Evaluation of FaaS as an alternative to build HPC environments

María Murazzo, Joaquín Lebetti, Nelson Rodríguez, Adriana Martin

Departamento e Instituto de Informática - F.C.E.F. y N. - U.N.S.J.
Complejo Islas Malvinas. Cereceto y Meglioli. 5400. Rivadavia. San Juan,
lebejoaquin@gmail.com, marite@unsj-cuim.edu.ar,
nelson@iinfo.unsj.edu.ar , adrianamartin1@gmail.com

Abstract. Running HPC applications in the cloud has proven to be a viable option to conventional parallel or distributed architectures, which require a high degree of management as well as poor resource scaling. The traditional approach for a user is to usually use the Cloud provider to provision resources to virtual machines (VMs), using them in a similar way to an on-premises infrastructure, with the consequent problem of resource management coupled with the degradation of application performance due to contextualization of virtualized environments. Serverless computing allows a user to run code written in the programming language of their choice, without first having to provision a virtual machine. On the other hand, elasticity, availability, scalability, and fault tolerance are provided transparently by the cloud provider. This way it is possible to reduce the complexity of infrastructure management for the developer, allowing him to focus on the logic of the application. Also, economic advantages arise, when just paying for usage time. The work focuses on the challenge of evaluating the cost, not only monetary but also of performance, of migrating HPC applications to serverless environments. This evaluation will allow the decision to be made REGARDING which infrastructure will be used, in order to obtain the best performance benefit.

Keywords: IoT, Cloud Computing, HPC Serverles Computing, FaaS, GCP, Cloud Function.

1. Introduction

The increasing popularity of IoT and the massification of cloud infrastructures have recently opened up a world of possibilities for HPC applications. This is due to the huge quantity of data that IoT devices generate, which makes it impractical to treat them with traditional paradigms. To achieve the adequate processing of this kind of data with significant speed and size characteristics, it is mandatory to dispense with traditional programming paradigms [1]. Therefore, it is necessary to apply algorithms to take advantage of the scalability of computing and data processing resources [2] [3]. In this sense, it is proposed high performance computing techniques (HPC) as a solution to data processing coming from IoT so as to increase processing performance.

Although HPC architectures have evolved to get better response times for applications, they have the drawback of scaling computing resources, which is why migrating to the cloud becomes a reasonable alternative [4].

Like any other service, Cloud Computing has been characterized as a technology focused on providing on-demand computing, which is an advantage for assembling applications where intensive processing is necessary [5].

Cloud providers claim many advantages in HPC application migration, such as fast access to resources, lower costs and flexibility in sourcing and provisioning [6]. However, the cloud has two major disadvantages, the first one being the degradation of the applications performance when assembled on virtualized architecture, since it generates overhead in the contextualization of virtual machines; and the second disadvantage when deploying applications in the cloud, is that the organization must be responsible for keeping the infrastructure needed to deploy the applications working properly, which leads to charging costs on the budget for its maintenance and support [7].

In this sense, the emergence of Serverless Computing [8] means that developers do not have to worry about infrastructure provisioning and scaling, so they can focus on their applications logic. In this way, it is possible to achieve the abstraction of server management (provisioning, configuration, scaling, etc.) so that users, in this case developers, can focus on the aforementioned logic.

2. Related jobs

The advantages and disadvantages of migrating HPC applications to the cloud have been mentioned in previous paragraphs. However, the time and effort required to configure the virtual resources may be greater than the actual time and effort spent doing the calculations. On the other hand, if the serverless paradigm is used it will be possible to have more granular control over the service provided, leaving infrastructure management in the hands of the cloud provider.

In [9], a systematic mapping of 89 use cases has been carried out where the serverless paradigm was applied to solve mostly HPC-related problems. But there is little information on a performance comparison between the applications running on the serverless paradigm versus the same applications running on a traditional cloud infrastructure, in which a behaviour analysis can be done to later decide which the best solution to run HPC applications is.

This work uses [10] as a starting point and deepens the research tasks based on [11], [12], [13] among others in recent years, in which it has been explored and evaluated the performance of the use of serverless in HPC applications. While these studies show that serverless is easy to use and inexpensive, its effectiveness over the conventional approach to cloud applications has not yet been quantified.

