Outsourcing Data Processing Jobs With Lithops

Josep Sampé, Marc Sánchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro García-López, *Member, IEEE*

Abstract—Unexpectedly, the rise of serverless computing has also collaterally started the "democratization" of massive-scale data parallelism. This new trend heralded by PyWren pursues to enable untrained users to execute single-machine code in the cloud at massive scale through platforms like AWS Lambda. Driven by this vision, this article presents *Lithops*, which carries forward the pioneering work of PyWren to better exploit the innate parallelism of à la MapReduce tasks atop several Functions-as-a-Service platforms such as AWS Lambda, IBM Cloud Functions, Google Cloud Functions or Knative. Instead of waiting for a cluster to be up and running in the cloud, *Lithops* makes easy the task of spawning hundreds and thousands of cloud functions to execute a large job in a few seconds from start. With Lithops, for instance, users can painlessly perform exploratory data analysis from within a Jupyter notebook, while it is the Lithops's engine which takes care of launching the parallel cloud functions, loading dependencies, automatically partitioning the data, etc. In this article, we describe the design and innovative features of Lithops and evaluate it using several representative applications, including sentiment analysis, Monte Carlo simulations, and hyperparameter tuning. These applications manifest the Lithops' ability to scale single-machine code computations to thousands of cores. And very importantly, without the need of booting a cold cluster or keeping a warm cluster for occasional tasks.

Index Terms—Serverless computing, cloud computing, distributed systems, lithops, PyWren, IBM cloud, multi-cloud

1 Introduction

The emerging cloud computing category of "*Functions-as-a-Service*" (FaaS) has recently become a hot topic in the cloud world. Solid proof of that is that big cloud vendors such as Amazon, Microsoft and IBM have rushed their own versions of FaaS to market, and we have already seen plenty of top conferences and articles dedicated to this subject [1], [2], [3], [4], [5]. The shift from the server to the programming level, the ability to quickly launch thousands of concurrent functions, and fine-grained subsecond billing have spurred many users to embrace serverless computing for a variety of applications (e.g., microservices, IoT, machine learning), some of them somewhat far from the original intents of the FaaS computing model.

One of these apparently *anti-natural* intents has been the use of cloud functions for massive-scale parallel computing. In other words, although cloud functions were intended for asynchronously invoked microservices, their scale and short provisioning time enabled researchers to examine a different use: as a *massively-parallel, on-demand cloud computing system*. Indeed, the first to do so were Eric Jonas *et al.* from Berkeley, which prototyped PyWren [3] in 2017, and showed how a serverless execution model with *stateless* cloud functions can significantly lower the barrier for users to leverage the cloud for massively parallel workloads. Other recent works such as [2], [6], [7] have validated this vision.

In general, running analytics applications on a bunch of cloud functions is difficult. The framework layer, which sits on top of the serverless execution layer, needs to address a number of restrictions *endemic* to this brand-new computing environment such as:

1. A simplistic way to describe parallel computations such as the MapReduce programming model;
2. The impossibility of direct, peer-to-peer communication between functions;
3. The lack of support for function synchronization, so as between the `map` and `reduce` stages;
4. The situation that dependencies and libraries may differ in a cloud function compared with a local machine; and
5. The lack of portability since each cloud provider has its own set of proprietary APIs.

As of today, the works that have been devoted to leveraging the FaaS platforms for large-scale data analytics, have partly overcome the above issues, but not all at once. For instance, ExCamera [8] and `gg` [7] address the impossibility of direct communication via TCP connections brokered by a TURN server, but fail to provide a simplistic programming model such as MapReduce. Or PyWren [3], which does not provide a `reduce` primitive and only works on top of AWS Lambda.

- *Josep Sampé and Marc Sánchez-Artigas are with Universitat Rovira i Virgili, 43007 Tarragona, Spain. E-mail: {josep.sampe, marc.sanchez}@urv.cat.*
- *Gil Vernik is with IBM Research, Haifa 3498825, Israel. E-mail: gilv@il.ibm.com.*
- *Ido Yehekzel is with Technion, Haifa 32000, Israel. E-mail: idoyehe@gmail.com.*
- *Pedro García-López is with Universitat Rovira i Virgili, 43007 Tarragona, Spain, and also with IBM T.J. Watson Research Center, Ossining, NY 10598 USA. E-mail: pedro.garcia@urv.cat.*

