Francisco Damián Maleno González

A COMPRESSED FILE PARTITIONER FOR SCALABLE GENOMICS ANALYSIS WITH SERVERLESS TECHNOLOGY

FINAL DEGREE PROJECT

Directed by Pedro Antonio García López

Double Degree in Biotechnology and Computer Engineering

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2021

Resum.

Els avanços en les tecnologies de seqüenciació de nova generació han revolucionat l'estudi de la biologia molecular en permetre la seqüenciació de milions de seqüències genòmiques de manera massiva. Quantitats inimaginables de dades genòmiques requereixen un processament bioinformàtic exhaustiu per a la seva correcta interpretació, una necessitat a la qual la computació tradicional li costa fer front. Per això s'ha recorregut a les arquitectures *serverless*, que permeten el processament des d'un ordinador personal de volums de dades d'una altra manera inviables, allunyant del programador responsabilitats com la provisió i gestió de recursos, i basant-se en els principis de simplicitat, escalabilitat i de facturació pels recursos utilitzats. Motivats pel seu millor rendiment i menor cost, grups de recerca bioinformàtica han decidit migrar els seus experiments a aquesta arquitectura mitjançant frameworks d'anàlisi de dades serverless, com Lithops. No obstant això, malgrat tenir menys limitacions quant a l'emmagatzematge de dades amb aquestes arquitectures, aquests *frameworks* no han estat dissenyats per a treballar amb tota mena de dades. Les dades genòmiques solen emmagatzemar-se en fitxers comprimits Gzip de desenes de terabytes, per la qual cosa és necessari implementar una utilitat que sigui capaç de descomprimir porcions d'aquests grans fitxers 'on-the-fly' per a la seva anàlisi en funcions *serverless*. Gràcies al particionador i recuperador de dades per a fitxers comprimits Gzip implementat en aquest treball, els bioinformàtics podran dur a terme els seus experiments mitjançant el *framework* d'anàlisi de dades *serverless* Lithops de manera senzilla, gaudint d'una experiència de programació impulsada per les dades i no per la gestió dels recursos. Per a validar l'eficiència d'aquest sistema, s'ha implementat el cas d'ús genòmic 'SNP Variant Caller' de Cloudbutton amb resultats satisfactoris.

Resumen.

Los avances en las tecnologías de secuenciación de nueva generación han revolucionado el estudio de biología molecular al permitir la secuenciación de millones de secuencias genómicas de manera masiva. Cantidades inimaginables de datos genómicos requieren de un procesamiento bioinformático exhaustivo para su correcta interpretación, una necesidad a la que la computación tradicional le cuesta hacer frente. Por ello se ha recurrido a las arquitecturas *serverless*, que permiten el procesamiento desde un ordenador personal de volúmenes de datos de otra manera inviables, alejando del programador responsabilidades como la provisión y gestión de recursos, y basándose en los principios de simplicidad, escalabilidad y de facturación por los recursos utilizados. Motivados por su mejor rendimiento y menor coste, grupos de investigación bioinformática han decidido migrar sus experimentos a esta arquitectura mediante *frameworks* de análisis de datos *serverless*, como Lithops. Sin embargo, pese a tener menos limitaciones en cuanto al almacenamiento de datos con estas arquitecturas, estos *frameworks* no han sido diseñados para trabajar con todo tipo de datos. Los datos genómicos suelen almacenarse en ficheros comprimidos Gzip de decenas de terabytes, por lo que es necesario implementar una utilidad que sea capaz de descomprimir porciones de estos grandes ficheros 'on-the-fly' para su análisis en funciones *serverless*. Gracias al particionador y recuperador de datos para ficheros comprimidos Gzip implementado en este trabajo, los bioinformáticos podrán llevar a cabo sus experimentos mediante el *framework* de análisis de datos *serverless* Lithops de manera sencilla, disfrutando de una experiencia de programación impulsada por los datos y no por la gestión de los recursos. Para validar la eficiencia de este sistema, se ha implementado el caso de uso genómico 'SNP Variant Caller' de Cloudbutton con resultados satisfactorios.

Abstract.

The advances made in next-generation sequencing technologies have revolutionized the study of molecular biology by enabling the sequencing of millions of genomic sequences on a massive scale. Unimaginable amounts of genomic data require exhaustive bioinformatic processing for their correct interpretation, a need that traditional computing is struggling to cope with. Therefore, serverless architectures have been resorted, which allow the processing of otherwise unfeasible volumes of data from a personal computer, taking responsibilities such as resource provisioning and management away from the programmer, and based on the principles of simplicity, scalability, and billing only for the resources used. Motivated by its better performance and lower cost, bioinformatics research groups have decided to migrate their experiments to this architecture using serverless data analysis frameworks, such as Lithops. However, despite having fewer limitations in terms of data storage with these architectures, these frameworks have not been designed to work with all types of data. Genomic data is often stored in Gzip compressed files of tens of terabytes, so it is necessary to implement a utility able to decompress portions of these large files 'on-the-fly' for their analysis in serverless functions. Thanks to the data partitioner and retriever for Gzip-compressed files implemented in this study, bioinformaticians will be able to perform their experiments using the Lithops serverless data analysis framework in a simple way, enjoying a programming experience driven by data rather than by resource management. To validate the efficiency of this system, Cloudbutton's genomic use case 'SNP Variant Caller' has been implemented with satisfactory results.

Index

Figure index

1 Introduction

The James Hutton Interdisciplinary Scientific Research Institute and the Biomathematics & Statistics Scotland (BioSS) Research Institute aim to develop and apply quantitative methods to improve scientific knowledge in fields such as molecular and biochemical sciences by means of information and computational sciences. One of the main problems this scientific coalition is facing resides in the field of genomic statistics and bioinformatics, as the development of new molecular genetics technologies for the study of genomes and the relationship between genomics and biological functions generates huge quantities of data which natural and biological sciences have never had to face before. The awakening of data analytics and cloud-based computing technologies have opened a wide range of opportunities to all those task that required an easy access to rapidly increasing computer processing power and data storage capacity.

With the aim of creating an automate methodology for the analysis of genetic biological data, this research union has enrolled in the CloudButton project which belongs to the EU's Horizon 2020 Framework Programme (H2020) and whose goal is to create a Serverless Data Analytics Platform. Therefore, by integrating Lithops, a Python multi-cloud distributed computing framework created by Cloudbutton, to their biological data analytics, computing processing power and storage won't be a limitation any longer. By using Lithops and serverless computing for biological data analysis, bioinformaticians and data analysts will be able to enjoy great advantages inherent to serverless computing, such as greater scalability and flexibility, reduced execution times, as well as its reduced cost. It seems that Cloud Computing is the way scientists must follow when the size of datasets quickly grows along with the computing power needed, in fact a new directive from the United Kingdom government urges its scientists to move their experiments to a cloud-based research computing infrastructure in order to save costs and improve computing capabilities.

The advantages of migrating the analysis of big biological data to the cloud don't stop there, since this architecture provides a way to remove architecture responsibilities from the developer and its computing workload, including resource provisioning, scaling and maintenance, in addition to being able to enjoy an automated scaling and a fine-grained billing system based on the resources used known as 'pay-as-you-go'. The scientific community has witnessed a substantial growth in the number of users of this technology, recent studies have found that 40 percent of organizations have adopted serverless architectures for their services [1].

Serverless architecture is not perfect, it is an evolving technology that leaves some developers reluctant to adopt it. Indeed, not all programmers are willing to go all-in with the serverless technology, sometimes the designs and tools used in their programs are not ready to migrate to the cloud, therefore developers must learn how to use this technology, as well as the technology must adapt itself to the requirements of the users.

An example of an adaptation that must be carried out is the modification of Lithops so that this distributed computing framework is able to work with large compressed genomic files. Lithops has been designed to work with files of the text family, which can be portioned in a simple and direct fashion, on the other hand, genomic sequencing files are usually stored in Gzip-compressed files that can reach sizes up to tens of terabytes. If we want to implement programs that use this data in serverless functions we have to figure out a way to portion and decompress data on-the-fly, so that many threads of execution can execute in parallel a genomic pipeline that would otherwise need a powerful cluster to be executed, resulting in a much more expensive practice.

For these reasons, in this final degree project we are going to design and implement a compressed data partitioner for the execution of cloud-based programs through serverless functions. Thanks to the development of this partitioner, bioinformaticians who integrate both the James Hutton Research Institute and BioSS will be able to migrate their tasks to a serverless architecture and execute their biologics pipelines in a more cost-effective fashion. In the development of the partitioner, we will have to follow the trend set by Lithops, since this framework aims to provide the greatest possible simplicity making a programmer-friendly experience from its use, making the provision of resources and the data partitioning completely transparent to the user, offering then a 'Data-Driven' experience.

2 Biological & Computational background

2.1 Biological background

Next-generation sequencing (NGS) technologies have been improving rapidly and have become the work-horse technology for studying nucleic acids such as DNA or RNA. The speed as well as the decreasing cost of NGS are the main culprits of the fast accumulation of raw sequencing data (sequencing reads). The public archive Sequence Read Archive (SRA) hosted by the US National Center for Biotechnology Information (NCBI), holded in October of 2017 about 14 petabases (millions of billions of bases), and the trend that has been followed is to double its size every 10-20 months (Figure 1) [2].

While advances in NGS have increased opportunities for reuse of data and collaboration, they have also created new computational problems. Omics aims at the collective characterization and quantification of pools of biological molecules that translate into the structure, function, and dynamics of an organism or organisms. The main workflow of conversion of raw data to scientific useful results requires coordinated computation, storage, and data movement. A larger quantity of raw data to process implies the need of greater computing power to solve the various computational problems encountered such as read alignment, *de novo* assembly, variant calling and quantification. This colossal increase in the amount of experimental data, as well as specialized databases spread over the Internet, has made classical (non-distributed) computing outdated. The storage, pre-processing and analysis of this gargantuan quantity of data has become the main bottleneck of the analysis of biological data.

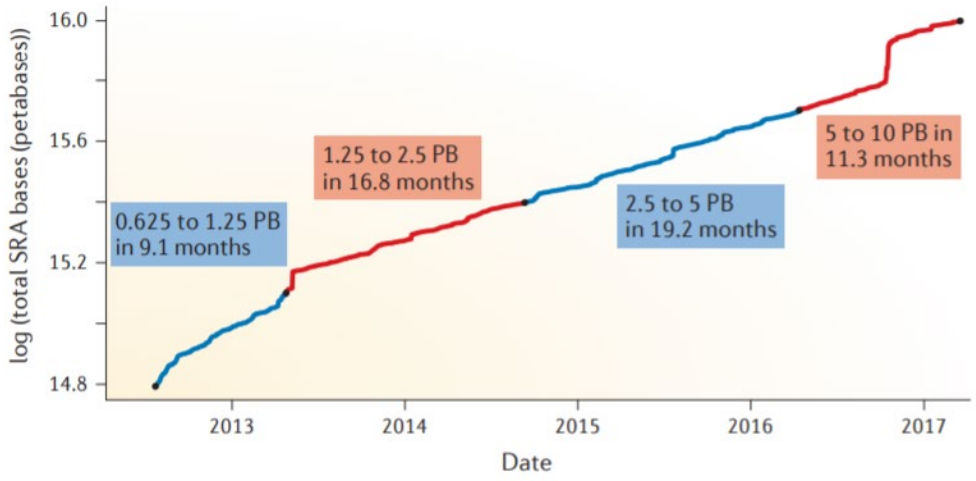

Figure 1. Increase of data generated by next-generation sequencing over the past few years (source:[2])

2.1.1 The real size of genomic data files

Next-generation sequencing technologies produce an enormous amount of genomic data and there is no real awareness of how large the files containing this type of information can get. If we had the perfect sequence of the human genome, without any type of technological flaws to worry about and therefore without including information of the quality or positioning of the sequence, the string of letters (A, T, C and G) that make up one strand of the reference human genome would occupy about 700 megabytes (3.2 billion base pairs) [3]. On a technical level, the reference human genome is a computational abstraction, in the real world the human genome is made up of 6.4 billion base pairs, but half of these are enough to represent the genome.

We can encode each base pair with 2 bits, using 00, 01, 10 and 11 for A, T, G and C respectively. If we multiply these 2 bits by the number of base pairs in the reference human genome, we will get a total of 6,000,000,000 bits or 750 megabytes. In fact, to sequence an entire genome, many reads (short sequences of about 100 base pairs) must be aligned with a reference genome. These reads are stored in specific formats such as FASTQ (Figure 2) in which for each read we have: a sequence identifier, the raw nucleotide sequence, a separator and finally the quality values of the nucleotide sequence [4]. This data format is not as simple as it would be using only 4 letters since all ASCII characters must be used. Genomes are normally sequenced with at least a 30x coverage, which means that on average each base of the genome is covered by 30 sequencing reads. By mapping a character to a byte and using an average depth of coverage of 30x, the resulting nucleic sequence in the FASTQ file would occupy around 180 gigabytes (ignoring labels, scores, control lines and carriage returns), that is, more than 200 times the size used in the aforementioned encoding.

Nowadays, genome sequencing data can be found on the servers of large sequencing companies that exploit technologies such as Illumina® or Roche454® to produce large genomic files. The computation of genomic programs with datasets of almost 200 gigabytes is unfeasible in the personal computers of scientists, not only because of the difficulties that it would entail storing more than one genome in a personal computer, but because its computing capacity is not adequate enough. For that reason, the execution of programs that use these datasets has been directed to parallel and distributed computing, where we do have the necessary resources for the execution of bioinformatics programs with big datasets [5].

Due to the large size of the FASTQ files resulting from the sequencing of a genome, these are usually compressed so that they take up less disk space. One of the most used compressions is Gzip, short for GNU zip, based on the deflation algorithm, a variation of the LZ77[1] file compression method, and on Huffman encoding [6]. By combining these two techniques, Gzip scans files for double character strings, so if repetitive sequences are found, it replaces them with a reference to the first string. Therefore, the resulting Gzip file, ending in .gz, is made up of strings and string references.

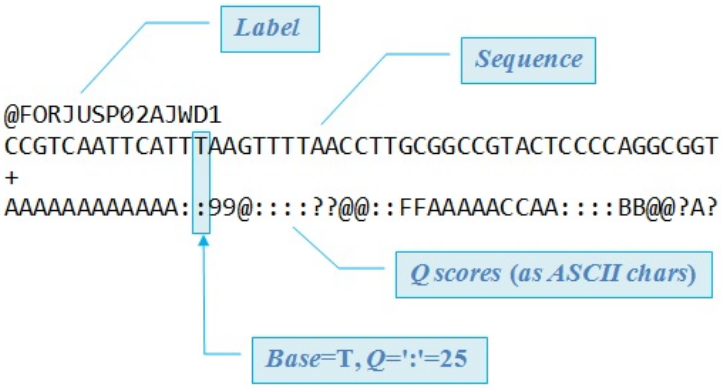

Figure 2. FASTQ format (source: [4])

[1] Lempel-Ziv 77

2.2 Computational background

2.2.1 Cloud computing and the serverless paradigm

"The cloud computing movement is motivated by the idea that data processing and storage can be done more efficiently on large farms of computing and storage systems accessible via the Internet" Dan C. Marinescu.

Computer clouds are the utilities providing computing services. In utility computing the hardware and the software resources are concentrated in large data centres. The users of computing services pay for the computing, storage, and communication resources they consume. Cloud computing is a computing paradigm, that required major changes in areas of computer science and engineering such as data storage, computer architecture, networking, and resource management.

In 2011, US NIST[2], defined cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction" [7]. Since its inception, cloud computing consisted of 5 main attributes: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. The era of Cloud computing can be considered to have arisen when in 2006 Amazon first offered EC2[3] (remote, configurable virtual machines) [8] and S3[4] (a low-cost, scalable cloud object) [9] as the first services provided by AWS[5]. Although it has been ingrained in the IT community for more than a decade, the promises made in UC Berkeley's report [10] have not been fully realized, therefore, users continue to have to deal with resource management tasks. Most users have difficulties configuring parameters in commercial cloud platforms, such as virtual machine instance types, number of instances or pricing model, and even the most experienced users are not able to take maximum advantage of the capabilities of the most common clouds platforms.

During the next 10 years after its emergence, Cloud computing has seen its advantages largely realized, and the number of its users has not stopped growing since then. But at the same time the number of its users grew, the number of its detractors did too, in fact the problems mentioned in [10] were not remedied and users continue to bear a burden from complex operations, and many workloads still do not benefit from efficient multiplexing. As a matter of fact, in Berkeley's report of 2019 [11] the problem was a stated as "Cloud computing relieved users of physical infrastructure management but left them with a proliferation of virtual resources to manage". The need for an alternative to the *IaaS*[6] was urgent, since an alternative in which the management of resources is transparent to the user would reach a greater number of developers and would let them exploit the real capabilities of Cloud computing. In response to this need, Serverless computing was born among cloud providers, this new paradigm offers advantages over traditional cloud-based or server-centric infrastructures, and the fact is that with serverless architectures developers do not need to

[2] National Institute of Standards and Technology

[3] Elastic Cloud Computing

[4] Simple Storage Service

[5] Amazon Web Services

[6] Infrastructure as a Service

worry about purchasing, provisioning, and managing backend servers. Serverless computing makes the user forget presumably about the management overhead and makes him solely responsible of the code implementation. As in a 'pay-as-you-go' phone plan, developers are only charged for the real usage, and not for the allocated resources as classical cloud services do. The code only runs when backend functions are needed by the serverless application, and the code automatically scales up as needed.

