Towards Analyzing Computational Costs of Spark for SARS-CoV-2 Sequences Comparisons on a Commercial Cloud*

Alan L. Nunes[1], Alba Cristina Magalhaes Alves de Melo[2], Cristina Boeres[1], Daniel de Oliveira[1], Lúcia Maria de Assumpção Drummond[1],

[1] Instituto de Computação – Universidade Federal Fluminense (UFF)
Niterói – RJ – Brasil

[2] Departamento de Ciência da Computação – Universidade de Brasília (UnB)
Brasília – DF – Brasil

alan_lira@id.uff.br, alves@unb.br, {boeres,danielcmo,lucia}@ic.uff.br

Abstract. *In this paper, we developed a Spark application, named* Diff Sequences Spark*, which compares 540 SARS-CoV-2 sequences from South America in Amazon EC2 Cloud, generating as output the positions where the differences occur. We analyzed the performance of the proposed application on selected memory and storage optimized virtual machines (VMs) at on-demand and spot markets. The execution times and financial costs of the memory optimized VMs outperformed the storage optimized ones. Regarding the markets,* Diff Sequences Spark *reduced the average execution times and monetary costs when using spot VMs compared to their respective on-demand VMs, even in scenarios with several spot revocations, benefiting from the low overhead fault tolerance Spark framework.*

1. Introduction

Over the last decades we have witnessed the generation of an unprecedented volume of data both in academia and industry [Hey and Trefethen 2020]. According to Domo[1], in 2020 each person on Earth produced 1.7MB of data every second. One challenge that arises is how to process/query this volume of data and extract useful knowledge from it in a timely manner. Since the traditional data management solutions such as Relational Database Management Systems (RDBMS) do not scale for this volume of (commonly heterogeneous) data [Hu et al. 2014], alternative solutions must be considered. In fact, several new approaches were proposed to bridge this gap. The most successful are the well-known Big Data frameworks, with Apache Spark being one of the most widely adopted, due to its high scalability and increasing popularity [Zaharia et al. 2010]. Spark improves the performance of applications by performing in-memory data movement (in contrast with Hadoop where operations are disk to disk) and automatically exploiting parallelism. An interesting characteristic of Spark is that it allows for users to develop their big data analytical applications without concerning about the parallel processing issues [Perera et al. 2016].

Although Spark can be deployed on a single computer or a cluster, it can further benefit from cloud environments [Yan et al. 2016]. Cloud providers aim at maximizing resource utilization and profits by delivering underutilized computing resources. For instance, Amazon AWS cloud offers a variety of resources for executing Spark applications.

*This research is supported by project CNPq/AWS 440014/2020-4, Brazil and by *Programa Institucional de Internacionalização* (PrInt) from CAPES (process number 88887.310261/2018-00).

[1] https://www.domo.com/solution/data-never-sleeps-6

This environment offers several advantages compared to dedicated infrastructures, such as rapid provisioning of resources and significant reduction of operational costs. Revocable virtual machines (VMs) are available at a reduced price on the *spot* market, that can be up to 90% cheaper than their *on-demand* counterparts (which are offered with a fixed financial cost per time unit of use and are not revoked by the provider). While big data applications can profit dramatically from executing on spot instances, their performance can, however, degrade, mainly if no fault-tolerance mechanism is provided. Running Spark in the cloud presents a range of challenges. For example, one of them is the selection of the proper configuration of parameters, both in Spark and cloud levels [de Oliveira et al. 2021]. There are many opportunities for optimizations. However, the user has to explore a huge configuration space in order to efficiently execute Spark applications. A poor choice of parameters not only significantly degrades the execution performance, but may also lead to big financial costs what can make the execution of the application unfeasible.

To illustrate the problem of optimizing Spark applications in a public cloud, the following example from the bioinformatics domain is consider: the studies associated with the Covid-19 pandemic are of particular interest, and the comparison of SARS-CoV-2 sequences is crucial to understand the behavior of this disease. More than a million SARS-CoV-2 sequences are available for general use in public genomic databases, *e.g.*, NCBI[2]. The comparisons of such sequences generate big textual files with alignments [Rochman et al. 2021] and it is difficult for the Biologist to identify mismatched positions, which correspond to mutations and their analyses is of great interest.

In order to identify the aspects related to performance and costs when running Spark applications on Amazon AWS cloud, we developed a Spark application, named *Diff Sequences Spark*[3], which compares biological sequences (all-against-all) and generates as output the positions where the nucleotide differences occur. We propose two versions of this Spark application that differ in the way data are structured and processed. Analyzes performed in preliminary tests showed that the Collection phase, which saves the results periodically in the disk, and the Diff phase, executed in memory, dominated the application execution time. Considering those results, we explored both memory and storage optimized virtual machines (VMs) types and markets (*i.e.*, *on-demand* and *spot*) to compare 540 SARS-CoV-2 sequences from South America. Previously in [Teylo et al. 2021, Brum et al. 2021], we observed the benefits of spot market when either the application can handle failures or it is executed within a framework that provides recovering mechanism in case of VM revocations. Since Spark provides fault tolerance mechanisms, we also investigate the effect of spot instances revocations, in terms of execution time and financial cost in our Spark application.