3. Work Scenarios

Three scenarios are proposed: two main scenarios and another secondary one; in which it will be evaluated: Contextualization time, Execution time and Price. The implementations and study were carried out in the Google cloud (Google Cloud Platform).

The first evaluation scenario is the "traditional" cloud. This scenario is made on a resource that functions as PaaS for which Datproc will be used, that allows to operate a cluster with Spark of 4 nodes, 3 workers and 1 master, with a maximum of 8 vCPUs in all. On the cluster the interfaces provided by Google are used to operate with the cluster, mainly the REST resource to send jobs via HTTP and the Jupyter notebook.

The second scenario runs on Cloud Functions, it consists of 4 nodes, 3 workers and 1 master to make an equivalence with a cluster of 3 workers and 1 master of the Dataproc configuration. The master node will be responsible for triggering the worker function 3 times to boot 3 instances "simultaneously" and then collect the results obtained by each of those instances, to figure out a unique result that will be the final result of the problem to solve. To solve each part of the problem, the worker function will use Pandas to work the data file as a dataframe.

The third scenario is a combination of the previous PaaS (Dataproc) scenarios and "on-demand" FaaS (Cloud Functions) cluster. Through an event, in this case an HTTP request, a function is executed which, in turn, generates a template for automatic cluster generation that is sent to the Dataproc API. After that cluster creation, you start executing processing and then shut down and remove the cluster.

4. Problem to solve and configuration behaviour

The problem to be solved is to determine, based on historical data obtained by sensors from 2009 to 2022, the time of day when the amount of CO (carbon monoxide) is lowest. For this purpose, a dataset from open data provided by the government of Buenos Aires on the site https://data.buenosaires.gob.ar/dataset/calidad-aire is used. The dataset has 110,000 records (one record for each time of day, from 2009 to 2022 in Centenario, Córdoba and La Boca cities). It was performed a record cleaning without data, after which it was obtained a dataset with 70,000 records.

To make a valid comparison, the resolution of the problem must be compared with two equivalent configurations. The minimum resource configuration for each node in Dataproc is 1 vCPU and 3.5GB of RAM; therefore, the evaluation will be made on customized machines with these resources on the first and third configuration, and on instances of 2GB of RAM and 1 vCPU on the second configuration since it is an even number of resources that is compared despite there being a difference in RAM.

4.1 Scenario 1: PaaS setup

The aspects evaluated in this configuration are considered once the cluster is in operation, the time from when the cluster is created until it shuts down will be taken into account in scenario 3 that corresponds to the PaaS Configuration with executing function.

Execution time. To obtain an accurate execution time, the average execution time of 10 jobs is considered (see Fig.1), where the resulting average time is 56 seconds.

Estado	Región	Tipo	Clúster	Hora de inicio	Tiempo transcurrido
Completado	us-central1	PySpark	cluster-3w	9 oct 2022 17:13:41	59 s
Completado	us-central1	PySpark	cluster-3w	9 oct 2022 17:11:33	1 min 2 s
Completado	us-central1	PySpark	cluster-3w	9 oct 2022 17:09:50	55 s
Completado	us-central1	PySpark	cluster-3w	9 oct 2022 17:07:48	57 s
Completado	us-central1	PySpark	cluster-3w	9 oct 2022 17:05:32	53 s
Completado	us-central1	PySpark	cluster-3w	9 oct 2022 17:03:51	54 s
Completado	us-central1	PySpark	cluster-3w	9 oct 2022 17:02:23	57 s
Completado	us-central1	PySpark	cluster-3w	9 oct 2022 17:00:50	55 s
Completado	us-central1	PySpark	cluster-3w	9 oct 2022 16:58:58	57 s
Completado	us-central1	PySpark	cluster-3w	9 oct 2022 16:55:50	55 s

Fig.1. Execution time of 10 jobs.