It was in 2015 when this computing paradigm came into being with the introduction of the first "general" serverless service AWS Lambda by Amazon [12], as a *FaaS*[7] suite. A stateless function is a function that has no knowledge about the external computation state, therefore, through the input parameters, it receives all the data necessary for its operation and returns all the outputs as return values. In the *FaaS* architecture, the source code is executed as stateless functions in the cloud, the tasks of management and allocation of resources are a responsibility of the cloud provider and a fine-grained pay-for-value billing model which only banks on the runtime memory selected by the user for the invoked executions and their aggregated execution time [13] is used. *FaaS* rapidly became the core of Serverless computing, although cloud providers also offer serverless *BaaS*[8] which are specialized serverless frameworks that cater to specific application requirements such as authentication or database access that cooperate with *FaaS*. From the paper "A Berkley View on Serverless Computing" [11] we can draw out two simple definitions, first Serverless computing can be defined as the sum of *FaaS* and *BaaS*, secondly a serverless service is the one that scales automatically with no need for explicit provisioning and can be billed based on usage.

The fact that computation is stateless and is provided by a different service from the remote storage system, consequently pricing for storage and computation becomes independent and can make them scale in the same way, adding that resource management is fully automated by the cloud provider and finally, that billing is based solely on resource usage and made *a posteriori*, are the three main features that detach Serverless computing from classic serverfull platforms. Serverless computing is built on top of a multi-tenant isolation based on VMs[9], the development of different approaches by the cloud providers to keep up with the rapid demand of resource provisioning gives the performance needed by stateless functions. The approach used by AWS Lambda [12] uses a pool of VM instances ready to receive a function and a pool of VM instances which have already hosted a function execution and are ready for later invocations, to solve the problem of resource provisioning previously mentioned.

2.2.2 Serverless computing in data analytics

FaaS lets developers write and update their codes on the fly, easy to scale, it represents a cost-efficient way to implement microservices. Thanks to their fine-grained elasticity and short start-up time, *FaaS* services have increased productivity of developers as well as it has shortened development time, reasons that justify its embracement by major corporations. To be able to face this growing demand, premier-class cloud providers offer their own *FaaS*

[7] Function as a Service

[8] Backend as a Service

[9] Virtual Machine

services, such as AWS Lambda [12], Azure Functions [14], Google Cloud Functions [15] or IBM Cloud Functions [16].

The operating dynamics of the different serverless platforms follow the same operating pattern. The developer codes a certain function in a high-level language and chooses the trigger that will prompt the execution of the function he has implemented, and as we have already mentioned before, it is the platform which is in charge of managing the resources needed for the execution, such as choosing the server instances where the function will run or ensuring the security and fault tolerance of the execution. In addition, users can include their own libraries, increasing the utility and versatility of *FaaS* compared to other architectures. Simplicity, flexibility, isolation, autoscaling and versatility make serverless platforms different from any other existing Cloud computing architectures, bringing resources "closer" to the programmer. In fact, not only do more experienced developers enjoy the benefits of forgetting about resource provisioning, but novice programmers can execute functions in the cloud ignoring the architecture that hides behind the execution.

The fields of engineering and science have welcomed this paradigm with great acceptance, and it seems that it has come to solve the problems encountered in big data analysis and processing where highly parallelizable tasks must be executed with variable data sizes and core number, fitting perfectly to the premises of Serverless computing. This led to the main cloud providers as well as third-party developers to implement data analytics platforms such as Apache Hadoop [17] or Apache Spark [18], a couple of open-source frameworks for computing large datasets in cluster computation which have been taken in by the academic and commercial scope. Even though the deployment of serverless functions is carried out in remote containers whose launch is significantly faster than the one of virtual machines, these platforms continue to have limitations related to the provisioning of hardware resources. For that reason, a few serverless data analytics parallel frameworks over stateless functions have been developed, two examples of these are PyWren [19, 20] based on AWS services and its extension Lithops [21, 22] that uses IBM Cloud, which allow us to execute code in parallel without explicit resource management and are based on the execution of jobs conforming to the MapReduce model [20, 21]. MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. The MapReduce program (Figure 3) is composed of a *map* procedure, which performs filtering and sorting (each worker node applies the *map* function to the local data and writes the output to a temporary storage, a master node ensures that only one copy of the redundant input data is processed), a *shuffle* stage, an all-to-all communication step between the map and reduce steps (worker nodes redistribute data based on the output keys, such that all data belonging to one key is located on the same worker node), and finally a *reduce* method, which performs a summary operation (worker nodes process each group of output data, per key, in parallel). The MapReduce infrastructure orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

Although alternatives to the MapReduce model have been implemented, PyWren and Lithops have both been implemented in Python and share the same future-based interface with primitives such as `map`, `wait`, `get_result` or `call_async`. In addition to being an extension of PyWren, Lithops improves the original implementation by adding a basic primitive `map_reduce`, automatic data partitioning, nested parallelism and customizable runtimes using Docker [23]. Unlike AWS Lambda, which is a proprietary platform, IBM Cloud Functions uses Apache OpenWhisk [24] which is an open source serverless cloud

platform, making Lithops a more flexible framework that allows a greater number of optimizations.

The transparency and elasticity required for decoupling the computation and storage end of an application by Lithops, is ensured by the stateless nature of the serverless functions. However, the model is dependent on an external storage system, being this way limited by its bandwidth and access throughput, and therefore by the persistence and communication between functions.

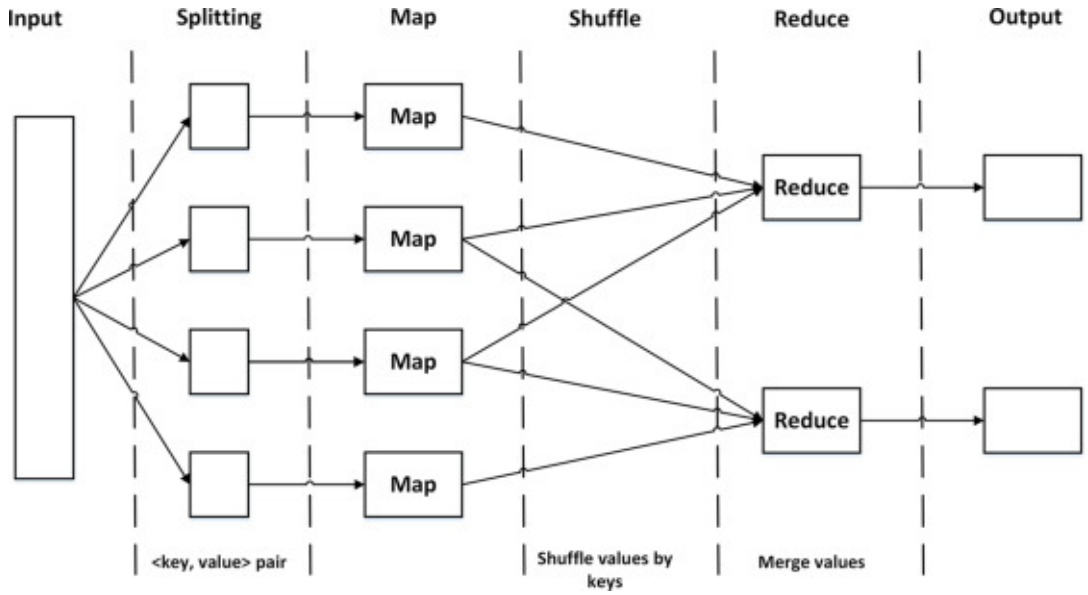

Figure 3. Scheme of the MapReduce model (source: [21])

In a nutshell, what Lithops intends is to bring serverless services closer to the programmer through the execution of unmodified local python code at a massive scale in the main serverless computing platforms. The developer's code is delivered into the Cloud without him knowing how it is deployed and run. One of its main goals is to provide the greatest simplicity and transparency, making both the development of the user's code and its execution a data-driven experience, in which the programmer can abstract himself from the provisioning and management of resources and focus on its code implementation and the data to be processed.

2.3 Serverless Computing to rescue genomics data analysis

Managing omics data usually stored in geographically distributed biological databases requires both space for data storing and services for data pre-processing, analysis and sharing. The resulting scenario comprises a set of bioinformatics tools, often implemented as web services [25].

Cloud computing and serverless architectures come to solve the obstacles found in the different stages of the bioinformatics analysis pipeline, from data management and processing to data integration and analysis, including data exploration and because it offers massive scalable computing and storage, data sharing, on-demand anytime and anywhere access to resources and applications.

With this change we would avoid executing the genomic data analysis in a cluster which results in an expensive practice, in favour of executing it through serverless functions, a much more cost-effective solution. The swap implies changes in the design of the bioinformatic pipeline, since when executed in a cluster a pre-processing step is required where the compressed files are unzipped to be read. On the other hand, when executed through serverless functions this pre-processing step can be skipped in favour of an 'on-the-fly' management. But, as mentioned in the introduction, Lithops is not set up to work directly with compressed genomic data or any kind of compressed files, so both Lithops and bioinformatics implementations must be adapted so that they can work together in unison.

2.3.1 The need for a genomic file specific partitioner for Lithops

As mentioned in section 2.1, the big data era of computational genetics presents new challenges for the field of bioinformatics. Novel computational approaches are being developed to handle the large, complex, and noisy datasets produced by high throughput technologies [26]. Lithops has built-in logic for processing data objects from public URLs and object storage services such as IBM Cloud Object Storage [27].

The built-in data-processing logic of Lithops integrates a data partitioner system that allows to automatically split the dataset (usually a csv or a text file) in smaller chunks. Splitting a file into smaller chunks permit to leverage and favour the parallelism provided by the compute backends. The built-in partitioner is configurable by specifying the size of the chunk or the number of chunks to split each file, and it has been designed within the `map()` and `map_reduce()` API calls. Although it is expected that in future releases more data types will be supported by Lithops, the current implementation can only partition files of the text file family, that is, which contain multiple lines or columns ending with '\ n' such as .txt files or .csv files among others. Normally when we use Lithops for data processing, data is not stored locally but stored in a Cloud Object Storage service [28]. Whether the input is a bucket, a list of buckets or a list of data objects, if the chunk size is not configured, each chunk will correspond to an entire object and therefore a function activation will be executed for each object. On the other hand, if a chunk size is configured, the partitioner will be activated within Lithops, and a function activation will be executed for each chunk generated.

Lithops' integrated partitioner is prepared to work with files that can be accessed randomly, such as text files or csv. On the other hand, compressed files such as gzip files cannot be accessed randomly without prior processing, which is why in this final degree project we will implement a specific partitioner for Gzip files commonly used for storing genomic data, and thus being able to enjoy the advantages of serverless computing in the analysis of genomic data.

2.4 Contributions

The main contributions of this final degree project are the following:

- Implementation of a gzip-compressed file partitioner and data retriever, including random extraction of chunks from the original compressed file.
- Modification and adaption of a genomic pipeline for the migration of its execution to serverless functions.
- Integration of the partitioner of compressed files in Lithops, to be able to enjoy the advantages of serverless computing in the analysis and processing of genomic data, and the implementation of data-driven analytics over compressed files.

The development of this final degree project is essential for the correct migration of the genomics use cases of the Hutton Institute and BioSS to the serverless technology. This migration is necessary since genomic datasets do no stop growing in size and complexity, and the serverless technology appears to be the most scalable alternative, besides being the winning horse in terms of cost and performance. With the advances in simplicity and modularity that this study will provide, bioinformaticians will be able to carry out biological data-driven analytics using Lithops in a simple fashion, with great advantages both in terms of computing performance and economic cost.

3 Project motivation and proposal

The development of this final degree project has been motivated by the joint work of the URV [10] with the James Hutton Institute and BioSS [11], in a collaboration framework for the CloudButton's Horizon 2020 Framework Program (H2020). The main objective of the CloudButton project is to create a Serverless Data Analytics Platform, democratizing *big data* by overly simplifying the overall life cycle and cloud programming models of data analytics thanks to serverless technologies. The interest of this project is of such importance that it has already been applied in different use cases for two fields of science: *bioinformatics* (genomics and metabolomics data) and *geospatial* (LiDAR and satellital data).

BioSS undertakes research, consultancy and training in mathematics and statistics as applied to agriculture, the environment, food, and health. In their *Statistical Genomics and Bioinformatics* department, they aim to develop and automate methodology for analysing genomic data, harnessing the computing power to extract maximum information from the data. This organization requested the collaboration of the URV which is a major contributor for the CloudButton project and primarily authored Lithops' paper [21] for the analysis and processing of genomic data through Lithops.

This study is integrated into the genomics use-case, and its main objective is the implementation of a library for the partitioning of Gzip-type compressed files, to facilitate the analysis and processing of large genomic datasets that are usually stored under this type of compression, using the serverless technology and more precisely the Python multi-cloud distributed computing framework Lithops. Currently, there is a built-in partitioner in Lithops, but it only works for files of the text file family. Csv (Comma-separated values) files, txt files or json (JavaScript Object Notation) files belong to this family of text files and what differentiates them is the separator between fields, while their common denominator is that they consist only of ASCII characters and that their *line feed* is made up of a '/ n'. A new partitioner is then needed in order to work with compressed files.

In this study we present the implementation of a data partitioner integrated in Lithops for Gzip-compressed files and its use in real genomic use case.

[10] Universidad Rovira i Virgili

[11] Biomathematics and Statistics Scotland

4 System overview and architecture

This section presents the different technologies (services, software, and tools) used for the development of this final degree project, as well as a brief description of the intended functionality of the designed system.

4.1 Lithops integrated partitioner

Lithops' built-in logic integrates a data portioner system that automatically splits datasets in smaller chunks in order to exploit the benefits of parallel and distributed computing. This data partitioner can be configured by specifying the size of the chunk or the number of chunks in which the file has to be partitioned, but has only been developed to work with files from the family of text files (.txt, .csv, json, etc.).

4.2 IBM COS

IBM Cloud Object Storage (COS) is a highly scalable cloud storage service, designed for high durability, resiliency, and security. IBM COS can be described as a high latency, high bandwidth, persistent and cheap cloud storage system. This remote storage service lets you store, manage and access unstructured data via a self-service portal and RESTful[12] APIs. The metadata pertaining to the serverless functions in Lithops is stored in IBM COS. Furthermore, in our design IBM COS is also used for storing large input files for the functions as well as for the communication between them.

4.3 IBM Cloud Functions

IBM Cloud Functions is a *FaaS* platform on IBM Cloud, built upon the Apache OpenWhisk open-source project, which allows code to be executed in response to events. At the time this report was written, a standard IBM Cloud lite account limited serverless functions to 1200 concurrent invocations, 2048MB of runtime memory, and 600 seconds of execution time.

4.4 Gztool

Gztool [29] is a GZIP files indexer, compressor, and data retriever, which creates small indexes for gzipped files and uses them for quick and random-positioned data extraction with no penalty. By default, Gzip-compressed files have not been designed to be accessed in a random way, that is, to know the value of a byte at a given position x, it is necessary to decompress the file from the beginning to the x byte. However, Mark Adler, author of zlib [30] provided the scientific community with 'zran.c' [31] a cryptic file that creates an index of "windows" filled with 32 kiB of uncompressed data at different positions along the compressed file. The index can be used to initialize the zlib library and make it behave as if the compressed file begins there, being in this way possible to extract chunks of the file and decompress them.

4.5 Lithops interface

Lithops' framework provides a series of primitives for launching MapReduce schemed workloads with serverless functions by means of IBM Cloud Functions, using IBM COS for input, output and intermediate data, along with function metadata. The client provided by

[12] Representational State Transfer

Lithops can work both in the cloud and locally on the client computer, allowing the invocation of serverless functions in IBM Cloud Functions remotely and storing data in IBM COS.

