A Framework for Executing Protein Sequence Alignment in Cloud Computing Services

Leonardo Reboucas de Carvalho[1], **Alba Cristina Alves Melo**[1], **Aleteia Araujo**[1]

[1]Campus Darcy Ribeiro, Department of Computer Science – University of Brasília
Brasília, Brazil.

leouesb@gmail.com, {alves,aleteia}@unb.br

Abstract. *Protein sequence alignment is a task of great relevance in Bioinformatics and the Hirschberg algorithm is widely used for this task. This work proposes a framework for executing sequence alignment with the Hirschberg algorithm in different cloud computing services. In experiments, our framework was used to align HIV-1 protease sequences using different instances of AWS EC2 and different configurations of AWS Lambda functions.The results show that, for this application, there is a tradeoff between the expected execution time and the cost, e.g., in most cases AWS Lambda provides the best runtime, however at a higher USD cost. In this context, it is important to have a framework that helps in deciding which approach is most appropriate.*

1. Introduction

Studies in Bioinformatics generally produce computational challenges that require large volumes of computing resources. An example of this type of challenge is protein alignment. In this task, two strings will be compared with the purpose of identifying their similarities and differences. With a focus on differences, researchers can make great discoveries about the protein in question. In this context cloud computing can collaborate with Bioinformatics.

The potential requirement for large volumes of computational resources that a sequence alignment application has signals that its processing is suitable to be sent to a cloud computing environment, since this type of environment offers the possibility of fast and dynamic allocation of resources [MELL and Grance 2011]. Cloud computing currently offers several service models and each feature of these models can result in benefits or losses, whether in runtime, cost or other aspects. It is therefore important to choose the type of cloud service best suited to the challenge to be faced.

In this paper, we propose a framework for the provision and execution of a sequence alignment application composed of multiple tasks in different service models. In the provision phase, our framework decides which model (AWS EC2 on demand or AWS Lambda) is more appropriate for a particular sequence alignment application, considering two goals (cost and execution time). The alignments are then executed concurrently in the chosen cloud model and, at the end of the execution, the results are returned to the user. Finally, the unprovision phase terminates the cloud environment.

The remainder of this paper is organized as follows. Section 2 presents the background of this work, including the fundamentals of sequence alignment and cloud computing. Section 3 describes the proposal of this work, while Section 4 discusses related

works. In Section 5 the methodology used in the experiment is explained, and in Section 6 the results obtained are detailed. Finally, in Section 7, conclusions are presented.

2. Background

The algorithms based on Needleman and Wunsch [Needleman and Wunsch 1970] and Smith and Waterman [Smith et al. 1981] seek to solve the alignment of two sequences by finding the maximum cost of transforming one sequence into the other. These algorithms compute a dynamic programming (DP) matrix of size m x n, where m and n are the respective widths of the sequences. Data dependencies must be respected, in such a way that position (i, j) depends on $(i-1, j)$, $(i, j-1)$ and $(i-1, j-1)$. Both algorithms have quadratic memory and time complexities $O(mn)$. As the size of the sequences involved in the alignment grows, so does the space required for processing the alignment, and this can potentially become a resource scarcity problem. For this reason, the use of an alignment execution approach oriented to cloud computing is adequate, since the continuous and dynamic resource allocation capability is able to overcome potential scarcity problems.

2.1. The Hirschberg Algorithm

The Hirschberg algorithm [Hirschberg 1975] consists of a strategy to calculate the similarity between two sequences using the concept of longest common subsequence (LCS). It uses a divide-and-conquer iterative approach that obtains the alignment recursively, with linear memory complexity $O(m)$ and quadratic time complexity $O(mn)$. As a result, the entire matrix is not stored, but only a few rows of it, and the algorithm recalculates portions of the matrix in each iteration.

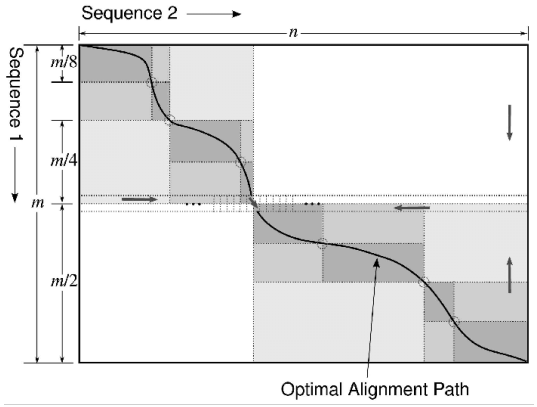

Figure 1. Hirschberg's space-saving scheme [Sarje and Aluru 2009].