Price of the solution. Having the necessary resources provisioned only involves solving the problem and it is the price per second of having an n1-standard-1 instance running: 0.00001319444 U$D. If the cluster consists of 4 instances of this type and is running for 56 seconds, the total cost of the 4 instances in Cloud Engine is: 0.00001319444 U$D / s * 56 s * 4 = 0.003166664 U$D. If the cost of the Dataproc fee is added, the final cost of having the cluster running for 56 seconds is: 0.0006222222 U$D + 0.003166664 U$D = 0.00378888884 U$D.

4.2 Scenario 2: FaaS Configuration

What it has been observed in this configuration covers all the aspects, since they are measurable from it and it does not require another particular scenario for an accurate measurement.

Contextualization time. A new function instance is started in two cases: When the function is deployed or when a new function instance is automatically created to scale up (vertically) to the load, or occasionally, to replace an existing instance. Starting a new function instance involves loading the runtime environment and code. Requests that include the launch of function instances, called cold starts, may be slower than those routed to existing function instances. The occurrence of an error in the execution of the code implies a restart of the instance and therefore its corresponding cold start.

In Fig. 2, you can see the difference in execution times between a hot and a cold instance, you can see that the light blue instance is a hot instance because the INIT log does not appear, in addition to the fact that the execution takes only 462ms. Although in the three, the execution of the function took approximately 450 ms, it can be seen that in the last two ones, since the initialization time is also added, the final execution time is 3322 ms and 3378 ms respectively.

9t8jca2pbvdj	Function execution started
by9j50pcessg	Function execution started
51rv9121ljhe	Function execution started
9t8jca2pbvdj	0.4379088878631592
9t8jca2pbvdj	Function execution took 462 ms. Finished with status code: 200
by9j50pcessg	INIT TOOK:2.041236400604248
51rv9121ljhe	INIT TOOK:2.0451221466064453
by9j50pcessg	0.43555116653442383
by9j50pcessg	Function execution took 3322 ms. Finished with status code: 200
51rv9121ljhe	0.4924287796020508
51rv9121ljhe	Function execution took 3378 ms. Finished with status code: 200

Fig. 2. Initialization time of a function instance.

The hot instance corresponds to the same instance that was initialized when updating the function, so the call of the first worker is routed to that instance, which of course also had its initialization time, but in a previous execution as shown in Fig.3.

```
2022-10-07 19:51:21.718 ART  worker-generic  9t8jhr8aq296  INIT TOOK:2.350526809692383
{
  insertId:
  "6340ad69000af76296f6708a"
  labels: {
   execution_id: "9t8jhr8aq296"
   instance_id: "00c61b117c43df13668582e06dee0cfc3765728c86c723f6abbcf1a8f30984a25f52db6e652abb01e485bb312d1a254aa3d83133d13e5a6f9d06"
2022-10-07 19:55:19.445 ART  worker-generic  9t8jca2pbvdj  0.4379088878631592
{
  insertId:
  "6340ae570006cc27b80b8950"
  labels: {
   execution_id: "9t8jca2pbvdj"
   instance_id: "00c61b117c43df13668582e06dee0cfc3765728c86c723f6abbcf1a8f30984a25f52db6e652abb01e485bb312d1a254aa3d83133d13e5a6f9d06"
```

Fig. 3. Detailed logs showing that the execution of the light blue instance corresponds to a previous initialization of the instance.

The contextualization time of the functions, calculated as the average of 12 initializations, resulted in: 2.02986383438 seconds, for this case.

Execution time. Since contextualization time is different for both hot and cold instances, the execution time is calculated for both types of instances. To obtain an execution time, 10 executions of the master function are considered.

Hot run executions. Hot run executions are those that already have the execution environment and the code loaded, so there is no contextualization time in them. Next, it will be described what was observed when solving the problem with hot functions.

- Master instances: After carrying out 10 executions of the master function, the average execution time obtained is 295.3 ms.
- Worker instances: The execution time for the worker instances was calculated with the time resulting from each execution in the total number of worker instances executed by calling the master instance 10 times. The time resulting from the

calculation of the average in 30 instances is 177.4 ms.

Cold run executions. Cold executions require an extra contextualization time to the time necessary for the execution of the function.