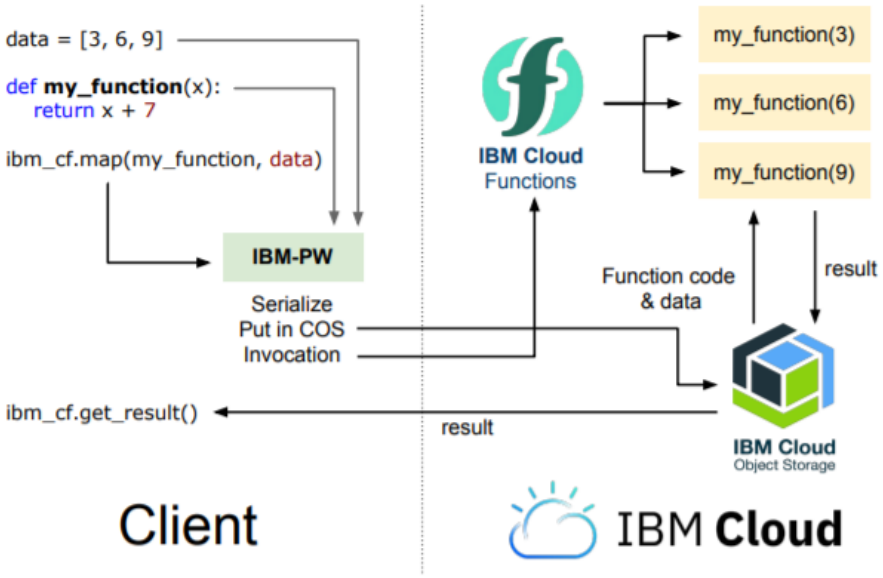

Figure 4. High level diagram of Lithops execution workflow (source: [21])

Figure 4 presents a diagram of the execution workflow in Lithops. In a general and summarized way: 1) The client takes the user's code as well as the input data, serializes it and stores it in the IBM COS storage service; 2) Then, it invokes the function through IBM Cloud Functions; 3) After that, on the server side, the IBM Cloud Functions framework gets the user's code and the input data from IBM COS and executes the function in parallel (one for each piece of data), storing the results back to IBM COS; 4) Finally, the results are pulled by the client straightway when they are ready [21].

4.6 Description of our system

We have implemented a new specific Gzip-compressed files partitioner integrated in Lithops. Among other functions, its main feature is that the partitioner can be called within the serverless function when a previously pre-processing stage has been done, retrieving a selected chunk from the original compressed file, and obtaining a text file corresponding to the chunk which will be used within the map function.

5 System design

In this section we will address at first the reasons that that have guided us to establish the design, to afterwards explain the actual design of our Gzip-compressed files partitioner and its integration in a Lithops' `map_reduce()` implementation.

5.1 Design motivation

5.1.1 Basis of Gzip files

Gzip is a file format and a software application used for file compression and decompression, intended for use by GNU. The decompression of the Gzip format can be implemented as a streaming algorithm, which is an important feature for Web protocols and data interchange. The file format is based on the DEFLATE algorithm, a combination of LZ77 and Huffman coding [6], and it is composed by:

- A 10-byte header which contains a magic number[13], the compression method, 1-byte of header flags, a 4-byte timestamp, compression flags and the operating system ID
- Optional extra headers as allowed by the header flags
- A body, composed of the DEFLATE-compressed payload
- An 8-byte footer, containing a CRC-32 checksum and the length of the original uncompressed data, modulo 2^{32}.

Genomic data produced by Next-Generation Sequencing technologies tends to be highly redundant and repetitive from an information processing perspective, as it is information intended for human consumption. Natural language is highly inefficient and as it has evolved it has added redundancy to itself.

For that reason, in the early 1960s computer scientists Abraham Lempel and Jacob Ziv published "A Universal Algorithm for Sequential Data Compression" [33] also known as LZ77 compression algorithm, which mechanism is pretty straightforward, for each sequence of bytes encountered it searches backwards to check if the same sequence has already occurred, then instead of outputting the sequence the algorithm makes a reference to the first occurrence of the sequence (Figure 5). The "<#,#>" brackets indicate backpointers in the form of distance and length, so the backpointer "<A,B>" indicates to search backwards a characters and reproduce the B characters found there, in order to decompress the document. All these references end up defeating the purpose of compressing, since the compressed file can get even longer than the original. For that reason, GZIP compression includes techniques out of our scope such as "variable length codes" and Huffman encoding in order to reach a greater compression. In a nutshell, GZIP's DEFLATE algorithm works as following, first the document is compressed with LZ77 compression algorithm, then LZ77's output is compressed with Huffman encoding generating multiple Huffman tables as output, and finally those Huffman code tables are compressed using more Huffman codes generating a Huffman code table unique to the document.

[13] A constant numerical or text value used to identify a file format or protocol

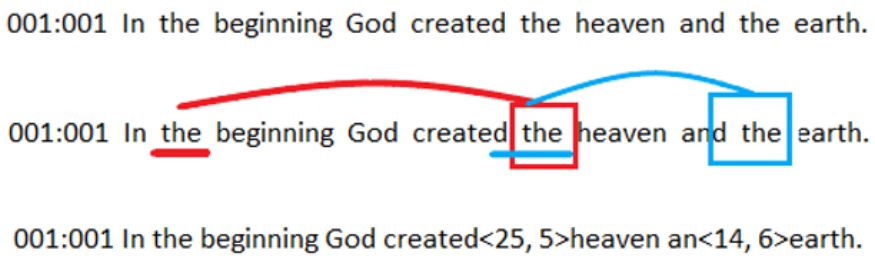

Figure 5. Conceptual example of LZ77 compression algorithm

5.1.2 Basis of Gztool

Gztool is a GZIP files indexer, compressor, and data retriever [29]. This tool creates small indexes for gzipped files and uses them for quick and random data extraction with no penalty. Gzip-compressed files cannot be accessed in a random way, meaning that in order to extract a byte at a given position N, it is needed to decompress the complete file from the beginning to the N byte.

Mark Adler, co-writer of zlib [33], a software library used for data compression which acts as an abstraction of the DEFLATE compression algorithm used in the GZIP file compression program, provided a cryptic file named "zran.c" [31] which creates an index of "windows" filled with 32 kiB of uncompressed data at different positions along the compressed file. In essence, it creates "points of access" to the compressed stream of data and those points can be used to initialize the zlib library. The index size of a gzip-compressed file is about 0.33% of the original Gzip file size. Therefore, gztool which is built upon "zran.c" is an utility that provides a set of command line tools to compress, decompress, create an index and retrieve randomly located data from a gzipped file. In addition to the aforementioned functionalities, gztool implements optimizations and new behaviours making the use of these indices more convenient. Among these new behaviours we can highlight that gztool can read incomplete Gzip-concatenated-files, store line and byte numbering information in the index and retrieve data from a specific line or byte. Moreover, this tool has its own inherent advantages such as the fact that data extraction and index creation are interleaved (no overhead for index creation), the index files are reusable, the span between the index points can be configured, and the fact that windows are not loaded in memory unless they are needed, making the memory footprint of the tool fairly low.

This application can have different dynamics of use and workflows, but in the case of exporting its utilities to a serverless function, the most interesting workflow would be generating at first an index for the Gzip-compressed file and then perform queries over chunks of the original file by means of the index file (Figure 6).

5.1.3 Basis of bioinformatics alignment programs

The availability of genome-wide epigenomic datasets enables in-depth studies of genetic variabilities and modifications. In bioinformatics, an alignment of sequences is a way of arranging DNA, RNA or protein sequences, with the objective of identifying regions possessing similarities which can be a consequence of functional, structural or evolutionary relationships between the sequences. In 1970 Needleman and Wunsch developed the first dynamic programming algorithm for pairwise sequence alignment, but despite this major

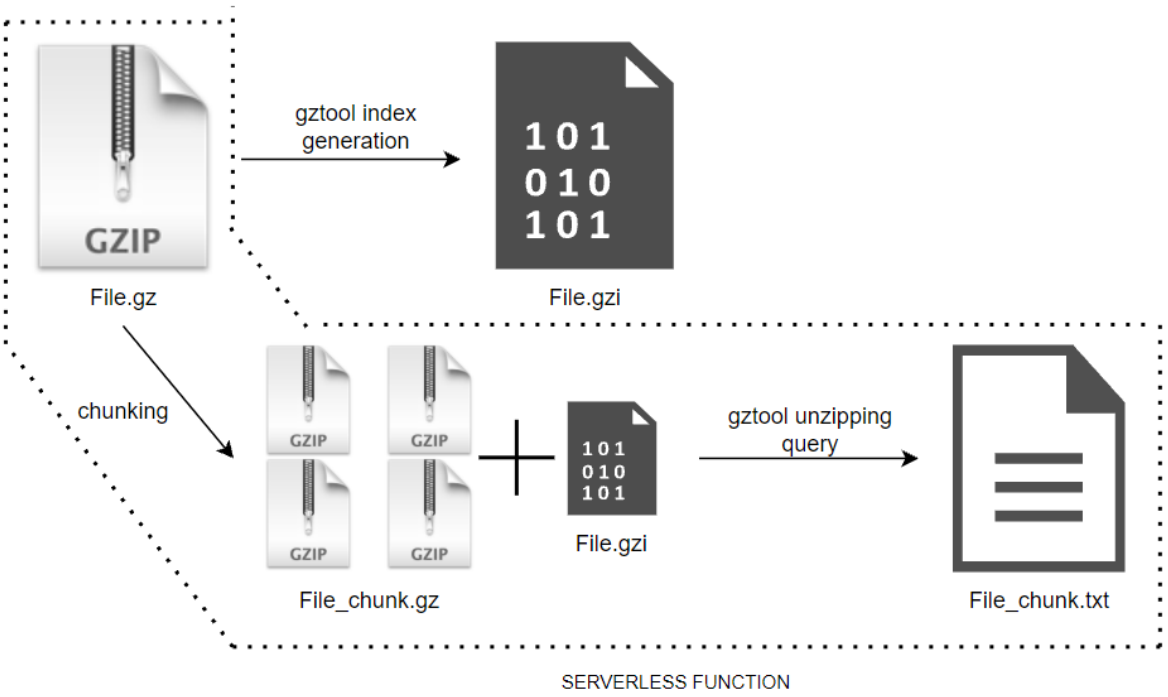

Figure 6. Gztool workflow scheme

advance, it was not until the 1980s when the first multiple sequence alignment algorithm emerged. Since then, alignment algorithms have found themselves to be the spearhead of computational biology and bioinformatics, and their evolution has made them characterized by performing highly parallelizable tasks.

The partitioner that has been implemented in this study is intended to be used for Variant Calling. Variant calling entails identifying single nucleotide polymorphisms (SNPs) and small insertions and deletions (indels) from next generation sequencing data. Briefly, it is the process by which we identify variants from sequence data, and it is composed of 3 steps:

- Whole genome or whole exome sequencing, creating FASTQ files
- Alignment of the sequences to a reference genome, creating BAM or CRAM files
- Identification of where the aligned reads differ from the reference genome, creating a VCF file

The Variant Calling process is usually carried out with bioinformatic tools and applications designed for use in UNIX environments. In the Cloudbutton Genomics use case, the Variant Caller demo designed for use with the gzipped files partitioner uses GEM3-Mapper and Samtools, which are a high-performance mapping tool for aligning sequenced reads against large reference genomes, and a set of utilities that manipulate alignments in the SAM (Sequence Alignment/Map), BAM, and CRAM formats respectively. Both programs are designed to work on streams of data and by means of UNIX pipes.

5.2 Design of the Gzip-compressed files partitioner for Lithops

Lithops is a Python multi-cloud distributed framework, which allows you to run unmodified local python code at massive scale in serverless computing platforms. As our main objective is the implementation of a black-boxed library for Gzip-compressed files partitioning with maximum transparency from the user interface, the implementation will use the default compute and storage backends of Lithops, being these, IBM Cloud Functions for serverless function invocation and IBM COS for input, intermediate and output storage. Nevertheless, the functions included in this library can also be executed locally instead of in serverless a serverless function, since some of them are meant to be executed in the programmer's computer. Therefore, this library has been implemented in Python programming language with the help of Bash programming language, thanks to the `subprocess` Python module.

As mentioned above, due to the design of the tools used in this partitioner of compressed files, to carry out the decompression of randomly positioned data, a pre-processing stage is necessary in which the index of the compressed file is generated as well as data that will be later used for the partial decompression of the original file. For these reasons, this library includes different methods for the correct partitioning of compressed files as well as for the use of its functionalities in serverless functions. The implemented methods are the following: `preprocess_gzfile()`, `download_index_files()`, `chunk_complete_gzfile()`, `preprocess_chunk_complete_gzfile()`, `retrieve_random_chunk_gzfile()`, `create_iterdata_from_info_files()`.

The design of the system and how to use it in a serverless function will be explained through the different methods. But in short, the partitioner has been divided into two main parts, the preprocessing stage, and the extraction and decompression stage.

5.2.1 Preprocessing step

As mentioned in previous sections, in order to decompress chunks of a compressed Gzip file, a pre-processing of the entire compressed file must be carried out beforehand. The initial complete index creation for an already Gzip-compressed file still consumes as much time as a complete file decompression and needs access to the complete file, but once created it can be reused as many times as we want for the same file. For these reasons, the system has been designed so that the index is intended to be created in a virtual machine where the original compressed file is stored or downloaded, generating an index file whose size is approximately 0.33% of the original file size as well as some complementary index files, which are then stored back in an IBM COS bucket (Figure 8).

For this reason, in this work a primitive `preprocess_gzfile()` has been designed, which takes a gz-compressed file and generates its index as well as complementary files to the index that will be used in the partitioning and decompression of the original file. In Figure 7 we can find its header.

```
    def preprocess_gzfile(bucket:BUCKET_NAME,file.gz:FILE_NAME):
// Downloads file.gz from BUCKET to TMP if not found in the VM
// Uploads index files to the BUCKET (.gzi, .gzi.info, .gzi_tab.info)
```

Figure 7. Header of our `preprocess_gzfile` primitive

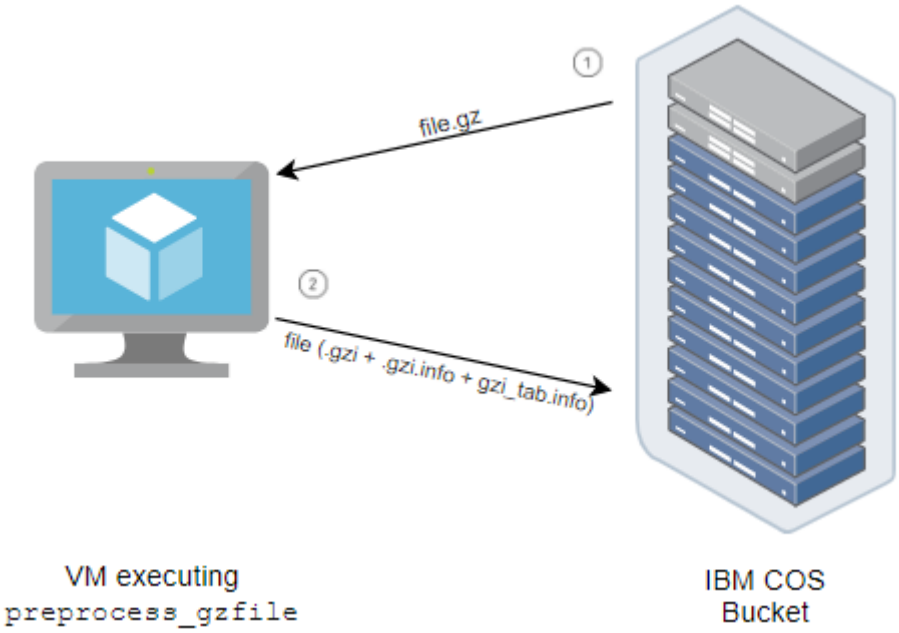

Figure 8. `preprocess_gzfile` primitive intended workflow

As the index generated by the gztool is a binary file (file.gzi) that will be used in later stages of the partitioner but cannot be interpreted or analyzed by humans, the index file is translated into ASCII characters using the gztool utility "-e" (don't stop the process on first error), plus "-ll" (show data about each index point) over the original index file, generating "file.gzi.info" a text file from which we will retrieve useful information for the partitioner (Figure 9). From the information found in this file, the partitioner uses the total number of lines that make up the original file and creates its own index "file.gzi_tab.info" for the original file that associates a compressed byte with a line number (@compressed_byte, L# line_number).

```
Checking index file 'file.gzi' ...
      Size of index file (v1)  : XX.XX kiB (XXXXX Bytes) (X.XX%/gzip)
      Guessed gzip file name   : 'file.gz' (XX.XX%) X.XX MiB (XXXXXXX Bytes)
      Number of index points   : XX
      Size of uncompressed file: XX.XX MiB (XXXXXXXX Bytes)
      Number of lines          : XXXXXX (XXX.XX k)
      Compression factor       : XX.XX%
      List of points:
      #: @ compressed/uncompressed byte L#line_number (window data size in Bytes
@window's beginning at index file), ...
#1: @ 10 / 0 L1 ( 0 @60 ), #2: @ 103511 / 1353675 L12920 ( 2903 @92 ),
```

Figure 9. Structure and information found in "file.gzi.info"

5.2.2 Chunking and decompressing step

Once the original file has been pre-processed, it is only necessary to have access to the index and its complementary files, as well as to a certain portion of the compressed file to be able to access the information the portion houses. From here, two main strategies for partitioning the original compressed file emerge, and the choice between them will depend on the user's needs and how it has implemented his code. The first strategy, which we could name the "total partitioning strategy" is based on the division of the Gzip-compressed file into parts with an equal number of lines, so that when these chunks are unzipped, we obtain files containing the same number of lines (except for the last one if the division is uneven), leading us to obtain all the chunks that make up the original un-zipped file. The second strategy that we could call the "random partitioning strategy" is based on the decompression of a chunk that has been delimited by the user, this would be the strategy to choose if we want to implement a solution based on serverless functions.