Fig. 1 shows some iterations of Hirshberg's algorithm. First, the matrix is processed from the beginning to the middle, row by row, storing only two rows: the one that is being calculated and the previous one. The middle row ($middle_{orig}$) is saved. Then, the matrix is processed from the end to the middle, over the reverses of the sequences, up to the middle row calculation ($middle_{rev}$). At this point, there are two middle rows, computed differently. Hirschberg proved that the point i, where the addition of $middle_{orig}(i)$ and $middle_{rev}(i)$ is maximal, belongs to the optimal alignment. This point is used to split the matrix in four parts and the same approach is applied to the upper left and bottom right parts (light gray) in the second iteration. In the figure, we can see the third (gray)

and forth (dark gray) iterations of the algorithm. The execution finishes when all the points that belong to the optimal aligment are retrieved.

2.2. Cloud Computing Service Models

Among the various models of cloud services available on the market today, Function-as-a-Service [Lynn et al. 2017] stands out, a service model in which the client sends the source code that he/she wants to be processed and the provider will be carry out the processing, triggered from its own platform, or through a call to an API. In addition, the provider takes care of the automatic elasticity of the infrastructure.

Fig. 2 was designed for this paper and shows the cloud user duties on IaaS and FaaS cloud services. As can be seen, in IaaS the customer has a greater responsibility in managing the service, especially if there is a need for elastic performance. On the other hand, in FaaS, the user is relieved of the tasks of configuring the environment, including aspects of elasticity. Furthermore, in FaaS, the deployment process is replaced by the publication of the function's source code on the provider, that will then be responsible for carrying out the deployment when necessary, in a transparent way to the user.

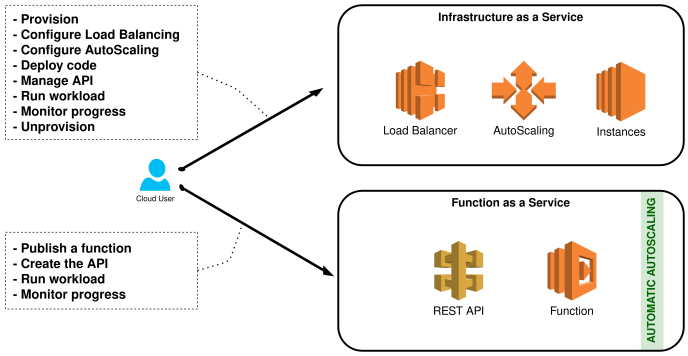

Figure 2. IaaS x FaaS cloud user duties.

Another important difference between IaaS and FaaS is the suppression of the need to unprovision the environment in FaaS when the environment is not being used, since unlike IaaS, in FaaS the charge occurs only when the execution of the function is required, while in IaaS the charging occurs until the instance is terminated.

The charge on FaaS model is based on the requests made and the runtime that each request takes. Thus, the provider charges the customer only for the effective runtime, different from the IaaS model, in which the provider generally charges for the time the machine is in operation, regardless of its actual use in the customer's business process.

The current leaders in the cloud market are Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform (GCP) [GARTNER 2021]. Founded in 2006, AWS tops this list and since the beginning of its operations the provider has been driving the cloud market. AWS offers over 200 cloud services, among them EC2 and Lambda.

AWS EC2 "is a web service that provides secure, resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers" [Amazon 2021]. EC2 allows to provision around 400 types of cloud virtual machine instances spread over 77 availability zones in 24 different regions of the planet. AWS charges per hour of use or per second for instances running a Linux Operating System.

AWS Lambda "is a serverless compute service that lets you run code without provisioning or managing servers, creating workload-aware cluster scaling logic, maintaining event integrations, or managing runtimes" [Amazon 2021]. Lambda functions can be written in many programming languages. It is possible to set up to automatically trigger it from 140 AWS services or call it directly from web applications. AWS allows the creation of Lambda functions with up to 10 GB of memory. Lambda allocates CPU and other resources linearly in proportion to the amount of memory configured [Poccia 2020], i. e., AWS grants access to up to 6 vCPUs in each execution environment of Lambda. AWS charges a fixed amount for each request made to a Lambda function. In addition, it also charges a value related to the function's execution time multiplied by the amount of allocated memory [Amazon 2021].