- Master instances: it was obtained by averaging the times of 10 executions carried out, and the resulting time was 4423 ms.
- Worker instances: it was calculated from the 30 executions carried out by the master function. The time resulting from the calculation of the average in 30 instances is 2701 ms. This value considers the initialization time of the cold function, removing the initialization time from this equation, the average final execution time of the 30 executions is: 0.38844459056 s.

Price of the solution. The price calculated for the FaaS solution takes into account the runtime of the cold instances. In Cloud Functions, the price of having each worker instance running for 2700 milliseconds is charged to the nearest 100 milliseconds: 27 * 0.000005800 USD = 0.0001566 USD. The solution to the problem requires 3 worker instances and 1 master instance, so the final price is: 0.0001566 U$D * 3 +

0.000012501 U$D = 0.000482301 U$D

4.3 Scenario 3: PaaS configuration with executor function

This configuration has as its main objective to measure the times involved in the contextualization and shutdown of the cluster, it is a configuration that complements what cannot be observed in scenario 1.

Contextualization time. An important aspect to take into account is that the cluster does not start working at the time the job starts executing, but rather it has a provisioning and contextualization time at the beginning, another one when it is turned off, during which it cannot be used, and also to be taken into account at the time it is charged.

As the objective of this configuration is to know the time dedicated to provisioning, contextualization of the cluster and next shutdown after processing the jobs, 10 executions of this configuration are carried out to obtain, on average, what is the time dedicated to these processes that go unnoticed when running a cluster.

- Cluster startup and resource provisioning time: 228 s = 3m48s
- Cluster removal time: 61 s = 1m01s

Making a sum of these times and the average execution time of a job (56 s) obtained in the first configuration results in a total time of: 4m49s + 56s = 5m45s

Execution time. To know the final execution time the cluster takes to solve the problem considering provisioning, contextualization, processing and next deletion, the average of 10 executions was calculated. The total average duration of the execution of the cluster obtained was: 340.5 s = 5m40s

In other words, considering the average execution of a job in the PaaS Configuration of scenario 1 and the contextualization and cluster removal times of the current Configuration, it coincides with a 5-second error since it is an average of the total

execution time of the cluster with the sum of the provisioning, running, and removed averages.

Another aspect to take into account is that this time is the time required one to create, provision, process, stop and delete. Once the cluster is provisioned, since creation is not required, the startup time decreases a significant portion of the initial time required.

Price of the solution. The price per second of having an instance of type n1-standard1 running is $0.00001319444. If the cluster is on for 5m 40s on average and consists of 4 instances, then the final price of the four instances on Compute Engine is 0.00001319444 U$D / s * 340 s * 4 = 0.01792333318 U$D. To this situation the Dataproc fee must be added, the final price of running a cluster for 340 seconds is 0.00377777777 U$D + 0.01792333318 U$D = 0.02170111095 U$D

4.4 Comparison of what was observed

When solving the problem with the different configurations, important differences were observed in the aspects studied.

Contextualization time. When comparing contextualization times of both configurations (see Fig.4), it is observed that the PaaS Configuration takes almost 5 minutes, adding the cluster provisioning and shutdown time after processing the job. On the other hand, the FaaS Configuration, when considering cold runs, have a contextualization time of 2 seconds.

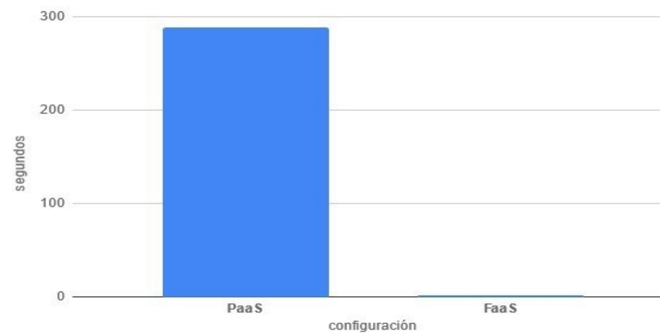

Fig.4. Contextualization time of each configuration.