Manuscript received 10 April 2020; revised 10 June 2021; accepted 14 November 2021. Date of publication 18 November 2021; date of current version 8 March 2023.
This research was supported in part by EU under Grant 825184 and in part by the Spanish Government through Project No. PID2019-106774RB-C22. Marc Sánchez-Artigas is a Serra-Húnter Fellow.
(Corresponding author: Marc Sánchez-Artigas.)
Recommended for acceptance by Y. Yang.
Digital Object Identifier no. 10.1109/TCC.2021.3129000

1.1 Contributions

To address all the above limitations within a single unified system, we present *Lithops* [9]. Lithops is a serverless, multi-cloud framework for building both embarrassingly parallel and MapReduce-like cloud-functions applications over the most popular cloud platforms, such as IBM Cloud, Google Cloud and AWS. Lithops helps non-cloud users outsource their "everyday" programs (e.g., Monte Carlo simulations, sentiment analysis, ...) to the cloud and execute them using FaaS platforms, thereby enabling hundreds-way parallelism on *short-lived* cloud functions. This provides major benefits in terms of performance for "occasional" users of the cloud, who are puzzled by the wide array of choices that must be made before executing a simple application in parallel (e.g., VM instance type, cluster size, programming models, etc.). Using simple commands such as the `map()` primitive, users can effortlessly spawn more than 1,000 concurrent function instances of the application code to be run transparently in the cloud with Lithops.

Specifically, Lithops overcomes the above challenges as follows. First, it implements a simple MapReduce interface that enables parallel jobs out of single-machine Python code in the cloud. Second, it comes along with an intuitive storage API to allow functions to communicate among one another, and with the Lithops client. Third, synchronization between `map` and `reduce` stages is implicit, including *nested* function compositions, thus releasing the user from this tedious task. Fourth, it automatically handles dependencies by exploiting the dynamism of the Python language. Finally, its has been architected to abstract away the details of cloud providers and mitigate vendor lock-in.

We summarize our contributions as follows:

- We present *Lithops*, a multi-cloud framework to enable the massively-parallel execution of ordinary programs à la MapReduce by running them over cloud functions. Many everyday computing and scientific tasks exhibit a significant degree of innate parallelism (e.g., sentiment analysis), and can be run as MapReduce tasks. Lithops allows to transparently instruct hundreds of CPU cores as cloud functions, run the resulting parallel task out of single-machine Python code, and collect the results. The source code is available on the Github repository [9].
- Using several scientific programs (i.e., stock prediction, hyper-parameter tuning, and sentiment analysis), we gauge its performance. Compared with a local machine, Lithops can lead to speedups larger than 100x, without the need of having a *warm cluster running continuously* in the cloud. We also assess its core architectural aspects when running it at large scale.

A preliminary version of this work has been accepted by Middleware'18 (Industry Track)[1] [10]. The rest of this paper is structured as follows. We review related work in Section 2, and present the system architecture in Section 3. We detail the programmability of Lithops in Section 4, and argue its elasticity in Section 5. Finally, we evaluate the framework in Section 6, concluding this work in Section 7.

1. Compared to [10], IBM-PyWren, now rebranded to Lithops, offers support for multiple cloud providers, event-based monitoring, a richer (storage) API and the evaluation of two new scientific applications.

TABLE 1
Differences Between PyWren [3] and Lithops

	PyWren	*Lithops*
MapReduce	*Mapping portion; reducing is still experimental.*	*Broader support for MapReduce jobs. It includes a `reduceByKey`-like operation to run one reducer per object key in parallel.*
Data discovery & partitioning	*None.*	*Automatic; data partitioning based on user-defined chunk sizes or on the data object granularity.*
Composability	*None.*	*Dynamic compositions of functions; e.g., sequences: $f_3 = f_2 \circ f_1$.*
Runtime	*Fixed; AWS Lambda, along with a Anaconda, a packaged version of Python.*	*Based on Docker; possibility for users to create its own custom runtime (a different Python version, extra packages) and share it with other users.*
Proxied function invocation	*Due to network overhead & AWS throttling, it may be slow to launch jobs with many functions.*	*Faster; client calls a remote invoker function, which starts all functions in parallel within the cloud.*
Monitoring	*Polling-based monitoring using Amazon S3*	*More efficient; event-based monitoring using RabbitMQ*
Open-source portability & extensibility	*Adapted to work with AWS Lambda and Amazon S3*	*Multi-cloud: IBM Cloud, Google Cloud, AWS, Azure, etc., in addition to Kubernetes*

2 Related Work

Based on the MapReduce programming model, Lithops has multiple antecedents —e.g., cluster-computing systems such as Hadoop [11] and Spark [12]. The key difference of Lithops with these systems is the use of a new computing substrate (cloud functions), mode of execution (parallel but scaling up from zero) and application domain (ordinary Python code featuring innate parallelism), all without managing servers.