3. Framework Proposal

Considering the differences in characteristics, purposes and degrees of complexity of use that can be found between the various models of cloud computing, in particular the IaaS and FaaS services models, such as AWS EC2 and AWS Lambda, this paper proposes a tool to make the decision-making process on the adoption of one of these models pragmatic, simplified and automated for the execution of genetic protein sequence alignments. In this sense, the proposed flow shown in Fig. 3 is divided into 5 phases:

- **Input:** to initialize the flow of the framework, it is necessary to provide some inputs, such as the provider's credentials in order to obtain access to the cloud account, as well as a file in FASTA format containing the strings that must pairwise aligned, and a reference parameter to guide the decision flow in relation to the expected objective, that is, cost reduction or reduction of the execution time;
- **Provision:** the framework has a decision-making process that will analyze the information obtained in the previous phase and decide which type of service is most suitable for the informed workload. In this initial version EC2 and Lambda are considered and the size of the sequences must be up to 10k. Once the type of cloud service is defined, then the environment will be properly provisioned using an architectural definition component of the orchestrator and Terraform [HashiCorp 2021], acting as a cloud orchestrator, after provisioning the applica-

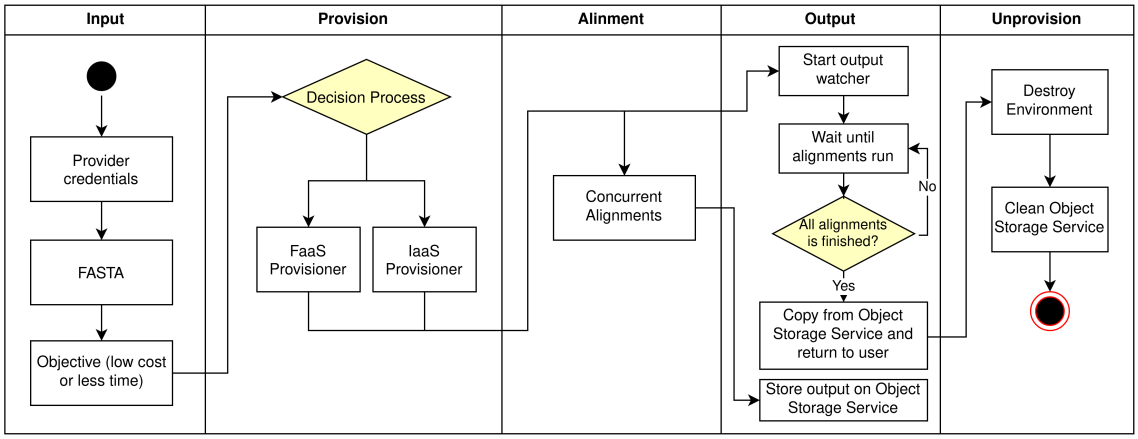

Figure 3. Cloud model agnostic genetic sequence aligner framework workflow.

tion will be deployed, in the case of EC2 or function, in the case of Lambda, which will perform the alignment of the sequences;

- **Alignment:** once the environment is provisioned, then there will be a URL waiting to be triggered to receive strings to be aligned. Thus, the framework will command parallel alignments according to the parameters defined by the decision process in order to meet the objective defined in the input phase;
- **Output:** the output of the alignments are not stored within the provisioned environments, because in the case of FaaS this environment is not permanent, that is, the provisioning process only defines the function that the provider will effectively provision only at the time of the request. Therefore, considering the characteristic of being ephemeral of FaaS, the framework stores the result of alignments in S3, which is an AWS object storage service. Thus, the framework will be monitoring S3 until all the alignments are finished, to then load them and deliver them to the client;
- **Unprovision:** the last phase of the framework consists of unprovisioning the environment using the reverse process defined in the architectural artifact for each environment. Terraform is also used for this process, this time using its "destroy" process. Finally, the S3 bucket that then received the results of the alignments is removed. This process prevents unnecessary charges from being applied to the user's bill, rationalizing the application of financial resources.

The five phases that make up this proposal allow users to perform genetic sequence alignments using the Hirschberg algorithm in the cloud service models that best suit their workload between EC2 and Lambda and meet the given objective (less financial cost or less time). All of this is done in a transparent, automatic and simplified way. For this, the user provides only the minimum information for the execution, eliminating the need to configure and manage the infrastructure.