Execution time. The PaaS Configuration requires 56 seconds on average, while the FaaS Configuration requires 366 milliseconds, being this less than 1% of the time that it takes the first configuration (see Fig.5). Also, the time required by FaaS is mainly dedicated to reading the file from storage.

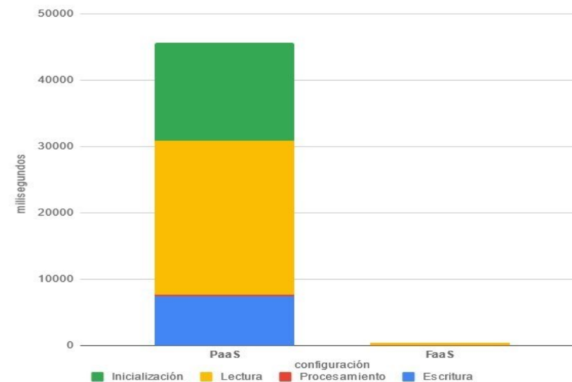

Fig.5. Runtime with stacked times.

Price. Finally, price is the aspect where there is a notable difference in using each of the configurations. As above, the price of PaaS Configuration exceeds the price of FaaS Configuration.

4.5 FaaS in detail

Next, some aspects and limitations of FaaS will be described, in addition to making some comparisons with some aspects of the scenario traditionally used in order to conclude if it is possible to build HPC environments based on FaaS and to what extent it would be feasible to do so.

Function execution time. Execution time is a key aspect for the resolution of a problem and this aspect in FaaS is limited; the maximum time that a function can be running is 540 seconds in GCP and varies depending on the provider.

Memory limit. Each instance of a function has a memory limit that is set when configuring it, function memory capacity can go from 128MB to 8GB.

To evaluate the behaviour of the functions in an HPC environment, it is necessary to take this aspect into account. The evaluation started by processing 2GB (56,999,943 rows) with each instance within the 1 master and 3 worker configurations, that is, each of the 3 worker instances read a 2GB file and processed it. From this first evaluation, it turned out that from 11 processes that were carried out (66 GB of data) only 3 of them responded correctly, that is, 27% of the total number of executions carried out (see Fig.6).

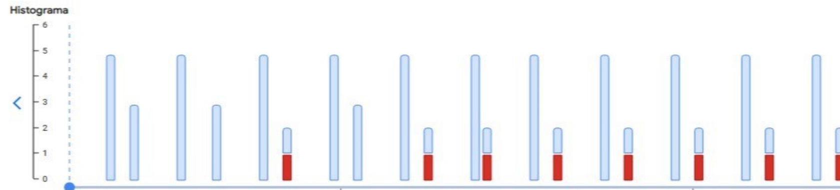

Fig.6. Percentage of correctness of the configuration.

Later on, the capacity of the worker separated from the complete configuration is evaluated to know the capacity of the running instance separated from the configuration. To do this, the percentage of executions that responded correctly with different loads was evaluated for 10 minutes, to know the point at which the percentage of correctness is 100%. The loads used, from highest to lowest, were 2 GB (56,999,943 rows), 1.3 GB (37,205,301 rows), 1 GB (29,999,971 rows), 500 MB (14,999,986 rows).

That is, for this particular problem with the processing that is done and the libraries that you use, for a function of 8 GB and 2 vCPU per instance, the optimal file size is 500 MB.

Higher scales and elasticity. Considering that the optimal file size for an 8 GB function is 500 MB, for each configuration of 1 master and 3 workers 1.5GB is processed.

As FaaS are self-scalable in terms of the number of received requests, it is possible to execute up to 10,000 instances simultaneously. For this case, 60 instances of the master function were executed simultaneously and, as it can be seen in Fig.7, the number of running instances can go from 0 instances to 60 almost instantly; so if each configuration of 1 master and 3 workers processes 1.5 GB, with 60 runs 90 GB are processed.