Now turning attention to the new computing substrate of cloud functions, we investigate how Lithops fits with the prior literature. Non-surprisingly, the closest related work is PyWren [3]. PyWren enables the automatic conversion of a Python *user-defined function* (UDF) into a massively-parallel `map` task. The differences between Lithops and PyWren are significant. First off, PyWren only works with AWS Lambda. Lithops is multi-cloud — i.e., the same code can run on IBM Cloud, AWS, or Google Cloud with no changes. It allows to execute a broader scope of MapReduce programs, supports multiple storage backends, and optimizes the spawning of thousands of cloud functions to 9 seconds. PyWren, function invocation can take up to 30 seconds due to AWS throttling [3]. A detailed comparative of Lithops and PyWren is listed in Table 1. To allow PyWren to shuffle data efficiently, [13] proposes to combine the AWS ElastiCache service for Redis [14] with the much cheaper Amazon S3 (object storage). As efficient data shuffling is critical for Map-Reduce programs, we are now integrating Primula [15] with Lithops.

Furthermore, several recent attempts have been made to implement MapReduce with serverless technology. Works such as Qubole [16] and Flint [17] have attempted to enable Apache Spark over AWS Lambda service. Qubole creates

the execution DAG on the client side while it executes each task as a cloud function. In addition to the obvious users' difficulty to learn the Spark's API, these approaches present performance issues. For instance, Qubole reports executor startup times to be around 2 minutes in the cold start case or the problem of data shuffling in Flint. In brief, Flint uses Amazon SQS for the shuffling of intermediate data, which can generate duplicate messages [17] and is slow [18].

Other projects have tried to implement a MapReduce-like serverless framework from scratch such as Corral [19] and Lambada [20], which are still under active development. Many open issues remain in these systems, being efficient data shuffling one of the most salient challenges to address. In the sense, the aforecited work of Pu *et al.* [13] have striven on how to make data shuffling more efficient by leveraging AWS ElastiCache.

Last but not least, there are other serverless systems that can be leveraged to run different flavors of MapReduce jobs such as Apache Pulsar [21] for serverless stream processing, or Crucial [22] for stateful FaaS-based processing.

3 Architecture

The high-level architecture of Lithops is depicted in Fig. 1. This architecture is general enough to support standard APIs such as the Python `concurrent.futures`[23] API. In its most fundamental incarnation, Lithops leverages just two different cloud services: the *compute backend* to launch MapReduce jobs; and the *storage backend* to store all data, including intermediate results. To keep Lithops completely serverless, the compute backend is typically a FaaS platform (e.g., IBM Cloud Functions) and the storage backend is a BaaS[2] storage service (e.g., IBM COS), so that its two main pillars can scale independently from each other.

Internally, the main components of Lithops are:

- *Executor*, which allows end users to execute their code in the cloud through simple API calls. Upon an API call, it serializes and uploads the single-machine user code and input data from her local machine (e.g., laptop) to the storage backend. When a cloud function finishes its execution, the output data generated by executing the user code within the cloud function is persisted to the storage backend. For this reason, the executor monitors the storage backend for the ouput data and transfers it to the user's local machine when available.
- *Invoker*, which performs the "appropriate" number of function invocations against the compute backend. We say "appropriate" since the number of cloud functions depends on the API call itself. The *invoker* can be run on the cloud to hide the high invocation latency when the Lithops client is very far from the compute backend.
- *Worker* is the workhorse of Lithops. In short, it runs on the compute backend, typically as a cloud function, and its main role is to execute the user code associated with the API call that spinned it up. In essence, it fetches the input data and user code from the storage backend, and executes it, eventually saving the output to the storage backend. Throughout the article, we shall use the term *worker* and *function executor* interchangeably.