5.2.2.1 Total partitioning strategy

The "total partitioning strategy" assumes that a single machine will perform the complete partitioning of the compressed file. In Figure 10, we can observe the header of the `chunk_complete_gzfile` primitive which has been implemented to carry out this task. First, it will obtain from the bucket the index files of the compressed file, and by means of the variable `LINES` that indicates the number of lines that the generated un-zipped chunks must contain and the file "file.gzi_tab.info" that associates a compressed-byte to the start of a certain line, it will obtain the range of bytes to be obtained from the original Gzip file. For the complete chunking an interval list is generated in which the information about the blocks (`start_line`, `end_line`, `start_byte` and `end_byte`) is stored, and then the list is looped for the download, decompression and upload of the chunks. Finally, we meet a state in which we have the entire file unzipped in the bucket but partitioned into smaller and more manageable chunks that can be downloaded independently and processed in parallel, thus reducing the execution time of a program working with all the data. The diagram showing the working dynamics underlying this strategy is shown in Figure 11.

```
    def chunk_complete_gzfile(bucket:BUCKET_NAME,file.gz:FILE_NAME,
                                  lines_number:LINES):
// Downloads chunks of file.gz from BUCKET to TMP
// Uploads un-zipped chunks to the BUCKET
```

Figure 10. Header of our `chunk_complete_gzfile` primitive

The `preprocess_chunk_complete_gzfile` primitive brings together the functions performed by both the `preprocess_gzfile` and `chunk_compelte_gzfile` primitives.

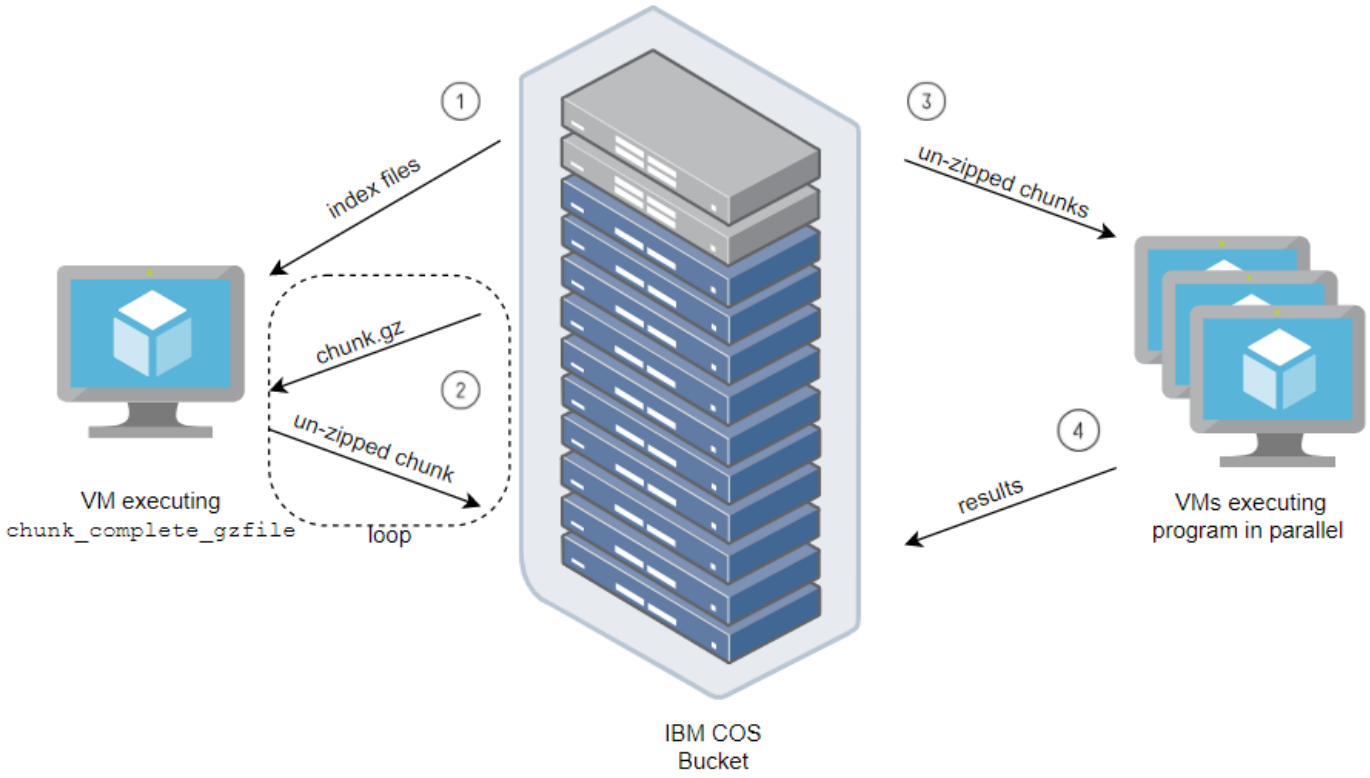

Figure 11. `chunk_complete_gzfile` primitive intended workflow

5.2.2.2 Random partitioning strategy

The "random partitioning strategy" has been designed in order to work over serverless functions and is based on the `retrieve_random_chunk_gzfile` primitive whose header can be observed in Figure 12. As in the strategy previously explained in section 4.4.2.1, this one is also based in portioning the Gzip-compressed file into portions delimited by a range of lines provided by the programmer, but instead of partitioning the entire file, only the on demand specified range of lines is partitioned and decompressed. Like in the previous strategy, partitioning requires access to the index files, although it also provides advantages, since the programmer can make each thread of execution deal with the decompressing of its specific portion of data, thus removing the inconvenience and overhead of portioning and decompressing the complete Gzip-compressed file initially, in addition to avoiding the extra costs of communication and download of the files from the bucket.

```
def retrieve_random_chunk_gzfile (bucket:BUCKET_NAME,file.gz:FILE_NAME,
                                  start_line:START_LINE,end_line:END_LINE)
// Downloads chunk of file.gz from BUCKET to TMP
// Uploads un-zipped chunk to the BUCKET
```

Figure 12. Header of our `retrieve_random_chunk_gzfile` primitive

The working dynamics of this strategy can be observed in Figure 13, nonetheless this alternative does not fully suit Lithops' framework intended workflow for the creation of serverless functions. Therefore, in the next section we will approach the design of the integration of this partitioner with a Cloudbutton's use case.

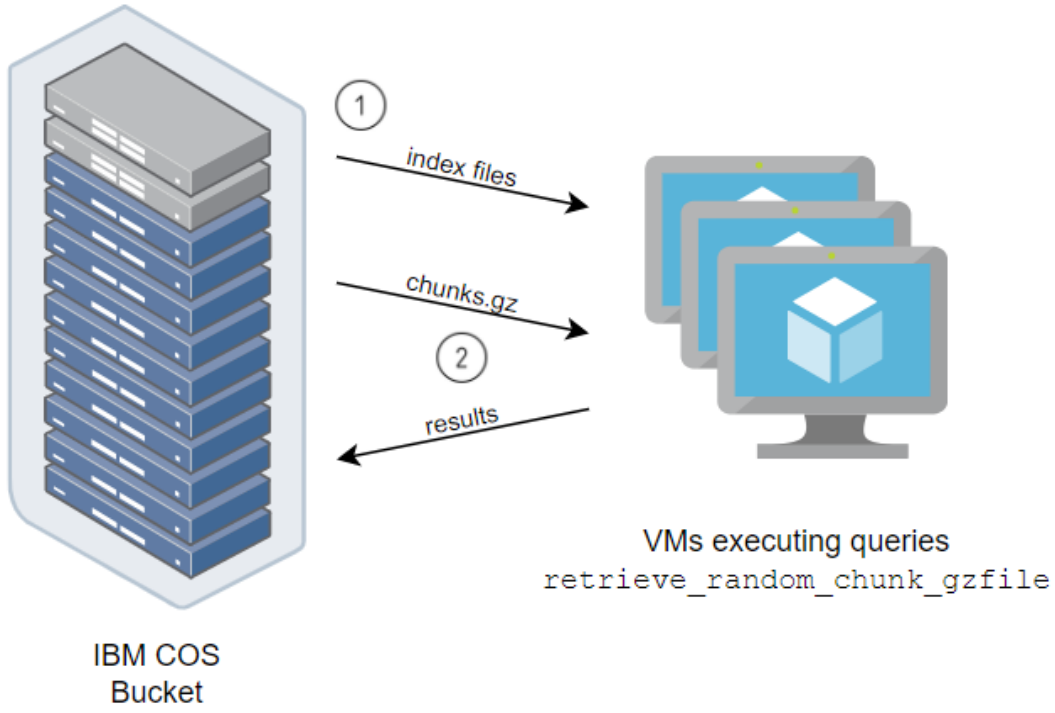

Figure 13. `retrieve_random_chunk_gzfile` primitive intended workflow

5.2.3 Integration with a Cloudbutton's genomics use case

One of the genomic use cases of Cloudbutton is the variant calling of single nucleotide polymorphisms. In this section the goal is to implement the design of a distributed Variant Caller program for the analysis of a sequenced genome in FASTQ format against smaller sequences in FASTA format with the aim of finding those polymorphisms. For this, all the FASTA format sequences must be confronted with all the chunks that make up the original Gzip-compressed file, using the bioinformatic tools "gem-mapper" and "samtools". Since Lithops' framework is intended for the launch of MapReduce schemed workloads with serverless functions, we have to design our distributed Variant Caller meeting certain specifications.

First and foremost, the aforementioned pre-processing step 4.4.1 continues to be mandatory for the partitioning since all the mappers will have to access the index files. Secondly, a tailored `iterdata` with access to the index files must be used. This customized `iterdata` will create the partitions of the compressed file accordingly to the number of lines indicated by the programmer, thus creating a list of coordinates (bytes and lines) that the mappers will use to decompress the chunks that are assigned to them, in addition to

performing a shuffling task to create all the possible combination between FASTQ chunks and FASTA sequences found in the bucket. The header of the primitive `create_iterdata_from_info_files` is presented in Figure 14. Finally, a set of map and reduce functions are created according to the iterdata list, each mapper task is in charge of downloading, decompressing and processing the chunk that has been assigned to it generating a temporary file as a result. The map function uses here a slightly modified version of `retrieve_random_chunk_gzfile` (section 4.4.2.2) which doesn't store back in the bucket the unzipped chunk. The reduce function gathers the results produced by the different mappers, process them, and generates a final text file with the results (Figure 15). The `map_reduce` scheme is executed in containers fitted with a custom runtime.

This strategy differs from the "total partitioning strategy" by partitioning and decompressing the chunk from the compressed Gzip-file "on the fly". This dynamic has therefore three main advantages:

- Since the download and decompression of the chunks are done "on the fly" (within the map function), it becomes a task than can be performed in parallel and not sequentially, shortening the execution time.
- Since the download and decompression of the chunks are done "on the fly", we save the time and communication costs associated with performing these tasks in advance.
- Since the download and decompression of the chunks are done "on the fly" and the resulting unzipped file is not further used in other processes, it is not stored back in the bucket, shortening the execution time and optimizing the use of the bucket's memory (differing from the "random partitioning strategy").

```
def create_iterdata_from_info_files (bucket: BUCKET_NAME, file.gz: FILE_NAME,
                                     fasta_prefix: FASTA_FILE_PREFIX, file.gz:
                                     FASTAQ_FILE_NAME, lines_number: LINES)
// Generates iterdata shuffling chunks from file.gz with FASTA files
```

Figure 14. Header of `create_iterdata_from_info_file` primitive

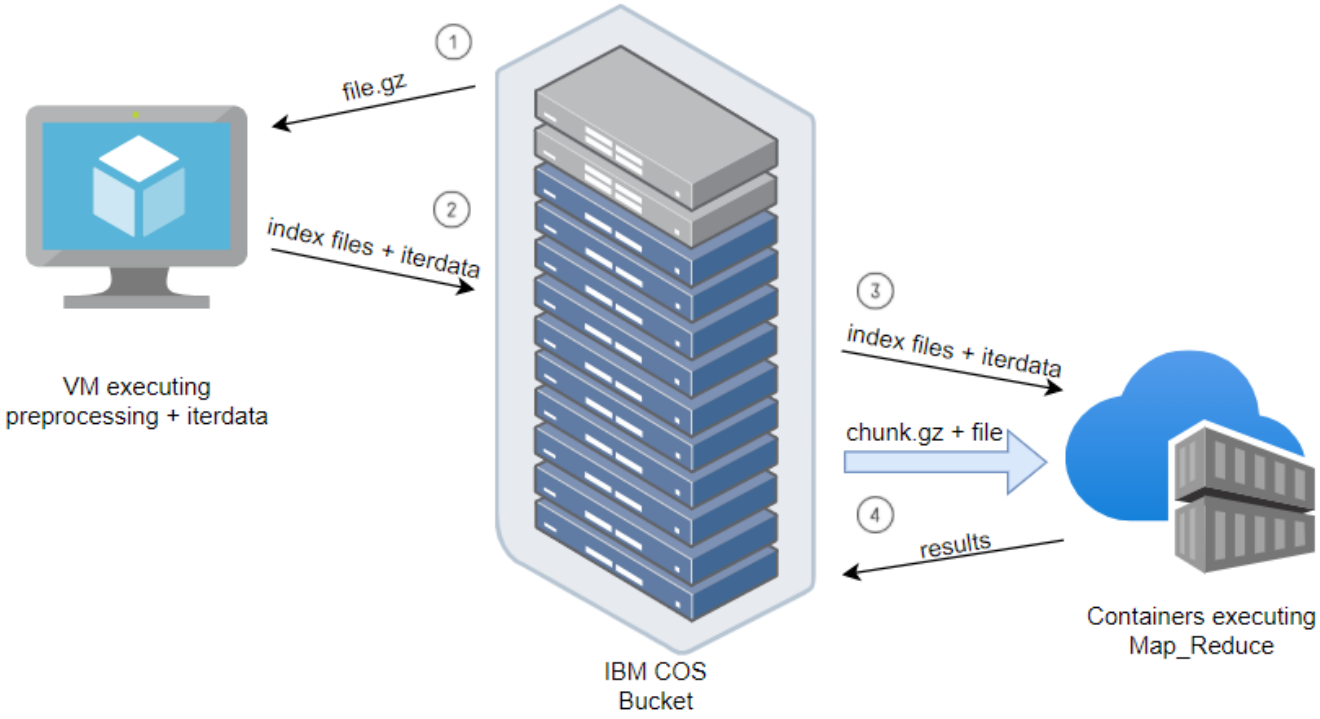

Figure 15. Cloudbutton's genomic use case workflow

As it has been commented in this section, the design of our partitioner functionality can be divided into two main parts. In a first step, a pre-processing of the Gzip compressed file is carried out to generate the index files, this being a synchronous process for which the next step must wait. The second step, which once the pre-processing is done can be executed asynchronously and in parallel, is based on the download and decompression of a portion of the compressed file, in addition when used in Lithops, this functionality is ready to be performed 'on-the-fly' within a serverless function.

6 Implementation

In this section we will address how the previously explained design has been implemented.

6.1 Pre-processing step

In the "pre-processing" step, we launch a synchronous sole function in the Python Virtual Machine, which performs a series of calls to the Gztool via the Python subprocess module over the complete original Gzip-compressed file. Essentially, this step will create the index of the Gzip-compressed file, as well as two auxiliary index files used by our functions (file.gzi.info and file.gzi_tab.info), that will be uploaded to the IBM COS bucket. From now on, every function of the partitioner or serverless function of the Cloudbutton's genomic use case will access those files through IBM COS.

The code for the pre-processing step is presented in Figure 16, firstly the complete Gzip-compressed file is downloaded as a temporary file, secondly a subprocess call is performed over the compressed file, and finally the resulting index files are uploaded to the bucket and the total number of lines of the bucket is returned. The subprocesses correspond to calls to bash scripts that are responsible for the use of the Gztool utility and can be found in the Annex C.

6.2 Chunking and decompressing step

6.2.1 Total partitioning

The total partitioning in which we chunk and unzip the original compressed file entirely based on the number of lines indicated by the programmer, consists of a synchronous sole function executed in the Python Virtual Machine. In this function, the different pieces that make up the original file are downloaded from the IBM COS bucket as temporary files and are then unzipped consecutively with the help of the index files previously generated, to be then stored in the bucket. After getting the index files, the function generates an interval list with the information of the blocks to be chunked and unzipped, and then iterates through the list downloading, unzipping and uploading the chunks, returning the number of unzipped chunks generated.