4. Related Works

A proof-of-concept case study in which 20,000 protein sequences were aligned using the Smith Waterman algorithm in the AWS Lambda and Google Cloud Platform providers proposed in [Niu et al. 2019]. This case study shows the potential for leveraging serverless computing resources for biomedical research in terms of ease of use, instant scalability and cost effectiveness. However, it does not explore competitive situations and does not consider service models besides FaaS, such as IaaS.

In [Crespo-Cepeda et al. 2019] the authors performed the execution of Cloud-DmetMiner, a Bioinformatics application created as a cloud version of the DMET-Miner algorithm. The use of the serverless cloud computing model in AWS Lambda has helped them to ease the execution of code without having to plan resource provisioning and management. However, they only analyzed input parameters and execution time. They did not analyzed costs and also did not perform tests using service models other than FaaS.

The paper [Hung et al. 2020] describes a RNA-seq workflow using Unique Molecular Identifiers (UMI) to obtain deduplicated transcription counts that are then processed to obtain a list of genes that have been differentially expressed after treatment with different combinations of drugs. These readings are then aligned to the human transcriptome using the Burrows-Wheeler Aligner (BWA). The resulting alignments are merged

and deduplicated to calculate the counts for each transcript. The authors processed this workload in an AWS EC2 instance, Lambda Functions and Google Cloud Functions and they observed a great reduction in the execution time in FaaS services. However, they did not present cost analysis for the tests performed.

The study [Malla and Christensen 2020] performed an evaluation of a workload with an all-against-all pairwise comparison of human proteins using a dynamic programming algorithm. The dataset consists of 20336 human protein sequences split into 41 files (each with about 500 proteins). Pairwise comparison of these 41 files, resulted in 861 tasks in total. They compared the workload execution time on Google Compute Engine (IaaS) and Google Cloud Functions (FaaS), both storing the results on Google Cloud Storage. However, they did not present cost analysis for the tests performed.

None of the described articles proposes a framework for performing sequence alignments in various cloud service models, as proposed in this work. Nor do they present performance studies involving FaaS and IaaS services that also analyze costs as in this study, although this analysis is essential to establish a cost-benefit relationship between these service models.

5. Methodology

Considering the differences between the IaaS and FaaS models in relation to the way of delivering computational processing and that each format submits the protein sequence alignment algorithms to different computability conditions, in this work an experiment was carried out to evaluate the behavior of the Hirschberg algorithm on different competitive situations in two AWS services: EC2 (IaaS) and Lambda (FaaS).

5.1. Experiment Architecture

Fig. 4 shows the architecture used in the experiment. It is possible to observe that the experiment was orchestrated from a local environment using a shell script. This script iteratively triggered the Lambda instances, or functions, involved in the experiment, each submitting the respective loads of bids according to the test case.

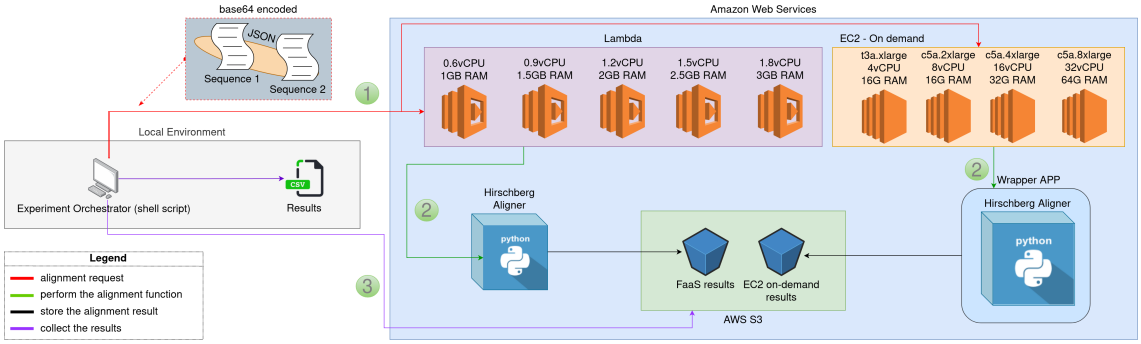

Figure 4. Experiment Architecture.

Each Lambda function had a different configuration for RAM, but it ran the same source code with the respective Hirshberg alignment algorithm[1]. As in AWS Lambda the

[1]The implementation used the algorithm developed by Gang Li at https://github.com/leebird/alignment

amount of RAM memory defines the number of vCPUs allocated in each execution of the function, so the values of vCPU are linearly linked to the value defined for RAM. The Lambda functions involved in the experiments are listed in Table 1, as well as the amount charged for an eventual execution of this function for a period of one hour in the US East (N. Virginia) region.