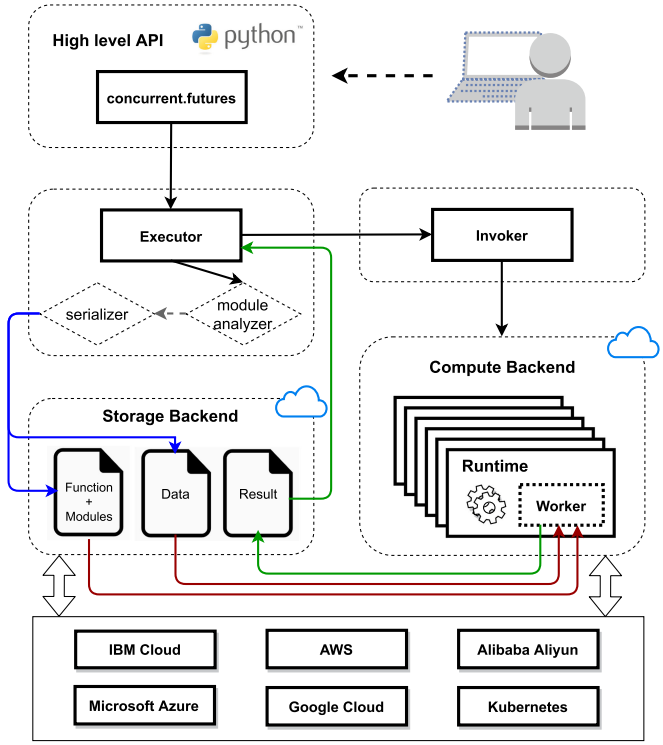


Fig. 1. High-level Lithops architecture.

Lithops has been implemented in the Python language. It capitalizes on Python's language dynamism to *transparently* capture dependencies and related modules, and send them to workers for execution *at runtime*. This has thwarted most of the hindering barriers about deployment, packaging and job execution that inhibit most users from *painlessly* entering the cloud. In this sense, Lithops can be seen as a *dynamic* task orchestrator, which can transparently take a user's code and input data, save them into the storage backend, and execute it at large scale by dynamically invoking a *generic worker function*. This approach removes the overhead for function registration, favors the reuse of the *single* registered function in order to mitigate cold starts, and allows to run user code that exceeds the deployment package size constraints, since dependencies are injected at runtime, not statically.

3.1 Runtime

The runtime is the environment where the user code runs. As of today, all the compute backends supported by Lithops permit to run functions in containerized environments, so it is not rare that Lithops runtimes are built on Docker images. The fact of leveraging Docker technology allows developers to create their own custom runtimes. Simply put, a user can prepare a Docker image with the required packages (Python modules, system libraries and binaries), and then store it to a container registry (e.g., Docker Hub registry). The compute backend service will get the container from the registry the first time is needed. From thereon, the image is cached in a local registry to speed up subsequent invocations.

2. BaaS (*Backend-as-a-Service*) is a term that has evolved in the last few years to describe any application-specific serverless cloud service, such as serverless databases.

Remarkably, this feature enables the possibility to easily share runtimes among colleagues. For example, a scientist may create a Docker image with the package `matplotlib`, a library to create 2D figures, and share it with other users via the Docker Hub registry, avoiding them the overhead to create the same runtime with this specific library.

To have no side effects, both the client and server ends have to use the same version of Python. To fulfill this need with zero user overhead, Lithops automatically detects the Python version of the client and accordingly deploys the runtime based upon this information. By now, Lithops can transparently deploy runtimes for Python $>= 3.5$. These runtimes include the most common Python modules, so that data scientists can start using Lithops right away. It is worth to mention here that all cloud providers have limitations in what refers to the resources available for the runtimes. At the time of writing, the most powerful cross-vendor setup allows an execution time limit of 600 seconds, 2GB of RAM per function execution, and 1,000 concurrent invocations, albeit these numbers can be increased if needed.

4 Programming Model

One fundamental principle behind Lithops is programming simplicity. As in the pioneering work of Jonas *et al.* [3], our objective was to make this tool as much usable as possible, irrespective of whether the programmer is a cloud expert or not. For this reason, our tool's API resembles that in PyWren, but with extra functionality like the `map_reduce()` method to offer a broader MapReduce support.

Furthermore, we have devoted extra efforts to integrate Lithops with other easy-to-use, scientific tools like Jupyter notebooks [24]. The dynamic nature of Lithops make it a perfect fit for exploratory data analysis (EDA) from within a notebook. Python notebooks are interactive computational environments, where you can combine code execution, rich text, mathematics, plots and rich media. To this goal, IBM Cloud offers a service called IBM Watson Studio [25], which, among other things, allows to create and execute notebooks in the cloud. Lithops can be easily imported from within these notebooks to run data-parallel jobs.