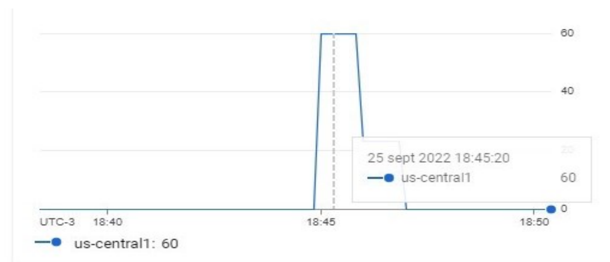

Fig.7. Number of instances of the master function running concurrently.

Large amount of data processed in a limited time. As seen above, 60 instances of the master function were executed, which involved processing 90 GB in all. The important thing about this is that, since FaaS runs in a different environment than the rest of the other executions of the same function, executions can overlap and manage to process large amounts of data in a short period of time. In Fig 8., it is shown that the average time taken to execute each processing in each execution is 29 seconds.

```
✓ _success_rate..................: 100.00% ✓ 60       ✗ 0
  data_received..................: 741 kB  14 kB/s
  data_sent......................: 39 kB   713 B/s
  http_req_blocked...............: avg=426.47ms min=413.53ms med=426.34ms max=441.73ms p(90)=434.5ms   p(95)=435.04ms
  http_req_connecting............: avg=70.69ms  min=34.84ms  med=70.37ms  max=134.82ms p(90)=94.57ms   p(95)=96.94ms
  http_req_duration..............: avg=28.93s   min=26.75s   med=28.42s   max=33.6s    p(90)=30.93s    p(95)=31.5s
    { expected_response:true }...: avg=28.93s   min=26.75s   med=28.42s   max=33.6s    p(90)=30.93s    p(95)=31.5s
  http_req_failed................: 0.00%   ✓ 0        ✗ 60
  http_req_receiving.............: avg=1.14ms   min=188.3▒s  med=466.1▒s  max=34.94ms  p(90)=935.48▒s p(95)=1.57ms
  http_req_sending...............: avg=159.04▒s min=73.4▒s   med=118.2▒s  max=714.6▒s  p(90)=283.57▒s p(95)=379.53▒s
  http_req_tls_handshaking.......: avg=185.49ms min=113.01ms med=190.45ms max=213.9ms  p(90)=207.73ms p(95)=209.62ms
  http_req_waiting...............: avg=28.93s   min=26.75s   med=28.42s   max=33.6s    p(90)=30.93s    p(95)=31.46s
  http_reqs......................: 60      1.109997/s
  iteration_duration.............: avg=49.36s   min=47.17s   med=48.84s   max=54.04s   p(90)=51.36s    p(95)=51.93s
  iterations.....................: 60      1.109997/s
  vus............................: 1       min=1      max=60
  vus_max........................: 60      min=60     max=60
```

Fig.8. Duration of http request to master function.

That is, if the average execution took 30 seconds, and 60 simultaneous executions were launched, as each execution processes 1.5 GB, 90 GB of data was processed in 30 seconds.

Costs of the processing performed. The price per 100 ms of running an instance of an 8 GB function is 0.000006800 U$D. If each instance ran on average 30 seconds, the price per instance is 0.000006800 U$D * 300 = 0.00204 U$D. If 60 instances of the master function were executed and each master function executes 3 instances of the worker function, then the total cost for processing 90 GB in 30 seconds is 0.00204 U$D * 60 * 3 = 0.3672 U$D. Adding to this, the cost of running the 60 master instances that are 128MB the total price for running the environment for 30 seconds is 0.3672 U$D + 0.004158 = 0.371358 U$D

5. Conclusions

It is possible to build an HPC environment in Functions as a Service, but depending on the problem to be solved, it may or may not be a viable solution.

Positive points for which it could be viable to build an environment based on Functions:

- Fast processing of medium loads.
- Elasticity, allowing large amounts of data to be processed with multiple instances running simultaneously.
- Saving on costs.
- Automation and integration with external services.

Negative points by which Functions as a Service are not a reliable alternative to build an HPC environment:

- While the user does not take over the infrastructure, there is no actual autoscaling, he must know approximately the number of instances to run.
- With little flexibility, making changes to the processing code could cause the environment to require an increase in resources or be destroyed if it reaches the maximum.