As an outcome of our contributions, performance results in Amazon EC2 reveals that *Diff Sequences Spark* application was able to run even on the VMs revocation scenarios, obtaining reductions of execution time up to 22.60% and of financial cost up to 62.22% when using the z1d.xlarge spot instance in comparison to the r5.xlarge on-demand instance (the cheapest on-demand instance used in the experiments) for 540 SARS-CoV-2 sequences comparison. This shows that clouds can play a fundamental role in ensuring the efficiency of the execution of such applications with reduced costs. The remainder of

[2]National Center for Biotechnology Information - `https://www.ncbi.nlm.nih.gov/`

[3]`https://github.com/alan-lira/diff-sequences-spark`

this paper is organized as follows. Section 2 details the Apache Spark framework. Section 3 brings background and related work. Section 4 describes the developed Spark application for all-against-all biological sequences comparison. Section 5 presents experimental evaluation and, finally, conclusions are drawn in Section 6.

2. Apache Spark in a Nutshell

Apache Spark is a large-scale data processing framework designed for optimizing both batch and iterative parallel operations over large datasets. Spark executes the applications by chaining a series of operations [Zaharia et al. 2010] and avoids the significant I/O overheads found in other frameworks such as Apache Hadoop. The main advantage of Spark is that it aims at keeping the data in memory (if possible) during the processing, which avoids reading data from and writing results back to file. In the context of a Spark application, we can find the *Driver* and the *Workers*. The Driver is the process where the user submits jobs in Spark (*i.e.*, it controls the execution). Workers are nodes where the data are processed, and each Worker may have several associated *Executors*, which in turn execute *Tasks* in a specific job. The number of Executors in a Worker is defined by the user, and it can vary according to the computing environment chosen to execute the Spark application.

One of the key advantages of Spark in comparison to other MapReduce frameworks is its in-memory structures such as *Resilient Distributed Dataset* (*RDD*) [Zaharia et al. 2012] and *DataFrames* [Armbrust et al. 2016], which essentially are in-memory collections of partitioned data instances that can be processed in parallel [Zaharia et al. 2012, Zaharia et al. 2010]. While RDDs are sets of objects representing data, DataFrames are distributed collections of data with named columns, *i.e.*, DataFrames act as tables in relational databases (*e.g.*, PostgreSQL, Oracle, *etc*.). Besides the advantage of representing data as a well-known relational table, DataFrames also benefit from the Spark's Catalyst Optimizer [Armbrust et al. 2016]. Catalyst is focused on optimizing the query processing, and thus has several sets of rules that handle different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile optimized queries for execution. Spark also provides more programming abstractions than other frameworks, such as filter, join, and collect [Zaharia et al. 2012] instead of just classical Map and Reduce operations found in Hadoop. In particular, all Apache Spark operators are classified in (i) *Transformations* and (ii) *Actions*. A *transformation* produces a new RDD from an existing one, whereas an *action* enables the modification of current datasets without generating new RDDs. Spark evaluates both RDDs and DataFrames lazily, *i.e.*, it does not compute their result immediately. Instead, Spark registers that a transformation is applied to some dataset (in an RDD or DataFrame) and then computes transformations only when an action requires a result to be sent to the Driver.

Besides the advantage of in-memory data processing, Spark also provides useful fault tolerance mechanisms. Two types of failures may occur in the context of a Spark application: (i) Worker failure and (ii) Driver failure. In the case of failure in the Driver, the SparkContext (*i.e.*, the entry point to Spark) becomes unavailable and all Executors lose their in-memory data. If the Worker fails, all Executors associated with that Worker are killed (and of course all data in their memory are lost). However, the data are commonly replicated to other Worker nodes to achieve fault tolerance. The RDD has the capability of handling if any failure occurs in the Worker. This is possible since Spark creates a logical execution plan (*i.e.*, lineage graph) for all tasks executed in the context of an application.

For example, if a Worker fails during the execution and an RDD is lost, Spark can apply the same computation on that Worker (by following the lineage graph) to generate the same dataset again. Spark can also work together with Apache Mesos [Hindman 2011] to provide another level of fault tolerance, where the Executors are restarted if they fail and a backup of the master node can be maintained, which is useful in the case of driver failures.

3. Biological Sequences Comparison

A biological sequence is a molecule of nucleic acid or protein. It is represented by a linear list of residues, which are nucleotide bases (for DNA and RNA sequences) or amino acids (for protein sequences). An alphabet of four nucleotides ($\Sigma = \{A, T, G, C\}$) is used for DNA sequences [Durbin et al. 1998]. Even though SARS-CoV-2 sequences represent the virus responsible for the covid-19 disease (*i.e.*, RNA), DNA sequences of infected hosts are the ones stored in public repositories. In this study, we use DNA SARS-CoV-2 sequences with host=human. DNA sequences are usually compared with sequence alignment algorithms, which produce a textual file where the matching characters are highlighted [Durbin et al. 1998]. When highly similar sequences are compared, which is the case of SARS-CoV-2 sequences and its variants, the biologists are mostly interested in the differences, which may indicate mutations, not in the similarities, which are highlighted in the alignment file. Besides, since there are more than a million SARS-CoV-2 in the public repositories, the analysis is often done by geographical region and period of time, comparing a set of sequences to each other (all-against-all). Therefore, we advocate that a tool that performs all-against-all comparisons and outputs the differences among the sequences is extremely useful for SARS-CoV-2 genetic studies [Lau et al. 2021].