The code for the total partitioning is presented in Figure 17, and as the other functions the subprocesses in it correspond to calls to Gztool via the Python subprocess module.

6.2.2 Random partitioning

The random partitioning in which we chunk and unzip a portion of the original compressed file based on a range of lines indicated by the programmer, consists of an asynchronous sole function that can be executed in parallel either in the Python Virtual Machine or in a serverless function. In this function a portion of the gzip-compressed file is downloaded as a temporary file from the IBM COS bucket and is then unzipped with the help of the index files previously generated, to be then stored back in the bucket. After getting the index files, the function converts the line interval to a byte range of the compressed file, downloads the specific range of bytes as a temporary file, unzips it and uploads the uncompressed chunk to the bucket.

The code for the random partitioning is presented in Figure 18, and as the other functions the subprocesses in it correspond to calls to Gztool via the Python subprocess module.

```
def preprocess_gzfile(bucket_name, file_name):
    """
    The function takes the gzip file, creates the necessary index files
    for partitioning and stores them in the bucket.
    """
    os.chdir(LOCAL_TMP)
    storage = Storage()
    # 0 DOWNLOAD FASTQ.GZ FILE TO LOCAL TMP
    local_filename = os.path.join(LOCAL_TMP, file_name)
    if not os.path.isfile(local_filename):
        print(f'Downloading cos://{bucket_name}/{file_name} to {LOCAL_TMP}')
        obj_stream = storage.get_object(bucket_name, file_name, stream=True)
        with open(local_filename, 'wb') as fl:
            shutil.copyfileobj(obj_stream, fl)

    # 1 GENERATING THE INDEX AND INFORMATION FILES AND UPLOADING TO THE BUCKET
    sp.run(CURRENT_PATH+'/generateIndexInfo.sh '+local_filename, shell=True, check=True,
universal_newlines=True)
    output = sp.getoutput(CURRENT_PATH+'/generateIndexInfo.sh '+local_filename)
    output = output.split()
    total_lines = str(af.only_numerics(output[-3]))

    # 2. UPLOAD FILES TO THE BUCKET
    remote_filename = INDEXES_PREIX+file_name

    print(f'Uploading {local_filename}i to cos://{bucket_name}/{INDEXES_PREIX}')
    with open(f'{local_filename}i', 'rb') as fl:
        storage.put_object(bucket_name, f'{remote_filename}i', fl)

    print(f'Uploading {local_filename}i.info to cos://{bucket_name}/{INDEXES_PREIX}')
    with open(f'{local_filename}i.info', 'rb') as fl:
        storage.put_object(bucket_name, f'{remote_filename}i.info', fl)

    print(f'Uploading {local_filename}i_tab.info to cos://{bucket_name}/{INDEXES_PREIX}')
    with open(f'{local_filename}i_tab.info', 'rb') as fl:
        storage.put_object(bucket_name, f'{remote_filename}i_tab.info', fl)

    os.remove(f'{local_filename}i')
    os.remove(f'{local_filename}i.info')
    os.remove(f'{local_filename}i_tab.info')

    os.chdir(CWD)

    return total_lines
```

Figure 16. Code of the pre-processing stage (`preprocess_gzfile` function)

```
def chunk_complete_gzfile(bucket_name, file_name, lines):
    """
    This function creates the partitions of x 'LINES' of the compressed file,
    unzips them and stores them in the bucket.
    """
    storage = Storage()

    # 1 DOWNLOAD INDEX FILES
    download_index_files(bucket_name, file_name, storage)

    # 2 GENERATING LINE INTERVAL LIST AND GETTING CHUNK'S BYTE RANGES
    print('\n--> Generating chunks')
    os.chdir(LOCAL_TMP)
    total_lines = af.get_total_lines(file_name)
    block_length = str(lines)
    sp.run(CURRENT_PATH+'/generateChunks.sh '+file_name+' '+block_length+' '+total_lines,
           shell=True, check=True, universal_newlines=True)
    chunks, chunk_counter = af.read_chunks_info(file_name)

    print(chunks)

    # 3 RETRIEVE CHUNKS FROM BUCKET AND UNZIP THEM
    for chunk in chunks:
        print("\n--> Processing chunk {}... ".format(chunk['number']))

        byte_range = f"{int(chunk['start_byte'])-1}-{int(chunk['end_byte'])}"
        obj_stream = storage.get_object(bucket_name, file_name, extra_get_args={'Range':
f'bytes={byte_range}'}, stream=True)

        local_chunk_filename = os.path.join(LOCAL_TMP, file_name.replace('.fastq.gz',
f'_chunk{chunk["number"]}.fastq'))
        local_chunk_filename_gz = f"{local_chunk_filename}.gz"

        with open(local_chunk_filename_gz, 'wb') as fl:
            shutil.copyfileobj(obj_stream, fl)

        cmd = f'gztool -I {file_name}i -n {chunk["start_byte"]} -L {chunk["start_line"]}
{local_chunk_filename_gz} | head -{block_length} > {local_chunk_filename}'
        sp.run(cmd, shell=True, check=True, universal_newlines=True)

        print(f'Uploading {local_chunk_filename} to cos://{bucket_name}/{CHUNKS_PREIX}')
        with open(local_chunk_filename, 'rb') as fl:
            remote_chunk_filename = CHUNKS_PREIX+file_name.replace('.fastq.gz',
f'_chunk{chunk["number"]}.fastq')
            storage.put_object(bucket_name, remote_chunk_filename, fl)

        os.remove(local_chunk_filename)
        os.remove(local_chunk_filename_gz)

    print(str(chunk_counter)+" chunks decompressed.")

    os.chdir(CWD)
```

Figure 17. Code of the chunking and decompressing step (total partitioning / `chunk_complete_gzfile` function)

```
def retrieve_random_chunk_gzfile(bucket_name, file_name, start_line, end_line):
    """
    The function retrieves a random chunk defined by 'start_line' and 'end_line'
    of the gzip file stored in the bucket, unzips it and stores it back in the bucket.
    """
    # 1 DOWNLOAD INDEX FILES
    download_index_files(bucket_name, file_name)

    # 1 GETTING CHUNK BYTE RANGE
    os.chdir(LOCAL_TMP)
    sp.run(CURRENT_PATH+'/randomChunkRange.sh '+file_name+' '+start_line+' '+end_line,
           shell=True, check=True, universal_newlines=True)
    chunk = af.read_chunk_info_random(file_name, start_line, end_line)

    # 2 RETRIEVE CHUNK FROM BUCKET AND UNZIP IT
    remote_file_name = file_name.replace(LOCAL_TMP+'/', '')
    storage = Storage()
    byte_range = f"{int(chunk['start_byte'])-1}-{int(chunk['end_byte'])}"
    obj_stream = storage.get_object(bucket_name, remote_file_name, extra_get_args={'Range':
f'bytes={byte_range}'}, stream=True)

    local_chunk_filename = os.path.join(LOCAL_TMP, file_name.replace('.fastq.gz',
f'_chunk{chunk["number"]}.fastq'))
    local_chunk_filename_gz = f"{local_chunk_filename}.gz"

    with open(local_chunk_filename_gz, 'wb') as fl:
        shutil.copyfileobj(obj_stream, fl)

    block_length = str(int(end_line) - int(start_line) + 1)
    cmd = f'gztool -I {file_name}i -n {chunk["start_byte"]} -L {chunk["start_line"]}
{local_chunk_filename_gz} | head -{block_length} > {local_chunk_filename}'
    sp.run(cmd, shell=True, check=True, universal_newlines=True)

    print(f'Uploading {local_chunk_filename} to cos://{bucket_name}/{CHUNKS_PREIX}')
    with open(local_chunk_filename, 'rb') as fl:
        remote_chunk_filename = CHUNKS_PREIX+file_name.replace('.fastq.gz',
f'_chunk{chunk["number"]}.fastq')
        storage.put_object(bucket_name, remote_chunk_filename, fl)

    os.remove(local_chunk_filename)
    os.remove(local_chunk_filename_gz)

    print("Chunk decompressed")

    os.chdir(CWD)
```

Figure 18. Code of the chuncking and decompressing step (random partitioning / `retrieve_random_chunk_gzfile` function)

6.3 Integration with a Cloudbutton's genomics use case

Our Gzip-compressed files partitioner has been integrated to a Cloudbutton's genomic use case by developing a program that implements genomic and bioinformatic tasks that are executed in parallel using the Map_Reduce scheme thanks to the lithops computing framework. The program aims to carry out a SNP Variant Calling on the results of a massive sequencing (Fastq file) comparing it to consensus genomic sequences (Fasta file) with the objective of showing the singularities that characterize the sequenced genome. For this, the following main program performing the task previously explained and presented in Figure 19, has been implemented.

```
if __name__ == "__main__":

    # Preliminary steps:
    # 1. upload the fastq.gz into BUCKET_NAME
    # 2. Upload the fasta chunks into BUCKET_NAME

    # Create index files (only once)
    #lithopsgenetics.preprocess_gzfile(BUCKET_NAME, fastq_file)  # Uncomment only one time

    # Generate iterdata
    iterdata = lithopsgenetics.create_iterdata_from_info_files(BUCKET_NAME,
fasta_chunks_prefix, fastq_file, 100000)

    fexec =
lithops.FunctionExecutor(runtime='lumimar/ibm_gem3_runtime:0.4',runtime_memory=1024,log_lev
el='DEBUG')
    fexec.map_reduce(my_map_function, iterdata, my_reduce_function)
    result = fexec.get_result()
```

Figure 19. Main's genomic use case program implementation

To prepare the program so that it can be executed, first some preliminary steps have to be carried out, such as uploading the entire compressed file to the bucket, as well as the fragments of the consensus sequence with which it is going to be compared. Furthermore, if it has not been done previously, the compressed file must be pre-processed in order to obtain its index files. These preliminary and pre-processing tasks only have to be carried out the first time the program is executed, since the content of its actions will not change unless the compressed file is changed, and the files they generate can be reused.

At the end of this first phase, we must have the following file folder and file structure in our bucket (Figure 20).

6.3.1 Custom Iterdata

As we are working with a bioinformatics program that has specific requirements, a custom iterdata must be passed to the Lithops' `map_reduce()` function. The objective of this iterdata function is to shuffle all the fastq file chunks with all the fasta files, thus generating all the possible combinations. In addition to carrying the keys of the files to be used by every map task, it also carries the byte ranges extracted from the index files to be downloaded from the compressed Gzip file in order to access the specific compressed file chunks.

Figure 20. Necessary files and folder structure in the bucket

```
def create_iterdata_from_info_files(bucket_name, fasta_file_prefix, fastq_file_name,
lines):
    """
    Creates the lithops iterdata from the fasta chunks and the fastq index files
    """
    storage = Storage()
    # 1 DOWNLOAD FASTQ INDEX FILES
    download_index_files(bucket_name, fastq_file_name, storage)

    os.chdir(LOCAL_TMP)
    total_lines = af.get_total_lines(fastq_file_name)
    total_lines = str(int(total_lines) + 1)

    list_fasta = storage.list_keys(bucket_name, prefix=fasta_file_prefix)

    # 2 GENERATING LINE INTERVAL LIST AND GETTING CHUNK'S BYTE RANGES
    print('\n--> Generating chunks')
    block_length = str(lines)
    sp.run(CURRENT_PATH+'/generateChunks.sh '+fastq_file_name+' '+block_length+'
'+total_lines,
           shell=True, check=True, universal_newlines=True)
    chunks, chunk_counter = af.read_chunks_info(fastq_file_name)

    list_fastq = []
    for chunk in chunks:
        list_fastq.append((fastq_file_name, chunk))

    iterdata = []
    for fasta_key in list_fasta:
        for fastq_key in list_fastq:
            iterdata.append({'fasta_chunk': fasta_key, 'fastq_chunk': fastq_key})

    os.chdir(CWD)

    return iterdata
```

Figure 21. Customized iterdata function for the genomic Variant Caller use case

The specific iterdata function (`create_iterdata_from_info_files` presented in Figure 21), first gets from the bucket the index files previously generated in the main program for the Gzip-compressed file containing the Fastq file, as well as the names of the Fasta files. Secondly, with the number of lines specified by the programmer, it generates the line intervals that will make up the chunks of the compressed file with their respective byte ranges. And finally, it creates a list in which all the chunks of the Fastq compressed file are shuffled with all the Fasta files, thus obtaining all the possible combinations. The elements of the iterdata list have the following structure:

```
{'fasta_chunk': 'A.fasta',
'fastq_chunk': ('B.fastq.gz', {'number': C, 'start_line': 'D',
'end_line': 'E', 'start_byte': 'F', 'end_byte': 'G'})
}
```

Figure 22. Custom iterdata format

6.3.2 Custom Runtime

A run-time system (or just runtime) is a software designed to support the execution of computer programs written in some computer language, or in other words, the execution environment. Lithops' runtimes are based on Docker, which is an open-source containerization platform, enabling developers to package applications and their dependencies into containers which are then run on a Docker Engine. To run our Variant Caller program, a custom runtime had to be used, adding to the Lithops' default runtime the bioinformatic tools 'Samtools' and 'gem-mapper', as well as Gztool for the unzipping of the compressed file.

6.3.3 Map function

To execute the genomic variant calling pipeline, a map function (Figure 22) that unifies the producer-consumer design has been implemented. Therefore, in the same thread of execution, both the reading of the data and its processing is carried out. For this reason, we can divide the map function into two large parts, the data download, and the data processing respectively.

In the first phase, corresponding to the data download to the function's runtime, the map function brings the necessary data to the execution of the bioinformatic pipeline. All the necessary data is stored in the IBM COS bucket, including the generic files that are used by all the mappers (genomics/indexes/ and genomics/references/), as well as the compressed FASTQ file and the FASTA files. The function starts by copying the FASTA file indicated by the iterdata. Then copies the specific byte-range of the compressed FASTQ file also indicated by the iterdata, as well as its specific index file (.gzi), to then make a call to the Gztool utility by means of Python's subprocess system. And finally, the reference files are downloaded.

Once all the necessary data is in the function's runtime, the second phase of the mapper begins by making a call to the gem-mapper (FASTQ to sam transformation) tool followed by 3 calls to the samtools' utilities (sam to bam transformation, sorting bam, generating mpileup), sequentially. Finally returning the output of the last call (mpileup) as the function's output.

```
def my_map_function(fasta_chunk, fastq_chunk, storage):
    """
    gem3 mapper to mpileup output
    """
    # copying fasta chunk to runtime
    temp_fasta = copy_to_runtime(storage, BUCKET_NAME, '', fasta_chunk)

    # copying fastq chunk to runtime
    fastq_file_key = fastq_chunk[0]
    fastq_chunk_data = fastq_chunk[1]
    byte_range = f"{int(fastq_chunk_data['start_byte'])-1}-
{int(fastq_chunk_data['end_byte'])}"
    temp_fastq_gz = copy_to_runtime(storage, BUCKET_NAME, '', fastq_file_key, byte_range)

    # getting index and decompressing fastq chunk
    temp_fastq = temp_fastq_gz.replace('.fastq.gz',
f'_chunk{fastq_chunk_data["number"]}.fastq')
    temp_fastq_i = copy_to_runtime(storage, BUCKET_NAME, idx_folder, f'{fastq_file_key}i')
    block_length = str(int(fastq_chunk_data['end_line']) - int(fastq_chunk_data['start_line'])
+ 1)
    cmd = f'gztool -I {temp_fastq_i} -n {fastq_chunk_data["start_byte"]} -L
{fastq_chunk_data["start_line"]} {temp_fastq_gz} | head -{block_length} > {temp_fastq}'
    sp.run(cmd, shell=True, check=True, universal_newlines=True)

    # copying reference genomes from cloud storage to runtime
    temp_gem_ref = copy_to_runtime(storage, BUCKET_NAME, ref_folder, gem_genome)
    temp_fa_ref = copy_to_runtime(storage, BUCKET_NAME, ref_folder, fa_genome)
    temp_fai_ref = copy_to_runtime(storage, BUCKET_NAME, ref_folder, fai_genome)

    # verify all files are in /tmp
    print(os.listdir("/tmp"))
    print('Going to process fastq chunk:', temp_fastq)
    print(fastq_chunk_data)

    # temporary intermediate file names
    sam_name = os.path.splitext(temp_fastq)[0]+'.se.sam'
    bam_name = os.path.splitext(sam_name)[0]+'.bam'
    bam_sorted_name = os.path.splitext(bam_name)[0]+'.sorted.bam'

    # 1. fastq to sam (gem3-mapper)
    sp.call(['gem-mapper', '-I', temp_gem_ref, '-i', temp_fastq], stdout=open(sam_name, 'w'))
    # 2. sam to bam (samtools)
    sp.call(['samtools', 'view', '-bS', sam_name], stdout=open(bam_name, 'w'))
    # 3. sort bam (samtools)
    sp.call(['samtools', 'sort', bam_name], stdout=open(bam_sorted_name, 'w'))
    # 4. generate mpileup (samtools)
    mpileup_out = sp.check_output(['samtools', 'mpileup', '-A', '-B', '-Q', '0', '-f',
temp_fa_ref, bam_sorted_name])

    print(mpileup_out)

    return mpileup out
```

Figure 23. Cloudbutton's Variant Caller map function implementation

6.3.4 Reduce function

The reduce function is a simple function that joins and orders the outputs of the different mappers in the same text file by means of a bash script that makes a call a bioinformatic utility called SiNPle. For this, all the mappers' outputs are concatenated into the same string, and then the string is passed into the script.

```
def my_reduce_function(results, storage):
    """
    Mpileup merge and SNP calling with SiNPle
    """
    lineout = []
    for line in results:
        line = line.decode('UTF-8')
        lineout.append(line)
    output = "".join(lineout)

    temp_mpileup = '/tmp/file.mpileup'
    with open(temp_mpileup, 'w') as f:
        f.write(output)
    sinple_out = sp.check_output(['bash', '/bin/mpileup_merge_reduce.sh', temp_mpileup,
'/bin/'])
    sinple_out = sinple_out.decode('UTF-8')

    storage.put_object(BUCKET_NAME, out_folder+'test2.txt', body=sinple_out)
```

Figure 24. Cloudbutton's Variant Caller reduce function implementation

7 Evaluation

To evaluate our system, different tests have been carried out and their results have been analysed. For each test, its meaning and relevance have been highlighted. We evaluated our Gzip-compressed files partitioner utility in the *eu-gb* region (London), with 2048 MB of runtime memory per function. Our dataset and end backend storage were located in IBM COS' buckets based in the same region.