- Inability to make an analysis of the data; an external application is required to view the results.
- Technically it works only for weakly coupled problems and specific cases.
- There are limitations of time and memory imposed by technology.
- It takes extra effort to develop the environment to solve a problem with some degree of coupling.
- Limitations in the amount of data that can be processed (5 TB) considering the limit of 10,000 running instances.
- The possible cost reduction translates into the rigidity of the environment, its development, maintenance and its possible disablement in case of a change in the characteristics of the problem to be solved.
- Limited customization of instances.

For all these aspects, depending on the problem that is sought to be solved, it may be feasible to build the environment based on FaaS. The most important benefits are the speed and elasticity of the functions, which require a loosely coupled or zerocoupling problem of low-medium load (up to 1TB) to work optimally. The monetary benefit can be considered, but it is not the main thing that should be considered when building an HPC environment.

References

1. Farhan, L., Kharel, R., Kaiwartya, O., Quiroz-Castellanos, M., Alissa, A., & Abdulsalam, M. (2018, July). A concise review on Internet of Things (IoT)-problems, challenges and opportunities. In 2018 11Th International Symposium On Communication Systems, Networks & Digital Signal Processing (CSNDSP) (pp. 1-6). IEEE.
2. Medel, D., Murazzo, M. A., Molina, A. L., Sánchez, F., Cornejo, M., Rodríguez, N. R., ... & Piccoli, M. F. (2019). La Computación de Alta Performance como soporte a los sistemas altamente distribuidos. In XXI Workshop de Investigadores en Ciencias de la Computación (WICC 2019, Universidad Nacional de San Juan).
3. Barrionuevo, M., Escalante, J., Lopresti, M., Lucero, M., Miranda, N. C., Murazzo, M. A., & Piccoli, M. F. (2020). Solución de grandes problemas aplicando HPC multitecnología. In XXII Workshop de Investigadores en Ciencias de la Computación (WICC 2020, El Calafate, Santa Cruz).
4. de Souza Cimino, L., de Resende, J. E. E., Silva, L. H. M., Rocha, S. Q. S., de Oliveira Correia, M., Monteiro, G. S., ... & de Castro Lima, J. (2017, November). IoT and HPC integration: revision and perspectives. In 2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 132-139). IEEE.
5. Biswas, A. R., & Giaffreda, R. (2014, March). IoT and cloud convergence: Opportunities and challenges. In 2014 IEEE World Forum on Internet of Things (WF-IoT) (pp. 375-376). IEEE.
6. Añel, J. A., Añel, J. A., Montes, D. P., Iglesias, J. R., & Romano. (2020). Cloud and Serverless Computing for Scientists. Springer International Publishing.
7. Malla, S., & 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.

8. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., ... & Suter, P. (2017). Serverless computing: Current trends and open problems. In Research advances in cloud computing (pp. 1-20). Springer, Singapore.
9. Eismann, S., Scheuner, J., Van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., ... & Iosup, A. (2020). A review of serverless use cases and their characteristics. arXiv preprint arXiv:2008.11110.
10. Rodríguez, N. R., Murazzo, M. A., Medel, D., Parra, L., Molina, A. L., Sánchez, F., ... & Vargas, L. (2021). Procesamiento paralelo sobre arquitecturas serverless para tratamiento de datos provenientes del IoT. In XXIII Workshop de Investigadores en Ciencias de la Computación (WICC 2021, Chilecito, La Rioja).
11. Niu, X., Kumanov, D., Hung, L. H., Lloyd, W., & Yeung, K. Y. (2019, September). Leveraging serverless computing to improve performance for sequence comparison. In Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics (pp. 683-687).
12. Spillner, J., Mateos, C., & Monge, D. A. (2017, September). Faaster, better, cheaper: The prospect of serverless scientific computing and hpc. In Latin American High Performance Computing Conference (pp. 154-168). Springer, Cham.
13. Chard, R., Skluzacek, T. J., Li, Z., Babuji, Y., Woodard, A., Blaiszik, B., ... & Chard, K. (2019). Serverless supercomputing: High performance function as a service for science. arXiv preprint arXiv:1908.04907.