3.1. Related Work

SparkSW [Zhao et al. 2015], as remarked by the authors, is the first Spark-based implementation of the Smith-Waterman algorithm that provides scalability and load-balancing efficiency for biological sequence pairwise alignment on distributed environment. *DSA* [Xu et al. 2017a] leverages data parallel strategy based on SIMD instruction to parallelize the algorithm in each core associated to a worker node and employs memory-based distributed file system *Alluxio*[4] as primary storage to speed up I/O performance and reduce network traffic. *CloudSW* [Xu et al. 2017b] also leverages Spark and SIMD instructions to accelerate SW algorithm and supports both alignment scores and trace-backs.

To the best of authors' knowledge, few works investigate the challenges of executing Spark-based applications on cloud spot VMs. *TR-Spark* [Yan et al. 2016], is a framework that allows for executing Spark applications on *transient resources* (spot instances in the case of AWS EC2) based on resource stability and data size reduction aware scheduling and lineage-aware checkpointing. [Yan et al. 2016] specify background tasks on nodes that are temporarily not fully utilized for their primary tasks. In their framework, re-computation costs are minimized by backing up intermediate results according to resources instability level, re-computation cost, and data lineage. Also, to reduce the re-computations costs, *TR-Spark* prioritizes those tasks that output the least amount of data. They devise a proactive checkpointing policy, by saving data blocks that cannot be read and processed by its next stage before the virtual instances failure. This revocation time

[4] https://www.alluxio.io

is defined following a probabilistic approach. Although they plan to minimize checkpointing overheads, revocation time is hard to accurately predict. Albeit the paper has shown promising performance results, close to those Spark executions on stable resources, their work is based on the stability prediction of the VMs. [Xu et al. 2019] propose *iSpot*, a resource provisioning framework that explores the use of transient servers in the cloud. The framework classifies stable transient servers during the job execution based on the Long Short-Term Memory (LSTM) method (*i.e.*, an artificial recurrent neural network architecture). Also, *iSpot* includes Spark application performance modeling by predicting the performance of Spark stages and jobs based on automatic job profiling. Furthermore, an analytical performance checkpointing mechanism is provided, considering the Spark performance model, data checkpointing and restoration overheads. Although the results show a decrease on the financial costs, their analysis relies only on the use of spots.

4. Spark-based All-Against-All Sequences Comparison

Rather than proposing a novel pairwise sequence alignment tool, the all-against-all sequences comparison is performed through the following phases: Diff and Collection. The Diff phase finds all different nucleotide letters occurrences from the input sequences, while the Collection phase consists of persisting these occurrences as comma-separated-values output files into non volatile disk storage. Further details are presented as following.

4.1. Notation

Let $S = \langle s_1, s_2, \ldots, s_N \rangle$ be the input list of nucleotide enumerated sequences to be compared. These sequences are represented in DataFrames, which are distributed collections of data with named columns. Thus, each DataFrame R is composed of a set of tuples (*i.e.*, $\{r_1, ..., r_d\}$ with $d = |R|$), that follows a schema $\Re$, with its respective attribute types. In the context of this paper, the schema $\Re$ has one attribute for each position of the enumerated sequence, and each attribute stores a char that represents one of the four nucleotide bases: (i) adenine - A, (ii) cytosine - C, (iii) guanine - G, and (iv) thymine - T. In the proposed approach, we define two sets of DataFrames according to the number of sequences they contain during the processing: (i) *single sequence* and (ii) *multiple sequences*. Let D_s be the set of *single sequence* DataFrames, where each $d(i) \in D_s$ stores a single sequence s_i. In turn, D_m is the set of *multiple sequences* DataFrames, where each $d(i) \in D_m$ stores the sequences $\{s_{i+1}, \ldots, s_N\}$.

4.2. Diff Phase

Two different versions of the Diff phase, named DIFF_1 and DIFF_{opt}, are presented. Although both approaches focus on comparing pairs of input sequences, their main difference resides in how the Spark DataFrames are generated and compared. Regarding the Diff phase itself, it consists of the following Spark transformation functions: join, filter and drop. Basically, the Diff phase does a positional comparison of DataFrames' nucleotide letters to identify which are different. For a given position, if any empty (null) nucleotide letter is found then the respective position is discarded from d_r as it means that one sequence is longer than the other, which makes them incomparable in that position.

In $\textbf{DIFF}_1$, an all-against-all comparison is carried out considering only the DataFrames in D_s, *i.e.*, during the Diff phase, the comparison is held between $d(i) \in D_s$ and $d(j) \in D_s, \forall i < N, j = i+1, \ldots, N$. While this version seems quite intuitive, it hides

a Spark application optimization issue: each DataFrame pair comparison (Diff phase) produces a resulting DataFrame d_r. Due to an all-against-all DataFrames comparison, the estimate Diff phases amount dp_a is $\frac{N(N-1)}{2}$. Figure 1 illustrates the resulting DataFrames obtained from DIFF$_1$ implementation when $N = 4$ micro sequences are compared.