7.1 File size comparison

This study and the subsequent development of a partitioner for Gzip-compressed files would not make sense if there was no real difference between the size of the compressed and decompressed file. In the following graph (Figure 25) we can observe the difference in size between files of different sizes in their zipped and unzipped versions, as well as the size of their corresponding index file. We can depict that on average the compression of a file through Gzip reduces its size to 30-40% of the original size and the index file occupies between 0.3% and 0.4% of the uncompressed file. This reduction in size justifies the development of this study together with the implementation of the Gzip-compressed files partitioner, since it represents a saving in storage and communication costs.

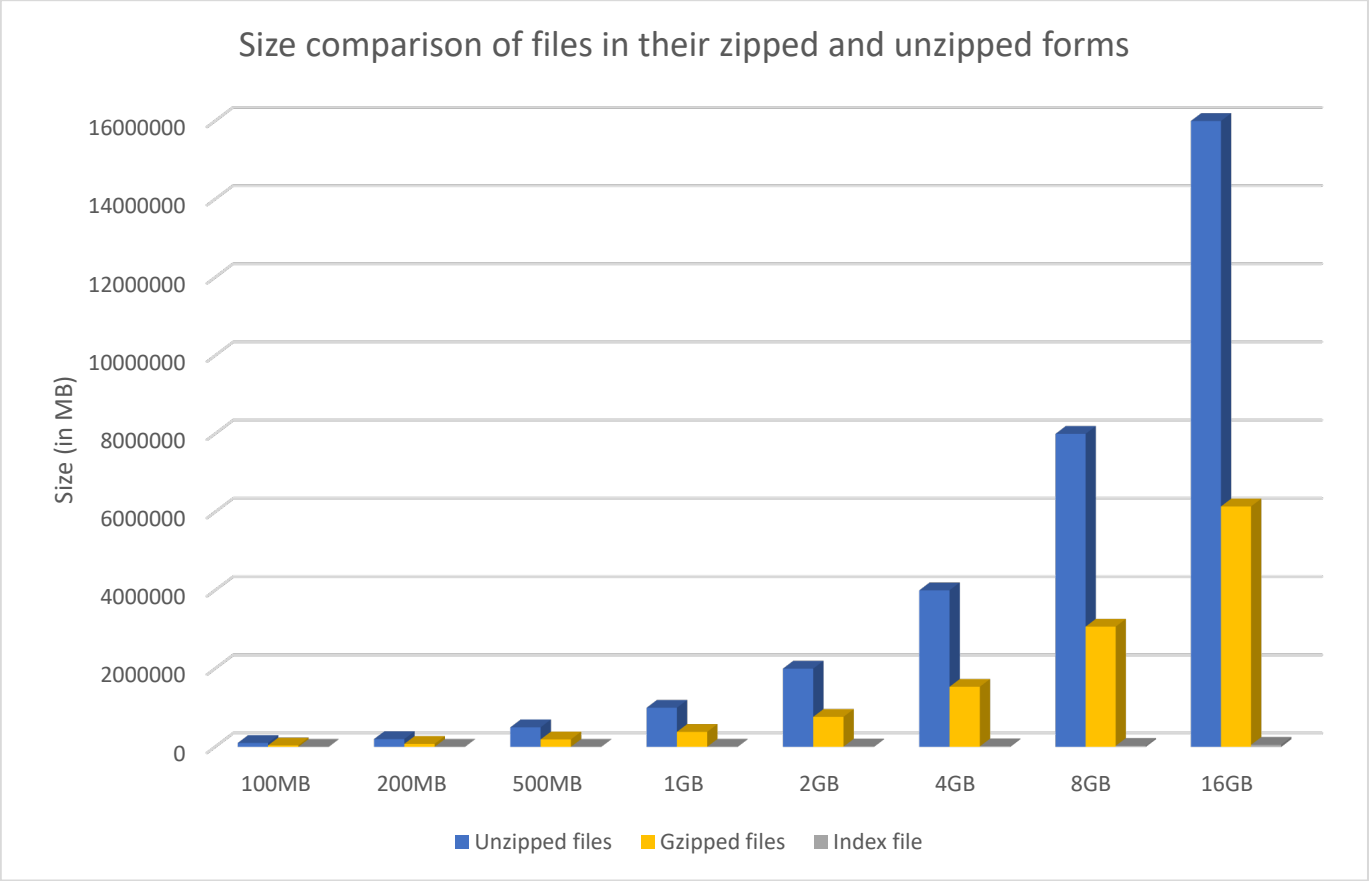

Figure 25. Size comparison of files in their zipped and unzipped forms

We can then conclude that there is a significant difference between the original size of a file, its compressed version size, and its index file size. In real life, the disk space occupied by a file, data transfer time, as well as the cost of said transfer are important factors that influence how resources at our disposal should be used. Working with compressed files results in less disk space occupancy, shorter transmission times and therefore a lower economic impact.

7.2 Pre-processing time evaluation

As explained in previous sections, the pre-processing stage is a mandatory stage prior to partitioning that must be done in a synchronous and non-distributed manner and intended to be executed in a virtual machine. Therefore, carrying out this task carries a time penalty, although once completed, the index files can be reused as many times as we want, pre-processing time costs won't be taken into account when analysing execution times. The comparison of pre-processing times of compressed large files is shown in Figure 26.

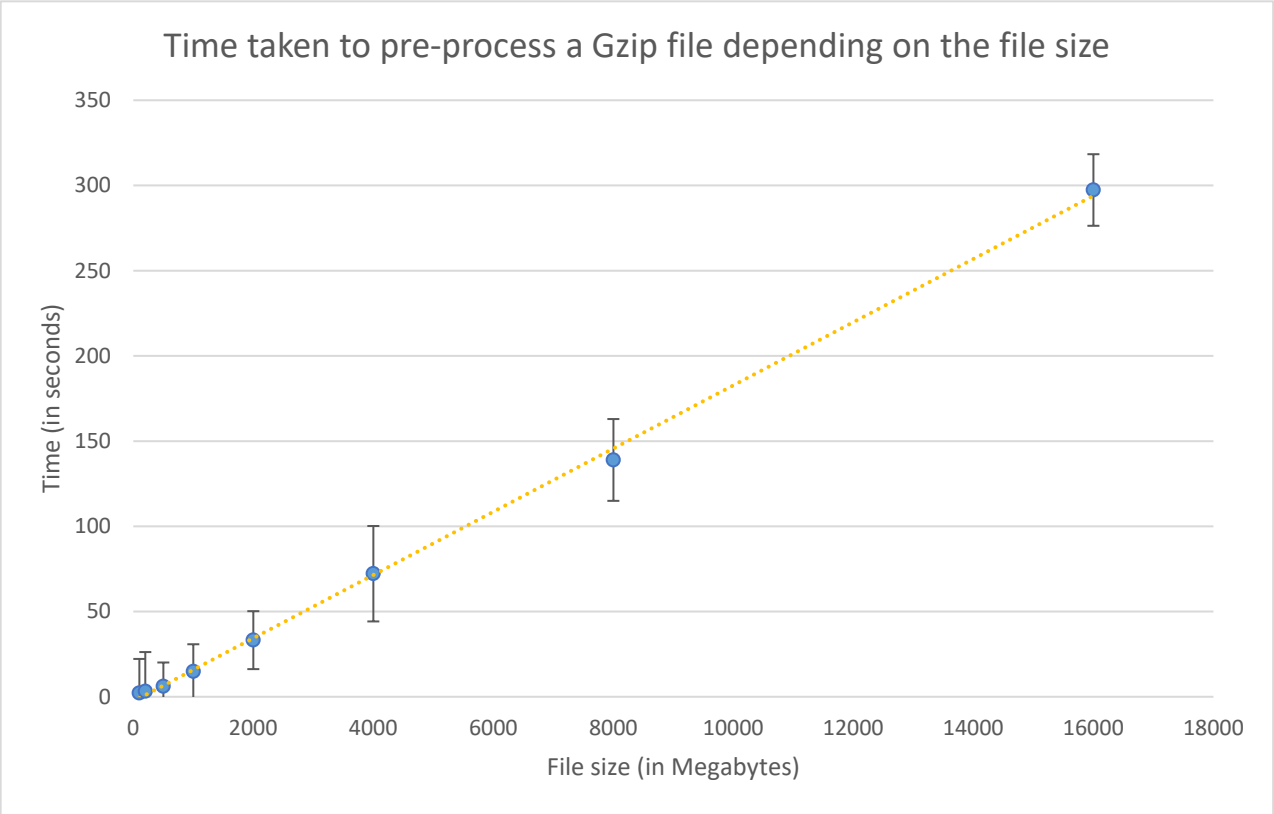

Figure 26. Time taken to pre-process a Gzip file depending on the file size

We can see that the pre-processing time of a Gzip-compressed file is proportional to its size and follows a linear trend. Therefore, pre-processing smaller files results in a much faster task than pre-processing larger files.

7.3 Data obtention time: parallel and sequential computing

In the bar chart of Figure 27, a comparison of the times required to obtain different portions of data is presented. Thanks to this comparison we can realize that within the same thread of execution it can sometimes be more interesting to use the partitioner or to not use it depending on the size on the number of lines to chunk. This comparison does not consider that the data chunk must be processed within a program, that there may exist a storage limitation in the bucket and that the internet connection was not homogenous during the experiment. By not taking into account all these factors, we can observe that it takes longer to download the entire compressed file and unzip it, than it does to download it directly unzipped.

But since this utility has been designed with the aim of performing parallel and distributed computing, Figure 29 shows us that when portioning an entire file into chunks of the same number of lines, working with serverless functions in parallel is much more advantageous and faster than working sequentially. Observing that there is a greater advantage when the number of chunks to be generated is greater.

Although increasing the number of workers by reducing the chunk size does not improve performance in a linear way, since an excessive number of mappers causes an overhead due to the management of all these execution threads. For this reason, for each file size, the optimal number of workers/chunk size must be inferred. In figure 28, the optimal chunk size is found between 250.000 and 500.000 lines in a 2.000.000 lines file (the experiment does not consider data processing).

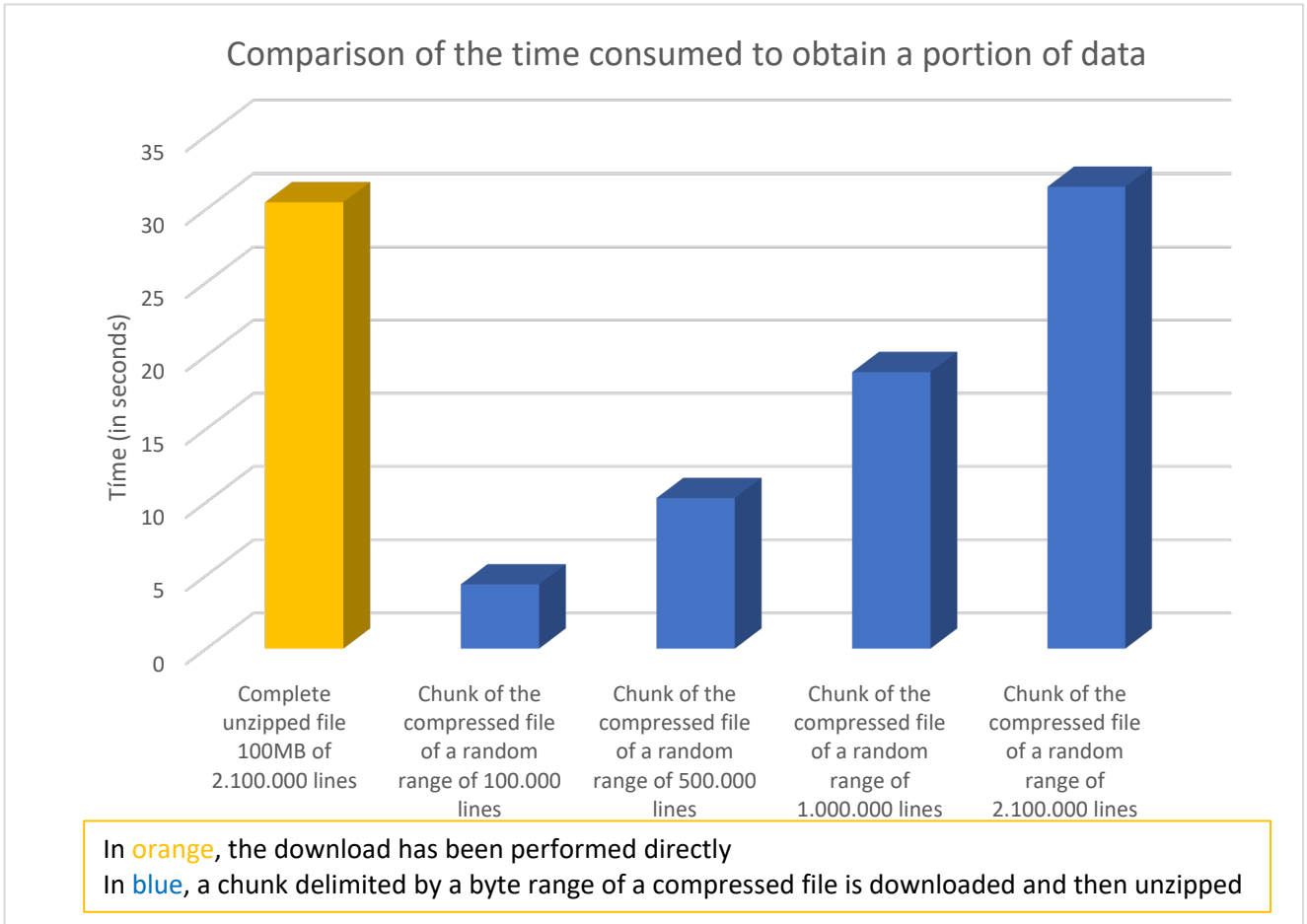

Figure 27. Comparison of the time consumed to obtain a portion of data in our local machine