Table 1. Services involved in the test and their prices recorded on April 5, 2021.

Service	Name	Memory	vCPU	Price / h ($)
Lambda	Configuration A	1	0.6	0.06072032
Lambda	Configuration B	1.5	0.9	0.09072038
Lambda	Configuration C	2	1.2	0.12072044
Lambda	Configuration D	2.5	1.5	0.15072050
Lambda	Configuration E	3	1.8	0.18072056
EC2	c5a.2xlarge	16	8	0.38000000
EC2	c5a.4xlarge	32	16	0.76000000
EC2	c5a.8xlarge	64	32	1.52000000
EC2	t3a.xlarge	16	4	0.17860000

The EC2 instances involved in the experiment were selected using the AWS provider instance search tool on their console. The following parameters were used as selection criteria: x86_64 architecture; minimum 4 vCPUs and maximum 32 vCPUs. Next, those that offered the lowest on-demand prices for the Linux per hour and had 4, 8, 16 and 32 vCPUs were selected. Thus, the instances selected for the experiments are listed in Table 1, as well as their respective prices announced by the provider for an hour of execution in the US East (N. Virginia) region.

In order to be triggered, the Lambda functions were configured to interact with the AWS API Gateway service that exposes a REST API and delivers a URL address through which it is possible to forward POST requests to a Lambda function. To act in a similar way with the AWS API Gateway, in each instance a layer of REST API was configured to receive the requests and forward them to the Hirshberg alignment algorithm. This layer was created using the FastAPI library in Python, which is the same language used in the Lambda's sequence alignment algorithm and in the instances application.

5.2. Experiment Workflow

Test cases were generated from the sequences selected for the experiment. These cases were assembled by joining two sequences drawn at random within the group of sequences. Lots of test cases were then generated with quantities related to the competition that the test intended to submit in the environments, that is, for a competition test of 20 simultaneous alignment requests, 20 test cases were generated and these same cases were used in all environments in order to guarantee a level playing field during the experiment. For this experiment, lots were generated that allowed the execution of competition tests with 1, 20, 40, 60, 80 and 100 simultaneous alignments.

Once the batches of test cases were generated, as can be seen in Fig. 5 which shows the workflow of the experiment execution, for each type of environment (FaaS and EC2) there was a different flow.

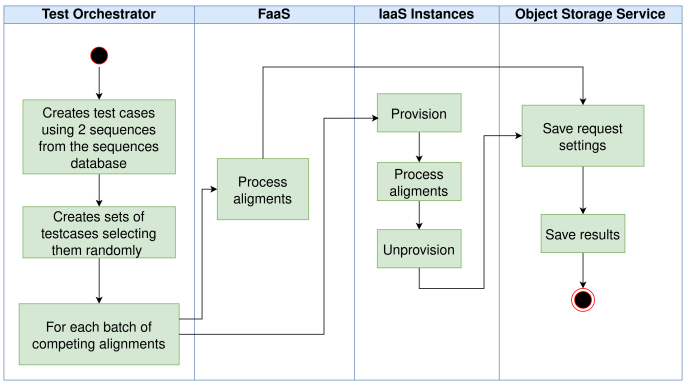

Figure 5. Experiment Workflow.

For FaaS, the alignment occurred directly from a POST request to the REST API created for the Lambda function. At the end of the alignment execution, the function itself saved the result in the AWS S3 storage service. For EC2, there was a step before the request for alignment, which was the provisioning of the environment. This step was also automated during the tests using Terraform, as well as the step after the alignment, which is the unprovisioning of the environment. With each test run, the environment was unprovisioned, even if the tests were to take place on an instance of the same type. This ensures that the instances would not eventually be accumulating remnants from the previous tests that could interfere with the results of the subsequent tests. Both process saved the results in the AWS S3 for later collection and analysis.

5.3. Experiment Dataset

To exercise the workloads of the experiment, sequences of random strings with a width of around 10,000 characters would suffice. This limit was obtained from tests performed with the Hirshberg Aligner running in the best Lambda configuration under the constraints of runtime and number of allocable resources imposed by the provider. However, for this study, it was adopted a real dataset. From a search in the Stanford University's database of HIV-resistance drugs, 112 complete sequences were found, with a length of around 10,000 characters.