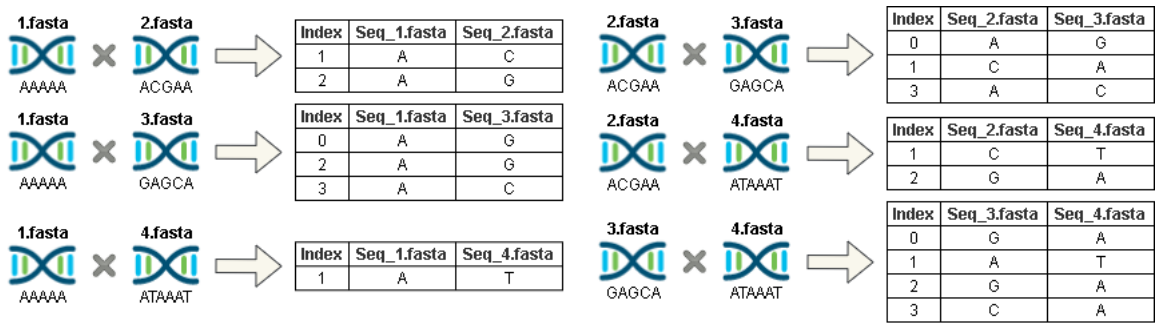

Figure 1. Results of DIFF$_1$ comparisons with $N = 4$ micro sequences.

DIFF$_{opt}$, on the other hand, works on both sets D_s and D_m by performing a Diff phase on $d(i) \in D_s$ and the respective $d(i) \in D_m$, for $i = 1, \ldots, N-1$. Since the DataFrames being compared are *single* sequence DataFrames and multiple sequences DataFrames, Spark optimizes transformation functions (join, filter and drop), by performing a multisequence parallel comparison. Nonetheless, due to Spark limitations on the multisequence parallel comparison mainly for large N, a DataFrame has a limited size max_D. In this way, whenever $N > max_D$, each $d(i) \in D_m$ is divided in $\frac{N}{max_D}$ DataFrames of size max_D to be used in the Diff phase with the respective single sequence DataFrame $d(i) \in D_s$. The estimated Diff phases amount dp_a, with a small absolute error, is then, $\frac{N(N-1)}{max_D} - \frac{N(N-max_D)}{2max_D}$, if $1 \leq max_D < \frac{N}{2}$; or $2(N-1) - max_D$, if $\frac{N}{2} \leq max_D < N$. Each line in each resulting DataFrame $d_r(i)$ produced by DIFF$_{opt}$ refers to the positional comparison between the nucleotide letter of the sequence in $d(i) \in D_s$ and the nucleotide letters of every sequence in $d(i) \in D_m$. For a given position, if they are mismatched, the corresponding nucleotide letters is shown in $d_r(i)$, otherwise a '=' character is produced. **Figure 2** illustrates the resulting DataFrames obtained from DIFF$_{opt}$ implementation when the same $N = 4$ micro sequences are compared with $max_D = 3$.

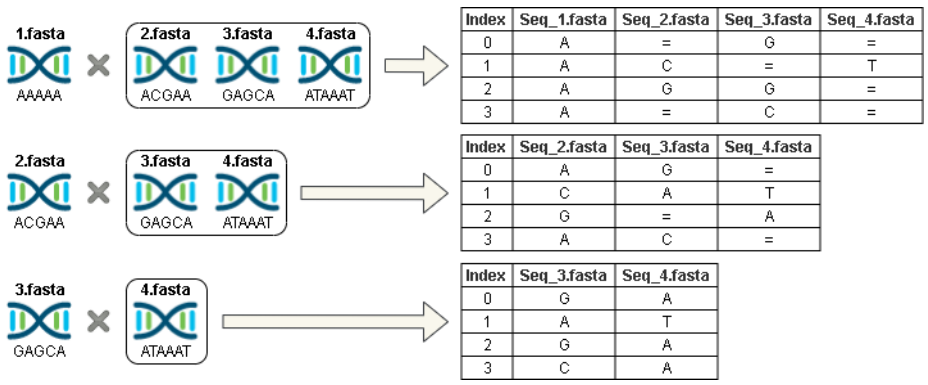

Figure 2. DIFF$_{opt}$ comparisons with $N = 4$ micro sequences and $max_D = 3$.

4.3. Spark DataFrames Customized Partitioning

Partitions are logical chunks of data which Spark defines and distributes over Worker nodes in the cluster in order to minimize the amount of I/O operations and to achieve efficient processing. Thus, each chunk of data assigned to a partition resides in a unique Worker. Let N_p be the number of partitions of a DataFrame and N_c, the number of CPU cores in the cluster. Whenever DataFrames are created, Spark automatically sets $N_p = N_c$. Furthermore, when any Spark transformation that triggers data shuffling over DataFrames is performed (*e.g.*, join function), the resulting DataFrame's N_p is *auto set* to 200. Having N_p defined, Spark creates a task per partition executed by Executor designated by Spark Scheduler. The auto setting can lead to poor performance depending on the amount of data loaded to each partition: either a small amount of data is distributed over too many partitions, corresponding to a large set of Spark tasks; or the opposite, each task is assigned to a large load that results in long running Spark tasks (too long tasks can cause an out of memory error). Yet, one must bear in mind that there is an overhead incurred due to Spark task creation, scheduling and management. This work proposes the use of the guidelines presented in Spark Tuning[5], and therefore, sets $N_p = 3N_c$ on DataFrames creation. Furthermore, the DataFrames re-partitioning after the join operation is carried out taking in account their estimated resulting size in bytes, N_c and a maximum size of 128 MB for each partition.