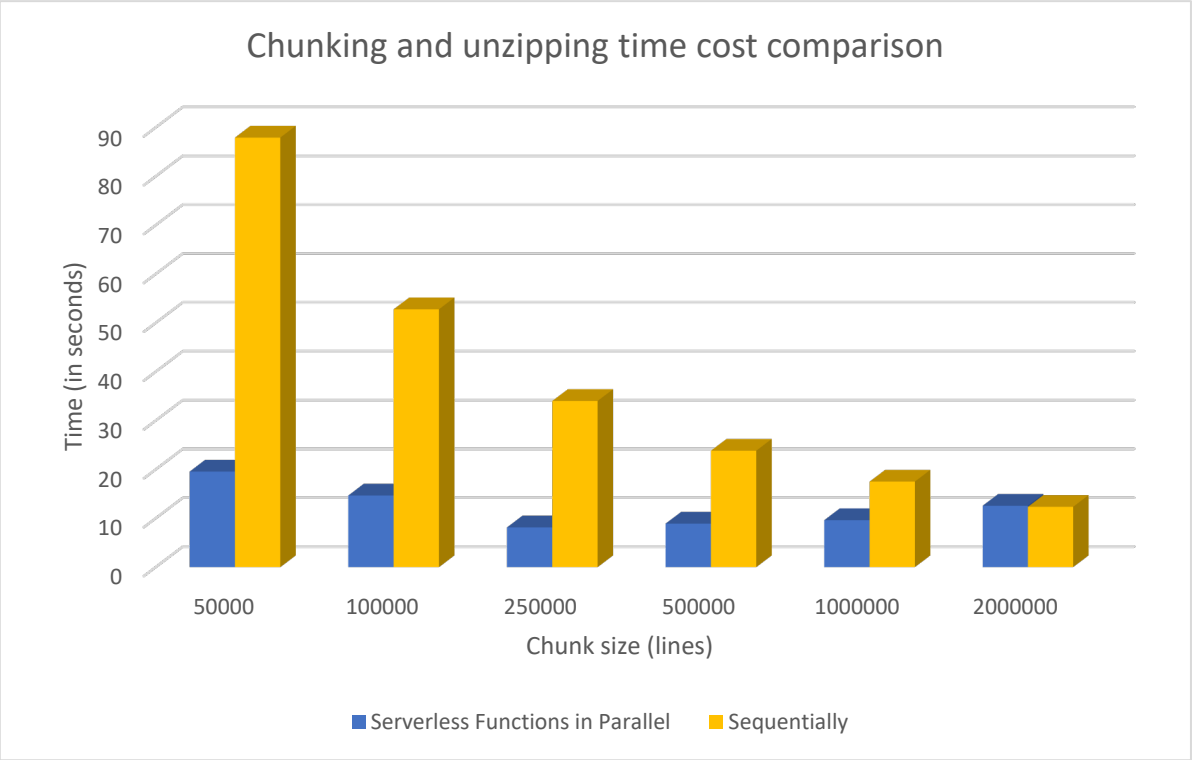

Figure 29. Chunking and unzipping time cost comparison (file stored in IBM COS bucket)

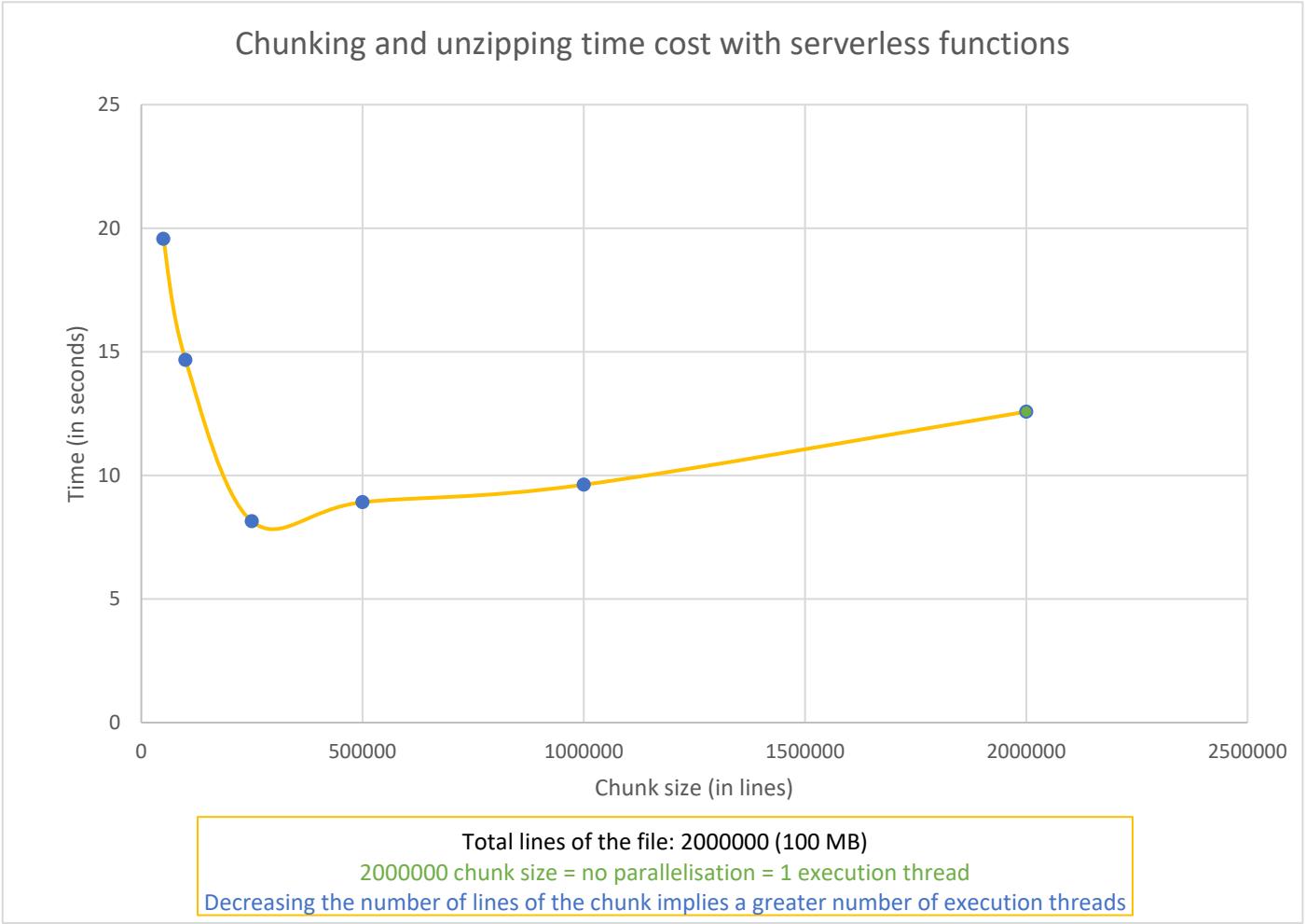

Figure 28. Chunking and unzipping time cost with serverless functions of a 100MB compressed text file

In summary, the access time to compressed portions of a Gzip file with our utility is way faster than downloading the entire file. Performing this portioning is parallel is much faster than doing it sequentially, since each thread of execution only must handle a small portion of data.

7.4 Performance over serverless functions

To evaluate the performance of our partitioning utility for Gzip-compressed files, measurements were made comparing the required time to obtain the entire chunked file using serverless functions. In the experiment, the required times when the downloads were performed over the original complete file, were compared to the required times when the downloads plus extractions were carried out of the compressed file. To carry through this analysis, two files were used, a smaller file of 100 MB (~2M lines) and a larger one of 5 GB (~100M lines). The results can be observed in Figure 30 and 31.

Similar behaviours were observed with both files, data obtention time through our utility is faster than direct download up to a certain point, thus highlighting two main ideas:

- If we only consider the results obtained through our partitioning utility, we can observer that performance reaches a maximum when the chunk size represents around 10% of the lines for the 100 MB file and around 5% of the lines for the 5% GB file, and then decreases. As previously observed in Figure 28, a trade-off must be found between the speed at which data is transmitted from the bucket to the container where our function is executed, the mappers management and the time penalty involved with the unzipping of a compressed file chunk. We observe when comparing the two figures, that the larger the file, the better the partitioner works.
- If we only consider the results obtained through direct download, we observe a similar pattern to the one mentioned previously (performance increases until a maximum and then decreases). For this reason, a balance must be found between the speed at which data is transmitted to the function's container and the management time of the mappers.
- Finally, if we contrast the results obtained by direct download and by download and decompression of the gzipped file (our partitioner), we can depict that with our utility there is a better performance, even at the peak of maximum performance. But, if the chunk's size is too small, the performance is worse than with direct download for the same chunk size, and, than for bigger chunk sizes with both techniques.

The larger the compressed file, the higher the speed-up our partitioner has compared to a direct download. There is a trade-off between chunk size and data access time, but again this experiment does not account for data handling.

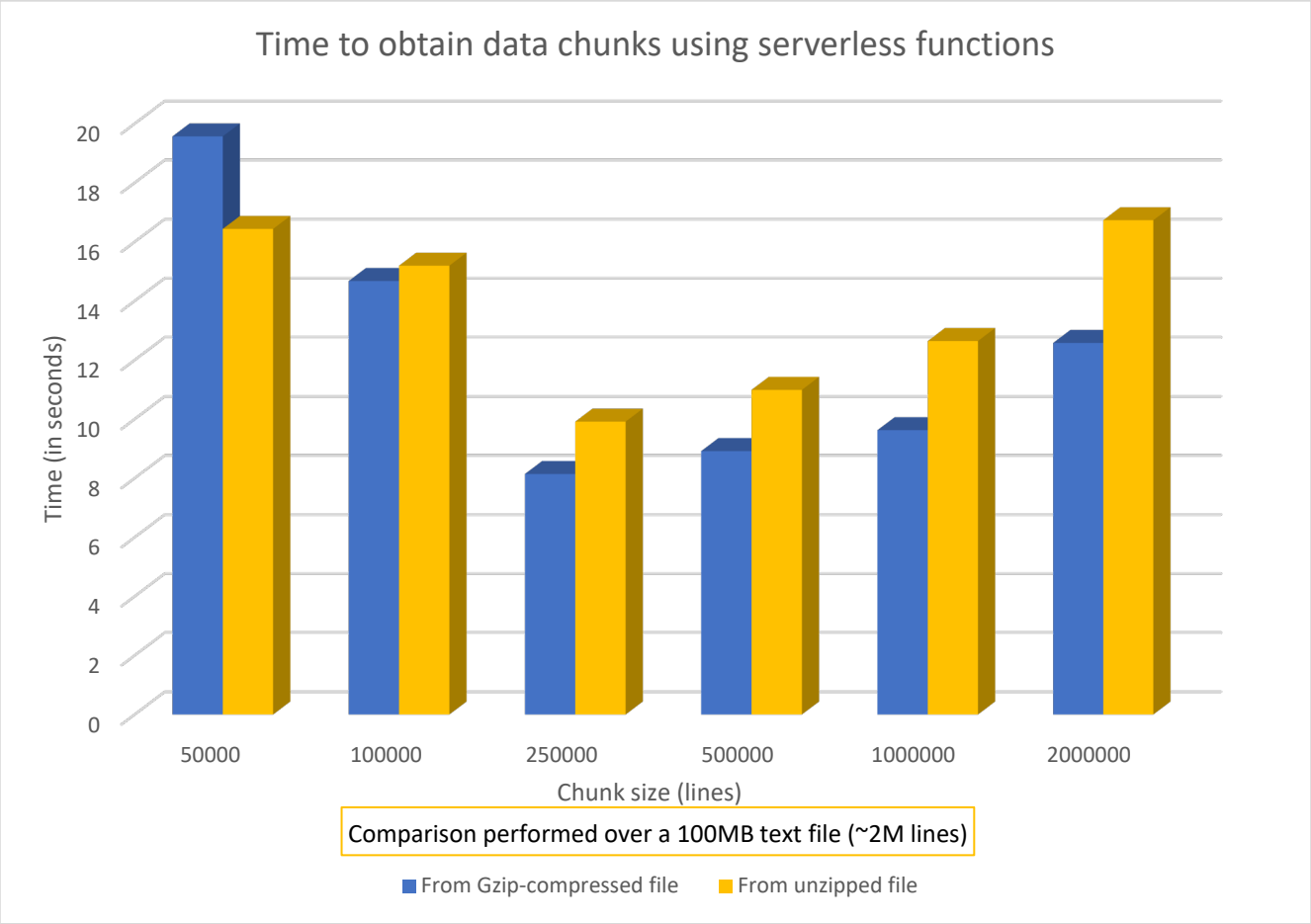

Figure 31. Time comparison to obtain data chunks using serverless functions (100 MB file)

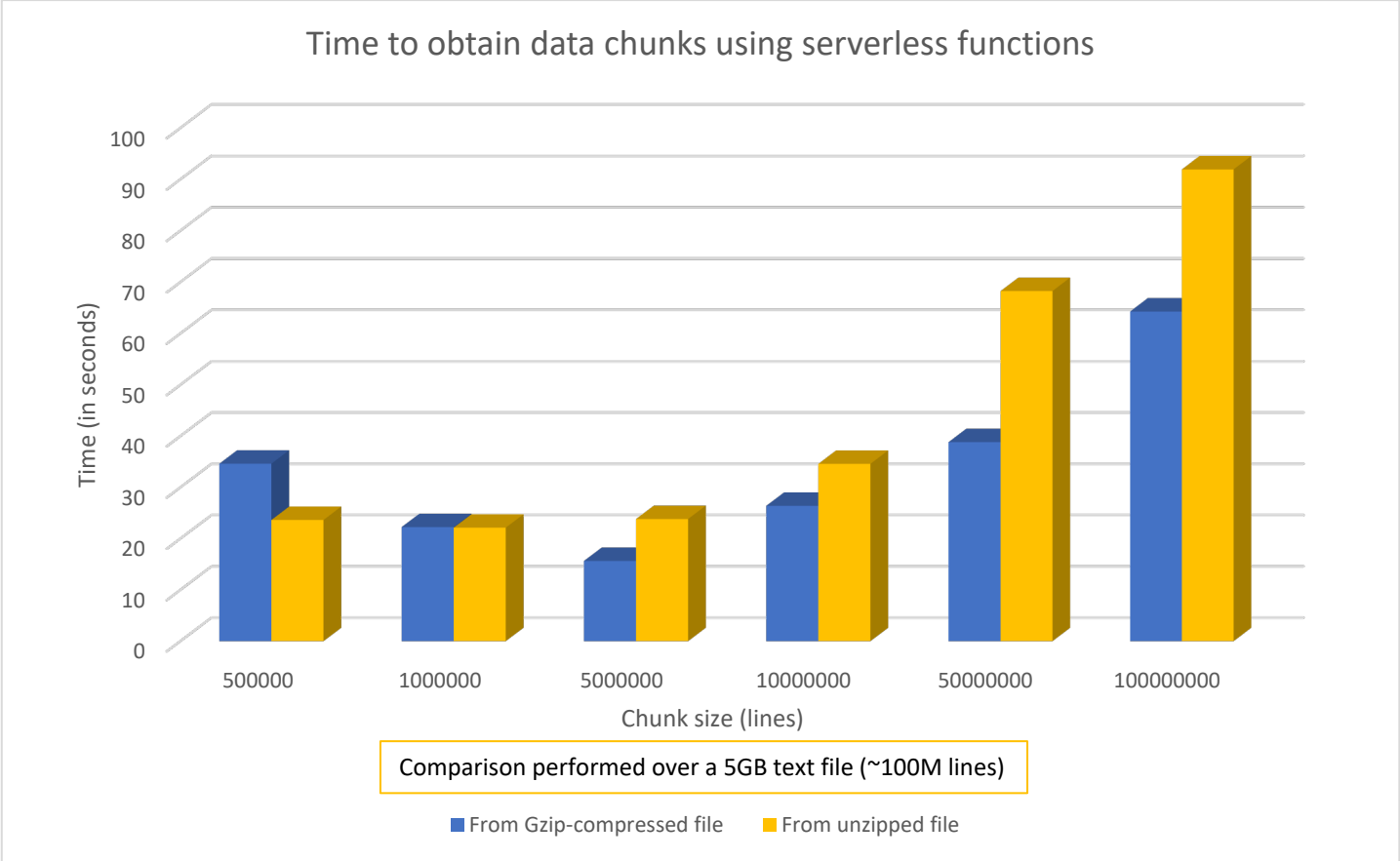

Figure 30. Time comparison to obtain data chunks using serverless functions (5GB file)

7.5 Cloudbutton's use case + partitioner performance

Finally, the performance of the integration of the Cloudbutton's genomics use case with our Gzip-compressed files partitioner was evaluated, obtaining for a relatively small dataset (30 MB compressed FASTQ file) a performance increase found between 30 and 40% (with the optimal chunk size) compared to its non-distributed version (maximum chunk size). The results are presented in Figure 32.