The source code of the Lambda function that was used in this experiment, as well as the source code of the application that was configured for the EC2 instances that took part in this experiment were published on GitHub[2]. In a separate project, the description of the sequences used in the experiment and scripts for orchestrating the experiment were also published on GitHub[3], as well as their secondary tools responsible for generating the test batches and the results obtained in the execution of the test cases.

6. Results

After the tests were performed, which took place between April 4th and 6th, 2021, the results generated in the AWS S3 service were collected. Considering the start of the first alignment and the end of the last alignment of each test case, it was possible to calculate

[2]https://github.com/unb-faas/sequence_comparison_app
[3]https://github.com/unb-faas/sequence_comparison

Table 2. Average duration results.

Service	Concurrence (simultaneous alignments)					
	1	20	40	60	80	100
Lambda (1.0GB / 0.6vCPU per request)	00:08:31	00:09:31	00:14:46	00:09:15	00:09:24	00:09:29
Lambda (1.5GB / 0.9vCPU per request)	00:05:53	00:06:09	00:06:23	00:07:04	00:06:26	00:06:32
Lambda (2.0GB / 1.2vCPU per request)	00:05:03	00:05:19	00:05:37	00:05:27	00:05:19	00:05:26
Lambda (2.5GB / 1.5vCPU per request)	00:05:02	00:05:34	00:05:21	00:05:12	00:05:23	00:05:42
Lambda (3.0GB / 1.8vCPU per request)	00:04:55	00:05:16	00:05:13	00:05:15	00:05:33	00:05:21
T3a.xlarge (16GB / 4vCPU)	00:05:52	00:46:30	01:29:32	02:15:27	03:20:19	03:50:57
C5a.2xlarge (16GB / 8vCPU)	00:03:49	00:17:05	00:34:11	00:50:50	01:06:47	01:20:53
C5a.4xlarge (32GB / 16vCPU)	00:03:46	00:10:04	00:17:19	00:25:45	00:35:08	00:42:48
C5a.8xlarge (64GB / 32vCPU)	00:03:53	00:05:47	00:10:29	00:12:58	00:17:54	00:21:34

the average times of each test as shown in Table 2. The standard deviation was less than 0.1% in all cases. This demonstrates the existence of a low dispersion of data, that is, it is possible to consider the average of these durations as information that adequately represents the universe of data generated during the tests.

Table 3. Effective cost ($) results.

Service	Concurrence (simultaneous alignments)						Sum per service
	1	20	40	60	80	100	
Lambda (1.0GB / 0.6vCPU per request)	0.0087	0.1802	0.3639	0.5304	0.7127	0.8892	2.6851
Lambda (1.5GB / 0.9vCPU per request)	0.0090	0.1753	0.3546	0.5273	0.7051	0.8865	2.6577
Lambda (2.0GB / 1.2vCPU per request)	0.0102	0.2031	0.4062	0.5991	0.8042	0.9985	3.0212
Lambda (2.5GB / 1.5vCPU per request)	0.0127	0.2507	0.5014	0.7421	0.9995	1.2494	3.7558
Lambda (3.0GB / 1.8vCPU per request)	0.0149	0.3004	0.5967	0.8860	1.1935	1.4868	4.4783
T3a.xlarge (16GB / 4vCPU)	0.0175	0.1384	0.2665	0.4032	0.5963	0.6875	2.1093
C5a.2xlarge (16GB / 8vCPU)	0.0242	0.1082	0.2165	0.3219	0.4230	0.5123	1.6060
C5a.4xlarge (32GB / 16vCPU)	0.0477	0.1275	0.2193	0.3262	0.4450	0.5421	1.7079
C5a.8xlarge (64GB / 32vCPU)	0.0984	0.1465	0.2656	0.3285	0.4535	0.5464	1.8388
Sum per test-case	**0.2432**	**1.6303**	**3.1907**	**4.6647**	**6.3327**	**7.7985**	**23.8601**

The analysis of the duration of the tests shows, in general, the alignments executing in the Lambda functions executed in a shorter average time in relation to the alignment they performed in EC2 instances. As can be seen in Fig. 6 (a), where the blue lines represent the Lambda services and the green ones represent the EC2 instances. The execution times of lambda functions remained at close levels under all concurrency levels, except for level 40, where there was an elevation for the function with the lowest resource availability setting. It is likely that during the execution of these test batteries, the cluster that supports the FaaS in the provider suffered some momentary overload that impacted the service, either due to network traffic or even processing bottlenecks.