4.4. Collection Phase

The main purpose of the Collection phase is to consolidate the Spark transformations performed during the Diff phase. By calling a Spark 'write.csv' action, all Spark Executors write into their non-volatile storage the Diff result's partial data as comma-separated values files. Each line of these files contains the position (index) of a diff occurrence and the corresponding nucleotide letters, one for each sequence that integrated the last Diff. Two versions of the Collection phase were explored: Distributed Write (DW) and Merged Write (MW). In the case of DW, for each Diff performed, every Executor E_k writes its Diff result's partial data into p local files, where $p \leq N_p$ is the number of exclusive partitions (tasks) assigned to E_k. On the other hand, in the case of MW, for each Diff performed, every Executor sends its Diff result's partial data to one Executor designated by Spark, which will merge and write all received data into a single local file.

4.5. Preliminary Experimental Evaluation

In order to evaluate DIFF_1 and DIFF_{opt} implementations, some preliminary experiments were conducted. Nine Amazon EC2 t2.medium virtual instances were used, of which: one was set as Spark Driver (Master) and the remaining eight used as Spark Workers. Each virtual instance had 2 vCPUs, 4 GiB of Memory and 8 GiB Storage (EBS). The Spark Driver instance was launched as on-demand, costing 0.0464 USD per hour, while the Spark Executors were launched using spot price market, each one costing 0.0139 USD per hour. A Spark 3.1.2 standalone cluster in client deploy mode was set up and the following job submit options were defined: 8 Executors (1 per Worker node), 2 GiB Memory per Executor and 16 Cores in total (2 Cores per Executor). The experiments were carried out considering subsets of SARS-CoV-2 nucleotides taken from 540 South America SARS-CoV-2 nucleotide merged sequences, which were obtained from the NCBI-Virus[6] database

[5] `https://spark.apache.org/docs/latest/tuning.html#level-of-parallelism`

[6] `https://www.ncbi.nlm.nih.gov/labs/virus/vssi/`

and split as individual sequences files using the Fasta-Splitter[7] tool, making possible the individual sequences comparisons. For the DIFF$_{opt}$ assessment, a $max_D = 63$ value was set. Each experiment was executed three times. **Table 1** summarizes the average times to execute the Diff and Collection phases, separately, and the overall average execution times and monetary costs, for three rounds of preliminary experiments as discussed next.

Table 1. Diff and Collection phases average times, overall execution times and monetary costs for DIFF$_1$ and DIFF$_{opt}$.

Experiments Round	*N* SARS-CoV-2 Sequences	Average Diff Phase Time (Minutes)		Average Collection Phase Time (Minutes)		Average Execution Time (Minutes)		Average Execution Cost (USD)	
		DIFF$_1$	DIFF$_{opt}$	DIFF$_1$	DIFF$_{opt}$	DIFF$_1$	DIFF$_{opt}$	DIFF$_1$	DIFF$_{opt}$
FR_{PE}	**2**	0.0022	0.0053	0.2815	0.2893	0.4256	0.4362	0.0011	0.0011
	4	0.0045	0.0083	0.9854	0.5825	1.1537	0.7530	0.0030	0.0020
	8	0.0105	0.0196	2.8430	1.1275	3.0950	1.3454	0.0081	0.0035
	16	0.0267	0.0468	8.1126	1.9528	8.6050	2.2651	0.0226	0.0059
	32	0.0762	0.1850	29.6007	3.5592	30.9374	4.2317	0.0813	0.0111
	64	0.3822	0.5770	123.6268	5.6632	130.7519	9.2003	0.3434	0.0242
SR_{PE}	**2**	0.0112	0.0141	0.2579	0.2881	0.4139	0.4731	0.0011	0.0012
	4	0.0200	0.0218	0.8406	0.4648	1.0336	0.6453	0.0027	0.0017
	8	0.0458	0.0419	2.6194	0.9115	2.9148	1.1462	0.0077	0.0030
	16	0.1237	0.1021	6.5475	1.7639	7.2007	2.1544	0.0189	0.0057
	32	0.4009	0.1576	18.9651	2.8872	20.9013	3.5074	0.0549	0.0092
	64	1.4076	0.5242	65.6623	4.9424	72.4692	6.5623	0.1904	0.0172
TR_{PE}	**2**	0.0112	0.0127	0.1836	0.1746	0.3596	0.3291	0.0009	0.0009
	4	0.0204	0.0215	0.5877	0.3152	0.7770	0.4955	0.0020	0.0013
	8	0.0478	0.0455	1.7232	0.5287	2.0287	0.7694	0.0053	0.0020
	16	0.1358	0.0883	5.0213	0.9464	5.7202	1.3004	0.0150	0.0034
	32	0.3989	0.2186	16.1036	1.8633	18.0104	2.5403	0.0473	0.0067
	64	1.4035	0.7150	61.7254	3.4916	68.4508	5.2757	0.1798	0.0139

The first round of preliminary experiments FR_{PE} assesses DIFF$_1$ and DIFF$_{opt}$ when using DataFrames auto partitioning and DW. Their standard deviation values ranged from 0.0002 to 0.21, from 0.01 to 16.78, from 0.0087 to 19.80 and from 0.0001 to 0.05, respectively. As N increases, DIFF$_1$ loses performance in relation to DIFF$_{opt}$, which was approximately 14.2 faster and cheaper than DIFF$_1$ considering 64 SARS-CoV-2 sequences.

