. John, K. Muenzen, and K. Ausmees, "Evaluation of serverless computing for scalable execution of a joint variant calling workflow," *PLOS ONE*, vol. 16, no. 7, pp. 1–12, 07 2021. [Online]. Available: https://doi.org/10.1371/journal.pone.0254363

[9] P. Grzesik, D. R. Augustyn, Ł. Wyciślik, and D. Mrozek, "Serverless computing in omics data analysis and integration," *Briefings in bioinformatics*, vol. 23, no. 1, p. bbab349, Jan 2022, 34505137[pmid]. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/34505137

[10] B. Langmead and A. Nellore, "Cloud computing for genomic data analysis and collaboration," *Nature reviews. Genetics*, vol. 19, no. 4, pp. 208–219, Apr 2018, 29379135[pmid]. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/29379135

[11] "Fastq Format," https://en.wikipedia.org/wiki/FASTQ_format, accessed: 2022-06-03.

[12] "Fasta Format," https://en.wikipedia.org/wiki/FASTA_format, accessed: 2022-06-03.

[13] P. Danecek, J. K. Bonfield, J. Liddle, J. Marshall, V. Ohan, M. O. Pollard, A. Whitwham, T. Keane, S. A. McCarthy, R. M. Davies, and H. Li, "Twelve years of SAMtools and BCFtools," *GigaScience*, vol. 10, no. 2, 02 2021, giab008. [Online]. Available: https://doi.org/10.1093/gigascience/giab008

[14] S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca, "The gem mapper: fast, accurate and versatile alignment by filtration," *Nature Methods*, vol. 9, no. 12, pp. 1185–1188, Dec 2012. [Online]. Available: https://doi.org/10.1038/nmeth.2221

[15] "Mpileup format," htslib.org/doc/samtools-mpileup.html, accessed: 2022-06-03.

[16] L. Ferretti, C. Tennakoon, A. Silesian, and G. F. a. Ribeca, "Sinple: Fast and sensitive variant calling for deep sequencing data," *Genes*, vol. 10, no. 8, p. 561, Jul 2019, 31349684[pmid]. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/31349684

[17] "IBM Cloud Object Storage," https://cloud.ibm.com/objectstorage, acessed: 2022-05-27.

[18] "IBM Cloud Functions," https://cloud.ibm.com/functions/, accessed: 2022-05-27.

[19] "Lithops Storage API," https://github.com/lithops-cloud/lithops/blob/master/docs/api_storage.md, accessed: 2022-05-28.

[20] "Lithops Futures API," https://github.com/lithops-cloud/lithops/blob/master/docs/api_futures.md, accessed: 2022-05-28.

[21] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, "Performance evaluation of heterogeneous cloud functions," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 23, p. e4792, 2018, e4792 cpe.4792. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4792

[22] "Amazon Web Services Simple Cloud Storage (S3)," https://aws.amazon.com/s3/, accessed: 2022-05-27.

[23] "Amazon Web Services Lambda Functions," https://aws.amazon.com/lambda/, accessed: 2022-05-27.

[24] "Amazon Web Services S3 Prices," https://aws.amazon.com/s3/pricing/, accessed: 2022-06-01.

[25] "Amazon Elastic Compute Cloud," https://aws.amazon.com/ec2/, accessed: 2022-06-01.

[26] "Redis | The Real-time Data Platform," https://redis.com/, accessed: 2022-06-01.

UNIVERSITAT ROVIRA I VIRGILI