The EC2 instances established an upward trend in runtime as competition has increased. It is possible to notice in Fig. 6 (a) a highlight for the instance t3a.xlarge, which showed a sharp increase in the execution time during the tests. Such an instance has only 4 vCPUs and the workload is CPU bound, the processing overhead naturally

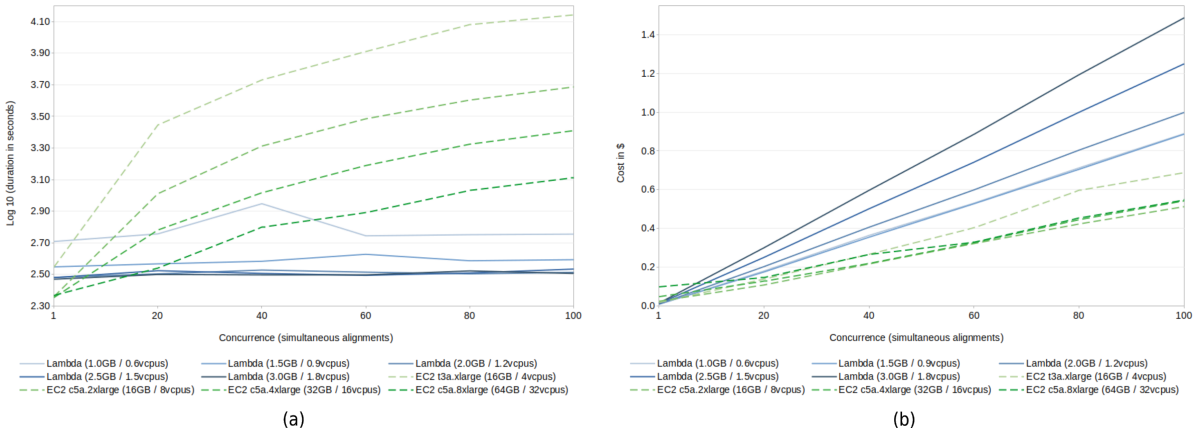

Figure 6. Experiment results: (a) duration and (b) costs.

caused a queuing that consequently impacted the execution time.

Considering the execution time of each test and the cost of each service, it was possible to calculate Table 3 with the costs of the tests. The cost calculation is done differently between EC2 instances and Lambda configurations. For EC2 AWS charges different amounts depending on the flavor of the instance, the mode of hire, which can be dedicated or on-demand, and the operating system. For instances running Linux the charge is made every second. Thus, for the experiment performed, the calculation of each test case is expressed by Equation 1, where the calculated cost for the different competition situations (TC_{EC2}) is obtained by multiplying the execution time in seconds by the amount charged for the flavor of the on-demand instance running Linux.

$$TC_{EC2} = Runtime * FlavorPrice_{Instance} \tag{1}$$

For Lambda, AWS practices a charging policy based on two aspects: the execution time per gigabyte (GB) in seconds plus a fixed amount charged per service request, as can be seen in Equation 2. Considering that in the experiment performed, configurations with different amounts of memory were used, from 1GB up to 3GB, then the calculation of the cost for the test cases using Lambda (TC_{Lambda}) is shown in Equation 3. In this calculation, the execution time is obtained from the average of the execution times of the alignments, since they occur in parallel. Furthermore, the cost is multiplied by the respective concurrency used in the test case, since for alignments requested in Lambda it entails an effective execution of the function.

$$Cost_{Lambda} = Price_{GB/s} * Memory_{GB} + Price_{Request} \tag{2}$$

$$TC_{Lambda} = Runtime_{avg} * Cost_{Lambda} * Concurrence \tag{3}$$

The results expressed in Table 3 brought a new perspective to the test results, although the Lambda functions performed the tests more quickly. Since their charging occurs by request and by the time of execution, the cost of tests using the Lambda func-

tions was in general much higher than those on the EC2 instances, whose pricing occurs only for the time that the instance remains on.

As can be seen in Fig. 6 (b), only when running without competition do Lambda functions have a lower cost than EC2 instances. After 20 simultaneous executions, the cost of the instances remains lower than that determined by the Lambda functions. These results shows that using a dataset composed of HIV sequences with widths of about 10k characters and subjecting the services to competitive situations that varied from 1 to 100 simultaneous alignments, it is possible to notice the existence of a trade-off between the expected execution time and the tolerable cost. In other words, running this type of workload using FaaS can lead to faster results, but with higher costs. On the other hand, if time is not a critical factor, it is more cost-effective to run these workloads using an IaaS instance-oriented approach.