A second round of preliminary experiments SR_{PE} analyzes the efficiency regarding a customized DataFrames partitioning scheme, but also using DW. Their standard deviation values ranged from 0.0001 to 0.07, from 0.02 to 4.08, from 0.02 to 3.78 and from 0.0001 to 0.01, respectively. Compared to FR_{PE} results, both DIFF$_1$ and DIFF$_{opt}$ achieved better performance (except for DIFF$_{opt}$ when $N = 2$). Interestingly, for $N = 64$ and SR_{PE}, DIFF$_1$ execution time and monetary cost had an average percentage decrease of 44.5% while DIFF$_{opt}$ reduced around 28.7%.

The third round of preliminary experiments TR_{PE} evaluates the efficiency of MW. Their standard deviation values ranged from 0.0015 to 0.02, from 0.0121 to 0.85, from 0.0080 to 0.85 and from 0.0001 to 0.0022, respectively. A percentage decrease on the collection time (write to disk) was noticed when switching from DW to MW. In the case of 64 SARS-CoV-2 sequences, the decrease in TR_{PE} was approximately 6% for DIFF$_1$ and 29.3% for DIFF$_{opt}$.

Therefore, bearing in mind the results obtained so far, the *Diff Sequences Spark* application proposed in this work employs customized DataFrames partitioning scheme,

[7] https://github.com/alan-lira/fasta-splitter

DIFF$_{opt}$ during the Diff phase and Merged Write (MW) during the Collection phase.

5. Diff Sequences Spark Application on Amazon EC2

Amazon EC2 offers a wide variety of virtual instance types which are optimized suited to different use cases. These instances consist of various combinations of CPU, memory, storage, and network capacity. In EC2, the user can select the following instance types to run an application: (i) general purpose, (ii) compute optimized, (iii) memory optimized, (iv) accelerated computing and (v) storage optimized. As shown previously, the execution time of the *Diff Sequences Spark* application is dominated mainly by: the Diff phase, which executes the tasks predominantly in memory (memory intensive); and the Collection phase that is disk intensive. Thereby, this work executed the *Diff Sequences Spark* application using the instances as seen in **Table 2**. All selected instances are from the us-east-1a region with a x86_64 processor and they are available on the spot market, costing less than 0.50 USD per hour on the on-demand market. The work developed here aims to analyze the benefits of executing *Diff Sequences Spark* application on different types of both spot and on-demand instances. Wherefore, this experimental evaluation also has the objective to indicate some relevant metrics for memory and disk intensive Spark applications for future deeper analysis.

Table 2. EC2 VMs selection: memory and storage optimized instances.

Optimization Family	Instance Name	Number of vCPUs	Memory (GiB)	Storage Type	Network Speed (Gbps)	Cost per Hour (USD) On-Demand	Spot
Memory	r5.xlarge	4	32	EBS	Up to 10	0.2520	0.1374
	r5dn.xlarge	4	32	1x 150 NVMe SSD	Up to 25	0.3340	0.1232
	z1d.xlarge	4	32	1x 150 NVMe SSD	Up to 10	0.3720	0.1116
Storage	i3en.xlarge	4	32	1x 2500 NVMe SSD	Up to 25	0.4520	0.1356
	h1.2xlarge	8	32	1x 2000 HDD	Up to 10	0.4680	0.1404
	d3.xlarge	4	32	3x 2000 HDD	Up to 15	0.4990	0.1497

5.1. Computational Results

In these experiments, one instance was set as Spark Driver (Master) and another eight instances as Spark Workers. The instances features are those in **Table 2**. A Spark 3.1.2 standalone cluster in client deploy mode was set up and the following job submit options were defined for all instances: 8 Executors (1 per Worker node), 29 GiB Memory per Executor and 32 Cores in total (4 Cores per Executor). The experiments were carried out considering the entire 540 South America SARS-CoV-2 nucleotide sequences ($N = 540$) and $max_D = 63$. Each experiment was executed three times.

The first round of the main experiments FR_{ME} compared the execution times and monetary costs considering all the virtual instance types. **Table 3** summarizes the average (Avg) execution times and monetary costs obtained when launching Workers using spot instances. Their standard deviation (SD) values ranged from 0.72 to 5.76 and from 0.02 to 0.15, respectively. Furthermore, r5.xlarge instance results were used as baseline to calculate the relative percentage changes obtained from the other instances' execution. **Table 4** shows the average (Avg) execution monetary costs and the cost percentage changes when switching all Spark Workers from on-demand to spot instances. Their standard deviation (SD) values ranged from 0.05 to 0.39 and from 0.02 to 0.15, respectively. As can be seen in **Table 3**, z1d.xlarge, a memory optimized instance, produced the best results regarding both execution time and financial cost, while h1.2xlarge produced the worst ones.