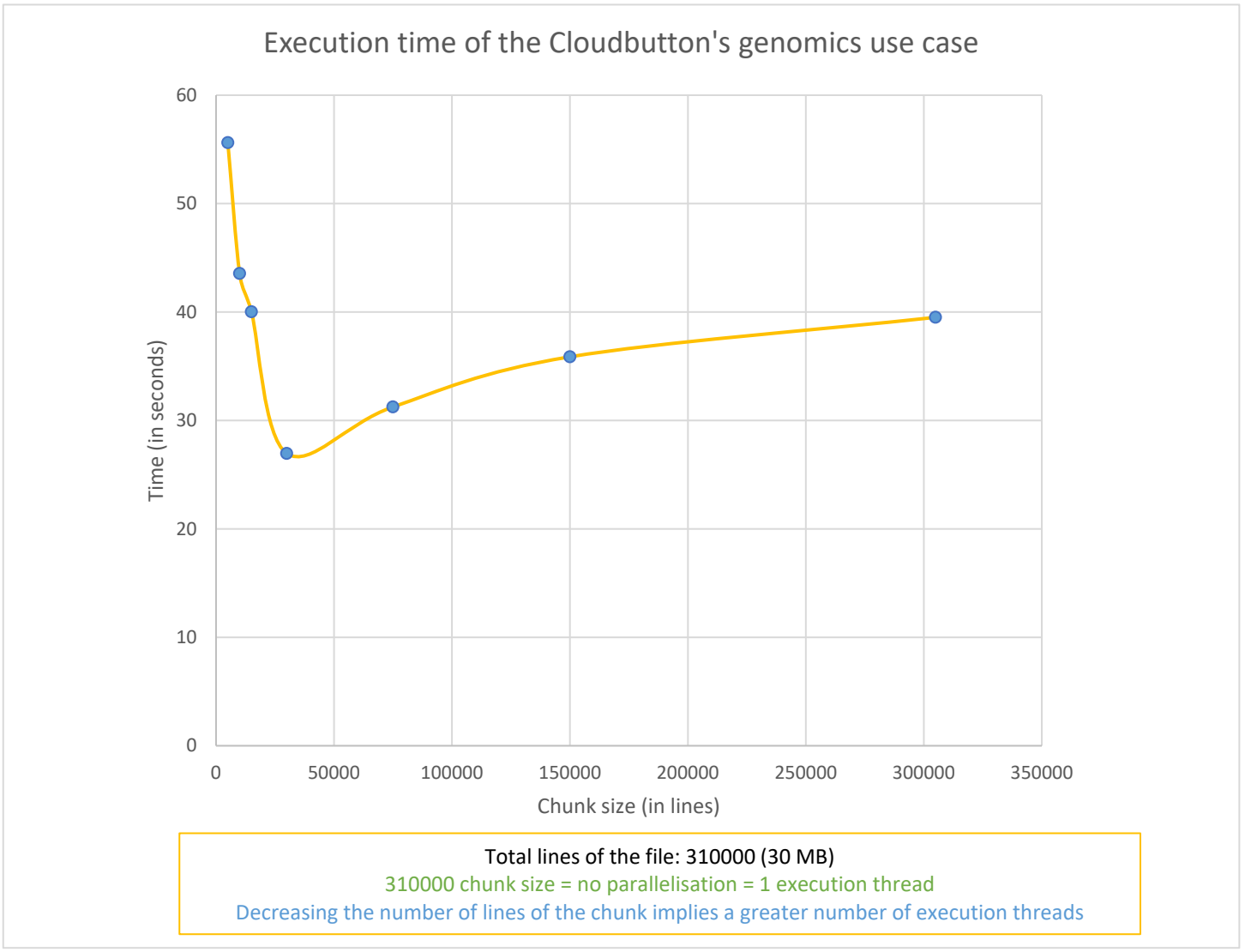

Figure 32. Execution time of the Cloudbutton's genomics use case incorporating our partitioner

Therefore, we can conclude that the 'on-the-fly' use of our Gzip-file partitioner in Cloudbutton's distributed genomic use case provides a real benefit since a shorter execution time can be observed.

8 Future perspectives

Once our Gzip-compressed files partitioner performance has been demonstrated for the execution of programs in a distributed manner through serverless functions, the next step is to take this utility to other compression formats such as ZIP, RAR or 7z, with the aim of reaching a wider audience of programmers whose datasets are stored in different compression formats.

An interesting approach on serverless architectures is managing its fine-grained billing model to reach optimal systems in all possible aspects. For this reason, it would be interesting to carry out a systematic study for the inference of the optimal number of workers based on the size and / or the lines that make up the file, thus obtaining a refined mathematical model that can be used by the programmer before deciding the chunk's size. This study would be difficult to implement since the mathematical model should also include the cost of data computation, and this cost varies depending on the processing that the programmer wants to give the given data.

Finally, although it seems that the development of the Gzip-compressed files partitioner for Lithops has concluded, I have decided to continue participating in this project of transition of CloudButton's genomics use cases to distributed computing through Lithops, which is integrated in CloudButton's Horizon 2020 Framework Program (H2020). Therefore, I have started developing a communication system between different threads of execution of a distributed program based on the Map-Reduce model, which will allow communication between mappers, between reducers and between both, making possible the migration of bioinformatics use cases to distributed computing.

9 Conclusions

By accomplishing this final degree project, it has been possible to respond to the challenge presented to the James Hutton Research Institute and Biomathematics & Statistics Scotland (BioSS), in terms of the migration of their genomics experiments to a cloud environment. In this study, a completely functional Gzip-compressed files partitioner has been implemented and has been prepared to be used 'on-the-fly' over serverless functions, facilitating the execution of genomic use cases with this serverless technology, and thus achieving the objectives set.

The complexity of the development of this project lies in the approach of different scientific fields such as molecular biology and computer engineering, as well as the approach of different informatics' technologies such as serverless computing and massive data analysis.

Thanks to the development of this partitioner and data retriever, from now on, bioinformaticians who want to run their experiments using Lithops on serverless platforms will be able to do so in the most simple and transparent way, enjoying a data-driven data analysis experience. Developers who perform genomic data analysis can now enjoy the benefits of serverless computing in a simple and efficient way, thanks to the automation of resource provisioning and management, with compressed data portioning and obtention into the bargain.

The impact of the implementation of this final degree project has been such, that the utility is already being used by genomic analysis research groups in multiple serverless platforms (IBM Cloud and AWS). Besides, in order to democratize knowledge in the field of bioinformatics, this scientific coalition intends to publish this project in one of its future research papers.

10 Self-evaluation

From my point of view and from my personal experience, I can affirm that this study has been a great final degree project which has taken me closer and made me more familiar with Cloud Computing and state-of-the-art architectures as serverless systems are.

Even though during the degree there is already an approach to Cloud Computing through subjects such as Distributed Systems, it is increasingly necessary to immerse oneself in a real project within a real research group to be able to understand the ins and outs of this computational paradigm.

Being able to independently make use of powerful systems such as IBM Cloud servers gave me a lot of confidence and made me adopt and feel closer and reachable computational concepts that were previously overwhelming.

Thanks to this experience, I am now fully aware of how Computer Science and Biology can work together hand in hand to solve the new questions that threaten mankind. I realized that Biology needs the help of Computer Science more than ever in order to continue evolving, giving an important role to the double degree I have studied, the URV's double degree in Biotechnology and Computer Science.

Finally, I would like to express my gratitude and give credit to Roberto S. Galende (https://github.com/circulosmeos) author of Gztool, for helping me solving the bugs found in its utility and making this project possible, as well as to CloudButton for providing the developers community with Lithops.

References

[1] Taibi, D., Spillner, J., & Wawruch, K. (2020). Serverless computing-where are we now, and where are we heading?. IEEE Software, 38(1), 25-31.

[2] Langmead, B., & Nellore, A. (2018). Cloud computing for genomic data analysis and collaboration. Nature Reviews Genetics, 19(4), 208-219.

[3] Burton, P. R., Hansell, A. L., Fortier, I., Manolio, T. A., Khoury, M. J., Little, J., & Elliott, P. (2009). Size matters: just how big is BIG? Quantifying realistic sample size requirements for human genome epidemiology. International journal of epidemiology, 38(1), 263-273.

[4] Cock, P. J., Fields, C. J., Goto, N., Heuer, M. L., & Rice, P. M. (2010). The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. Nucleic acids research, 38(6), 1767-1771.

[5] Matsunaga, A., Tsugawa, M., & Fortes, J. (2008, December). Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In 2008 IEEE Fourth International Conference on eScience (pp. 222-229). IEEE

[6] Deutsch, P. (1996). GZIP file format specification version 4.3.

[7] Marinescu, D. C. (2017). Cloud computing: theory and practice. Morgan Kaufmann.

[8] "Amazon EC2." https://aws.amazon.com/ec2 (accessed Aug. 18, 2021).

[9] "Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service (S3)." https://aws.amazon.com/s3 (accessed Aug. 18, 2021).

[10] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., ... & Zaharia, M. (2009). Above the clouds: A berkeley view of cloud computing (Vol. 17). Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.

[11] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C. C., Khandelwal, A., Pu, Q., ... & Patterson, D. A. (2019). Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383.

[12] "AWS Lambda – Serverless Compute – Amazon Web Services." https://aws.amazon.com/lambda (accessed Aug. 20, 2021).

[13] García-López, P., Sánchez-Artigas, M., Shillaker, S., Pietzuch, P., Breitgand, D., Vernik, G., ... & Ferrer, A. J. (2019). Servermix: Tradeoffs and challenges of serverless data analytics. arXiv preprint arXiv:1907.11465.

[14] "Azure Functions serverless compute | Microsoft Azure." https://azure.microsoft.com/en-gb/services/functions/ (accessed Aug. 25, 2021).

[15] "Cloud Functions | Google Cloud." https://cloud.google.com/functions/ (accessed Aug. 25, 2021).

[16] "IBM Cloud Functions." https://cloud.ibm.com/functions/ (accessed Aug. 25, 2021).

[17] "Apache Hadoop." https://hadoop.apache.org/ (accessed Aug. 25, 2021).

[18] "Apache SparkTM - Unified Analytics Engine for Big Data." https://spark.apache.org/ (accessed Aug. 25, 2021).

[19] Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., & Recht, B. (2017, September). Occupy the cloud: Distributed computing for the 99%. In Proceedings of the 2017 Symposium on Cloud Computing (pp. 445-451).

[20] "Riselab | UC Berkeley | Projects | PyWren" https://rise.cs.berkeley.edu/projects/pywren/ (accessed Aug. 25, 2021).

[21] Sampé, J., Vernik, G., Sánchez-Artigas, M., & García-López, P. (2018, December). Serverless data analytics in the ibm cloud. In Proceedings of the 19th International Middleware Conference Industry (pp. 1-8).

[22] "Lithops-cloud / Lithops" https://github.com/lithops-cloud/lithops (accessed Aug. 30, 2021)

[23] "Empowering App Development for Developers" https://www.docker.com/ (accessed Oct. 13, 2021)

[24] "Apache | OpenWhisk | Open Source Serverless Cloud Platform" https://openwhisk.apache.org/ (accessed Aug. 30, 2021)

[25] Calabrese, B., & Cannataro, M. (2016). Cloud Computing in Bioinformatics: current solutions and challenges (No. e2261v1). PeerJ Preprints.

[26] Aniba, M. R., Poch, O., & Thompson, J. D. (2010). Issues in bioinformatics benchmarking: the case study of multiple sequence alignment. Nucleic Acids Research, 38(21), 7353-7363.

[27] "IBM Cloud Object Storage" https://www.ibm.com/cloud/object-storage (accessed Aug. 30, 2021)

[28] "IBM Cloud – IBM Cloud Learn Hub – What is Serverless Computing? – Serverless Computing" https://www.ibm.com/cloud/learn/serverless (accessed Aug. 24, 2021)

[29] Gztool https://github.com/circulosmeos/gztool (accessed Oct. 18, 2021)

[30] “Zlib A Massively Spiffy Yet Delicately Unobtrusive Compression Library” https://zlib.net/ & https://github.com/madler/zlib (accessed Sep. 3, 2021)

[31] “zran.c indexing a gzip stream and randomly accessing it”

[32] Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. IEEE Transactions on information theory, 23(3), 337-343.

[33] https://github.com/madler/zlib/blob/master/examples/zran.c (accessed Sep. 3, 2021)

Annex A: Accessing and configuring Lithops

All information about Lithops can be found at its GitHub repository.

https://github.com/lithops-cloud/lithops

By default, Lithops works on 'Localhost' if no configuration is provided. To run workloads on the Cloud, compute and storage backends must be configured. First you will need an account on your favourite Cloud provider, in this study we used an IBM Cloud's (https://cloud.ibm.com/) smart tier account, which provides us with free tiers for IBM Cloud Functions and IBM COS (bucket). To configure Lithops, once you have ensured that your account is up in both services, you only have to create/modify Lithops' configuration file (~/.lithops/config) by adding your IBM COS and IBM Cloud Functions credentials. For more information about Lithops' configuration, please check Lithops' configuration instructions.

Annex B: Disponibility of the code and datasets

All the code commented during this study can be found and is freely accessible on my personal GitHub website, as well as in CloudButton's private gitlab repository (Biomathematics & Statistics Sctoland, James Hutton Institute). The link to the specific repository of the project in my GitHub website is the following:

https://github.com/Damian-MG/TFG

Annex C: Bash scripts

```
#!/bin/bash
# Script generateUploadIndexInfoBucket.sh: Takes a gzip file and creates an index for it,
reformates the index info

# COMMAND LINE INPUT
file=$1

# 1. INDEX FASTQ.GZ AND REFORMAT INDEX INFO
# Create index -i, with line number information -x, with span in uncompressed MiB between
index points
gztool -i -x -s 1 $file
# Produce index info: showing internals of all index files in this directory -e, showing
data about each index point
gztool -ell "${file}i" > "${file}i.info"
# Get total lines and size form filei.info
tot_lines=$(awk ' /Number of lines/ { print $5 } ' "${file}i.info")
tot_size=$(awk ' /Guessed gzip/ { print $10 } ' "${file}i.info" | awk
'{gsub(/^.{1}/,"");}1')
# add 1 to tot_size, as in all blocks 1 is subtracted to match end of previous block
tot_size="$((tot_size + 1))"
# reformat .info file to have all points in list start at new line and remove header
sed -e 's/#/\n#/g' -e 's/L//g'  "${file}i.info" |
    awk ' /^#[0-9]+/ { printf ("%s %s\n", $3, $6);} ' > "${file}i_tab.info"
echo $tot_size" "$tot_lines >> "${file}i_tab.info"

echo "Total lines:"$tot_lines
echo "Total size:"$tot_size
```

Figure 33. `generateIndexInfo.sh` bash script

```
#!/bin/bash
# Script generateChunks.sh: From the previous index files generated by the gztool,
generates the byte ranges that will be used to retrieve the chunks of the gzip file

# COMMAND LINE INPUT
file=$1
block_lenght=$2
total_lines=$3

rm "${file}.chunks.info"

# 1. GENERATE LINE INTERVAL LIST
intervals=$(awk -v block_len="$block_lenght" -v totlines="$total_lines" '
   BEGIN {
      end=""
      for( i = 1; i+block_len < totlines; i+=block_len )
         print paste i":"i+block_len;
         end=i
      print paste end":"totlines;
   }')

# 2. OBTAINING THE BYTE RANGES CORRESPONDING TO THE LINE INTERVALS
for intr in $intervals;
do
    # extract start and end line no.
    start=$(echo $intr | awk -F':' '{print $1}')
    end=$(echo $intr | awk -F':' '{print $2-1}')
    # get start and end block
    # find closest point with line no. lower than target line no.
    start_block=$(awk -v val="$start" '
        BEGIN{c=2; l=$c}
        {d=val-$c; if (d >= 0) {l=$c; bl=$1}}
        END{print bl}' "${file}i_tab.info")
    # find start of first block after desired chunk, to get last value of last block in
chunk
    end_block=$(awk -v val="$end" '
        BEGIN{c=2; l=$c}
        {d=$c-val; if (d >= 0 && d_prev <=0) {l=$c; bl=$1-1}
        d_prev=d
        }
        END{print bl}' "${file}i_tab.info")
    echo "Start and end lines of the chunk:" $start $end "Start and end Bytes of the
chunk:" $start_block $end_block
    echo $start" "$end" "$start_block" "$end_block >> "${file}.chunks.info"
done
```

Figure 34. `generateChunks.sh` bash script

```
#!/bin/bash
# Script generateChunks.sh: From the previous index files generated by the gztool,
generates the byte ranges that will be used to retrieve the chunks of the gzip file

# COMMAND LINE INPUT
file=$1
block_lenght=$2
total_lines=$3

rm "${file}.chunks.info"

# 1. GENERATE LINE INTERVAL LIST
intervals=$(awk -v block_len="$block_lenght" -v totlines="$total_lines" '
   BEGIN {
      end=""
      for( i = 1; i+block_len < totlines; i+=block_len )
         print paste i":"i+block_len;
         end=i
      print paste end":"totlines;
   }')

# 2. OBTAINING THE BYTE RANGES CORRESPONDING TO THE LINE INTERVALS
for intr in $intervals;
do
    # extract start and end line no.
    start=$(echo $intr | awk -F':' '{print $1}')
    end=$(echo $intr | awk -F':' '{print $2-1}')
    # get start and end block
    # find closest point with line no. lower than target line no.
    start_block=$(awk -v val="$start" '
        BEGIN{c=2; l=$c}
        {d=val-$c; if (d >= 0) {l=$c; bl=$1}}
        END{print bl}' "${file}i_tab.info")
    # find start of first block after desired chunk, to get last value of last block in
chunk
    end_block=$(awk -v val="$end" '
        BEGIN{c=2; l=$c}
        {d=$c-val; if (d >= 0 && d_prev <=0) {l=$c; bl=$1-1}
        d_prev=d
        }
        END{print bl}' "${file}i_tab.info")
    echo "Start and end lines of the chunk:" $start $end "Start and end Bytes of the
chunk:" $start_block $end_block
    echo $start" "$end" "$start_block" "$end_block >> "${file}.chunks.info"
done
```

Figure 35. `randomChunkRange.sh` bash script