Another perceived fact was due to the automatic elasticity service offered in conjunction with the AWS Lambda. In this service, as requests are received by the environment, more resources are allocated dynamically so that the performance of the environment remains uniform regardless of the competition to which it is submitted. This differs from the behavior of a traditional IaaS environment, such as EC2, in which elasticity needs to be actively controlled by the customer.

The results described above allow us to infer that although FaaS services initially present more attractive prices, it is important to consider that each request made to the service will be charged according with their configuration (memory, CPU , etc) and the time it spend. Cumulatively, the amount for processing workloads such as the one presented in this work may exceed the fixed amount that is charged for IaaS services. Therefore, having a framework available that makes the decision process about which cloud service model is the most suitable for genetic sequence alignment processing can represent the difference between being able to make a major biological discovery within feasible time and cost, or then fail disastrously.

7. Conclusion

In this work, a framework is proposed to simplify the execution of genetic sequence alignments in multiple cloud computing services. An experiment was carried out to perform protein sequence alignments using the Hirshberg algorithm on AWS EC2 instances of different configurations with varied numbers of allocated vCPUs, and on AWS Lambda functions with different memory configurations that resulted in vCPU allocations from 0.6 to up to 1.8 per run.

Thus, it is possible to conclude that the execution of CPU bound workloads that hold Lambda functions for considerable periods can generate high costs in relation to the same workload executed in EC2 instances. On the other hand, in situations of growing competition, environments oriented to Lambda functions will keep their execution times uniform and more predictable, while environments using EC2 instances will suffer degradation in the execution time.

In future work it is possible to extend the types of cloud services supported by the framework, as well as the providers, such as GCP and Azure, for example. Other algorithms can also compose the decision process in order to establish alignment strategies that favor not only low memory consumption, but execution time as well.

References

[Amazon 2021] Amazon (2021). Amazon web services. https://aws.amazon.com/about-aws/. [Online; accessed 21-April-2021].

[Crespo-Cepeda et al. 2019] Crespo-Cepeda, R., Agapito, G., Vazquez-Poletti, J. L., and Cannataro, M. (2019). Challenges and opportunities of amazon serverless lambda services in bioinformatics. BCB '19, page 663–668, New York, NY, USA. Association for Computing Machinery.

[GARTNER 2021] GARTNER (2021). Magic quadrant for cloud infrastructure and platform services. https://www.gartner.com/en/documents/3989743/magic-quadrant-for-cloud-infrastructure-and-platform-ser. [Online; accessed 28-August-2021].

[HashiCorp 2021] HashiCorp (2021). Terraform: Write, plan, apply. https://www.terraform.io/. [Online; accessed 31-May-2021].

[Hirschberg 1975] Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. Commun. ACM, 18(6):341–343.

[Hung et al. 2020] Hung, L.-H., Niu, X., Lloyd, W., and Yeung, K. Y. (2020). Accessible and interactive RNA sequencing analysis using serverless computing. bioRxiv.

[Lynn et al. 2017] Lynn, T., Rosati, P., Lejeune, A., and Emeakaroha, V. (2017). A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In IEEE CloudCom), pages 162–169.

[Malla and Christensen 2020] Malla, S. and Christensen, K. (2020). Hpc in the cloud: Performance comparison of function as a service (faas) vs infrastructure as a service (iaas). Internet Technology Letters, 3(1):e137.

[MELL and Grance 2011] MELL, P. and Grance, T. (2011). The NIST definition of cloud computing. National Institute of Standards and Tecnology.

[Needleman and Wunsch 1970] Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology, 48(3):443–453.

[Niu et al. 2019] Niu, X., Kumanov, D., Hung, L.-H., Lloyd, W., and Yeung, K. Y. (2019). Leveraging serverless computing to improve performance for sequence comparison. BCB'19, page 683–687. Association for Computing Machinery.

[Poccia 2020] Poccia, D. (2020). New for AWS lambda – functions with up to 10 GB of memory and 6 vCPUs.

[Sarje and Aluru 2009] Sarje, A. and Aluru, S. (2009). Parallel genomic alignments on the cell broadband engine. IEEE TPDS, 20(11):1600–1610.

[Smith et al. 1981] Smith, T. F., Waterman, M. S., et al. (1981). Identification of common molecular subsequences. Journal of molecular biology, 147(1):195–197.