Table 3. FR_{ME}: Averages of execution times and costs.

Instance Name	Execution Time (Minutes) Avg	SD	Execution Cost (USD) Avg	SD	Percentage Change Time	Cost
r5.xlarge	174.6113	3.0408	3.9322	0.0685	0%	0%
r5dn.xlarge	174.3515	1.0521	3.8346	0.0231	−0.1487%	−2.4820%
z1d.xlarge	135.1479	1.9064	2.8489	0.0402	−22.6007%	−27.5494%
i3en.xlarge	169.0511	5.7611	4.3300	0.1476	−3.1843%	+10.1164%
h1.2xlarge	201.6883	1.1435	5.3488	0.0303	+15.5070%	+36.0256%
d3.xlarge	165.1419	0.7194	4.6697	0.0203	−5.4231%	+18.7554%

Table 4. FR_{ME}: Averages of on-demand and spot execution costs.

Instance Name	On-Demand Execution Cost (USD) Avg	SD	Spot Execution Cost (USD) Avg	SD	% Cost Change
r5.xlarge	6.6003	0.1149	3.9322	0.0685	−40.4239%
r5dn.xlarge	8.7350	0.0527	3.8346	0.0231	−56.1007%
z1d.xlarge	7.5413	0.1064	2.8489	0.0402	−62.2226%
i3en.xlarge	11.4617	0.3906	4.3300	0.1476	−62.2220%
h1.2xlarge	14.1585	0.0803	5.3488	0.0303	−62.2219%
d3.xlarge	12.3609	0.0538	4.6697	0.0203	−62.2220%

The second round of main experiments SR_{ME} aims to present the impact of Spark Workers spot instances revocations over execution times and monetary costs of the two best instances, one for each optimization family, in terms of cost-benefit that were obtained of FR_{ME}: z1d.xlarge and i3en.xlarge. Their respective previous results (with no instances revocation) were used as baseline for the following spot instances revocation scenarios comparisons: RS_1: two Spark Workers revoked after 30 minutes of execution; RS_2: two Spark Workers revoked after 60 minutes of execution; RS_3: two Spark Workers revoked after 120 minutes of execution time. **Table 5** summarizes the average (Avg) execution times and monetary costs considering these scenarios. Their standard deviation (SD) values ranged from 1.48 to 5.76 and from 0.03 to 0.15, respectively. The time and cost percentage changes are also presented for each instance, considering their respective baseline values. As it can be noticed, the fewer Workers available, the longer *Diff Sequences Spark* application run. The RS_1 revocation scenario had the biggest increase of average execution times for both z1d.xlarge and i3en.xlarge instances, 4.19% and 8.67% respectively, meaning that the application performed better when there were more Workers executing the first hundreds DataFrames comparisons, as they were typically larger. On the other hand, Workers revocations on RS_3 (near the end of execution) did not cause a big increase of average execution time for i3en.xlarge instance (4.07%), which can be explained by smaller DataFrames comparisons. Interestingly, z1d.xlarge obtained 1.64% average execution time reductions for RS_3, possibly because performed more Diffs during the first hour of execution (1114 Diffs on average) compared to the no-revocation scenario (1084 Diffs on average), somewhat compensating the Workers revocation near the end of execution. Regarding execution costs, all revocation scenarios caused cost reductions. RS_1 had the greatest reductions: 10.28% and 23.77% for z1d.xlarge and i3en.xlarge instances, respectively. The relatively small size of SARS-CoV-2 nucleotide sequences may be the reason for this cost savings when losing Workers, as, in the no-revocation scenario, Spark must have spent more time creating, scaling and managing tasks than the Workers spent

doing the task computation itself. Knowing that RS_1 is the worst revocation scenario, nevertheless using the spot instances allowed 66.11% and 71.20% average execution cost savings for z1d.xlarge and i3en.xlarge spot instances, respectively, while their respective average execution times increased only 4.19% and 8.67%, in comparison with their on-demand instances prices market. Finally, memory optimized instances presented better execution times and financial costs than the storage optimized ones, and, particularly, the z1.xlarge instance outperformed all the other selected instances.

Table 5. SR_{ME}: Average execution and financial costs in several Worker spot instances revocations scenarios.

Instance Name	Revocation Scenario	Execution Time (Minutes) Avg	Execution Time (Minutes) SD	Execution Cost (USD) Avg	Execution Cost (USD) SD	Percentage Change Time	Percentage Change Cost
z1d.xlarge	None	135.1479	1.9064	2.8489	0.0402	0%	0%
	RS_1	140.8041	1.5435	2.5560	0.0268	+4.1851%	−10.2811%
	RS_2	139.2890	1.6629	2.6413	0.0289	+3.0641%	−7.2870%
	RS_3	132.9266	5.0956	2.7540	0.0885	−1.6436%	−3.3311%
i3en.xlarge	None	169.0511	5.7611	4.3300	0.1476	0%	0%
	RS_1	183.7072	2.0837	3.3008	0.0362	+8.6696%	−23.7690%
	RS_2	177.4600	1.4847	3.3039	0.0258	+4.9741%	−23.6974%
	RS_3	175.9235	2.9039	3.5004	0.0504	+4.0652%	−19.1593%

6. Conclusions and Future Directions

Aiming to analyze the execution of Spark applications with reasonable time and cost in a public cloud, this work proposed the *Diff Sequences Spark* application. When using memory and storage optimized spot instances without any revocations, reductions of up to 22.60% of the average execution time and up to 62.22% of the average monetary cost were observed compared to their respective on-demand instances for the study case of 540 SARS-CoV-2 all-against-all sequences comparison. Nonetheless, on the worst tested spot revocation scenario, it was possible to reduce the monetary cost in 10.28% and 23.77% respectively for the z1d.xlarge and i3en.xlarge instances, while their respective execution times slightly increased to 4.19% and 8.67%, benefiting from the low overhead fault tolerance Spark framework. The experiments also showed cost-benefit of running it on the memory optimized instances, most outstanding being z1d.xlarge. As future directions, considerations on data input variations (*i.e.*, size and quantity of biological sequences), worker nodes and executors per worker, as well as VMs resources metrics (*e.g.*, IOPS, bandwidth) will allow better analysis of the performance and of the scaling bounds for Spark applications in the cloud.

References

[Armbrust et al. 2016] Armbrust, M., Bateman, D., Xin, R., and Zaharia, M. (2016). Introduction to spark 2.0 for database researchers. In *SIGMOD '16*, page 2193–2194.

[Brum et al. 2021] Brum, R., Sousa, W., Melo, A., Bentes, C., Castro, M. C., and Drummond, L. (2021). A fault tolerant and deadline constrained sequence alignment application on cloud-based spot GPU. In *27th EuroPar Conference, to appear*.

[de Oliveira et al. 2021] de Oliveira, D., Porto, F., Boeres, C., and de Oliveira, D. (2021). Towards optimizing the execution of spark scientific workflows using machine learning-based parameter tuning. *CCPE*, 33(5):e5972.

[Durbin et al. 1998] Durbin, R., Eddy, S., Krogh, A., and G., M. (1998). *Biological sequence analysis*. Cambridge University Press.

[Hey and Trefethen 2020] Hey, T. and Trefethen, A. E. (2020). The fourth paradigm 10 years on. *Inform. Spektrum*, 42(6):441–447.

[Hindman 2011] Hindman, B. *et. al*. (2011). Mesos: A platform for fine-grained resource sharing in the data center. In *Proc.s of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 295–308, USA. USENIX.

[Hu et al. 2014] Hu, H., Wen, Y., Chua, T.-S., and Li, X. (2014). Toward scalable systems for big data analytics: A technology tutorial. *IEEE Access*, 2:652–687.

[Lau et al. 2021] Lau, B. T., Pavlichin, D., and Hooker, A. C. e. a. (2021). Profiling sars-cov-2 mutation fingerprints that range from the viral pangenome to individual infection quasispecies. *Genome Medicine*, 13:28:1–28:23.

[Perera et al. 2016] Perera, S., Perera, A., and Hakimzadeh, K. (2016). Reproducible experiments for comparing apache flink and apache spark on public clouds.

[Rochman et al. 2021] Rochman, N. D., Wolf, Y. I., Faure, G., Mutz, P., Zhang, F., and Koonin, E. (2021). Ongoing global and regional adaptive evolution of sars-cov-2. *Proceedings of the National Academy of Sciences*, 118(29).

[Teylo et al. 2021] Teylo, L., Arantes, L., Sens, P., and Drummond, L. M. (2021). A dynamic task scheduler tolerant to multiple hibernations in cloud environments. *Cluster Computing*, 24(2):1051–1073.

[Xu et al. 2017a] Xu, B., Li, C., Zhuang, H., Wang, J., Wang, Q., Zhou, J., and Zhou, X. (2017a). Dsa: Scalable distributed sequence alignment system using simd instructions. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 758–761, Los Alamitos, CA, USA. IEEE Computer Society.

[Xu et al. 2017b] Xu, B., Li, C., Zhuang, H., Wang, J., Wang, Q., and Zhou, X. (2017b). Efficient distributed smith-waterman algorithm based on apache spark. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 608–615.

[Xu et al. 2019] Xu, F., Zheng, H., Jiang, H., Shao, W., Liu, H., and Zhou, Z. (2019). Cost-effective cloud server provisioning for predictable performance of big data analytics. *IEEE Transactions on Parallel and Distributed Systems*, 30(5):1036–1051.

[Yan et al. 2016] Yan, Y., Gao, Y., Chen, Y., Guo, Z., Chen, B., and Moscibroda, T. (2016). Tr-spark: Transient computing for big data analytics. In *SoCC*, page 484–496.

[Zaharia et al. 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *USENIX*, 1:1–14.

[Zaharia et al. 2010] Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(1-7):95.

[Zhao et al. 2015] Zhao, G., Ling, C., and Sun, D. (2015). Sparksw: Scalable distributed computing system for large-scale biological sequence alignment. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 845–852.