4.1 Function Executor

The core object in Lithops is the *executor*. This object allows to perform calls to the Lithops API to run parallel tasks. The standard way to get everything set up is to import the module `lithops`, and call the class `FunctionExecutor()` to get an instance of the executor:

```
import lithops
lth = lithops.FunctionExecutor()
```

When an instance of the executor is created, a unique ID is assigned to the instance. This unique ID is used later to keep track of function invocations and the results stored in the storage backend. The executor loads the configuration (e.g., account details) required to grant Lithops access to the compute and storage backends necessary to launch Lithops.

TABLE 2
API Specification

Method	Type	Input parameters
`call_async()`	Async.	function code, data
`map()`	Async.	map function code, map data
`map_reduce()`	Async.	map/reduce func. code, map data
`wait()`	Sync.	list of *futures*, when to unlock
`get_result()`	Sync.	list of *futures*
`plot()`	Sync.	list of *futures*, destination folder
`clean()`	Async.	list of *futures*

4.2 Application Programming Interface (API)

Lithops mimics the `concurrent.futures`[23] API. The executor instance provides access to the methods in the API. The API is comprehensive enough for novice users to execute their computations out of the box, but also simple enough for experts to easily tune the system by adjusting relevant knobs and switches (parameters). The API is listed in Table 2. It includes three main methods to run users' code in the cloud: `call_async()`, `map()` and `map_reduce()`. Moreover, it has one method to keep track of the function executions: `wait()`, and another one to download the final results from the storage backend: `get_result()`. There is also a method to create automated plots of the execution trace of the different function invocations: `plot()`, and a cleaning method to delete all the unnecessary intermediate data generated by Lithops: `clean()`.

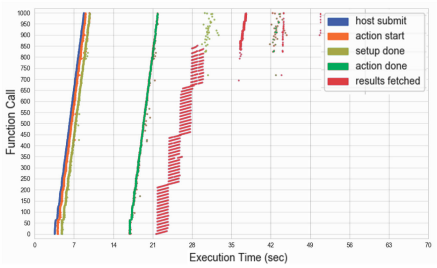
When a computing method is called (e.g., `map()`), both the user-defined (UDF) code to run as a cloud function and input data are first serialized and then saved to the storage backend. Next, the platform *transparently* interacts with the compute backend to effectively execute the user code. This requires unserializing the code and data, and run it through a Lithops worker. Once execution concludes, the results and some metadata about the status of the Lithops workers (e.g., execution times) are stored back to the storage backend.

To cater for non-skilled users, the API has been kept very simple, so that the typical behavior of invoking a computing method (e.g., `map()`) followed by a call to `get_result()` to retrieve the results does not require dealing with complex artifacts. Actually, the readiness of the results is internally managed by Lithops (e.g., see the listing in the description of the `map()` method). To deliver a finer control, all the three computing methods return *future*[3] objects. This allows more experts users to monitor the status of the function executors and retrieve the results when available.

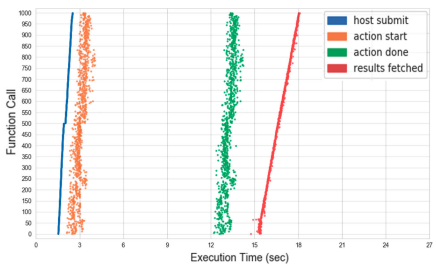
Let us review succinctly the main methods of the API:

▷ **call_async()**. This method permits to asynchronously execute just one single UDF in the cloud. As all computing methods, the output of the UDF as specified by its `return` statement is saved to the storage backend for an eventual retrieval with a call to `get_result()`. This method is non-blocking. That is, the sequential execution of the local code resumes without waiting for the results. The parameters of this command are the `function_code` and the input `data` that the Lithops worker receives.

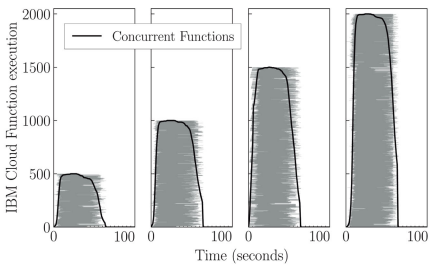
3. We mimic the Python 3.x futures interface (https://pythonhosted.org/futures/).



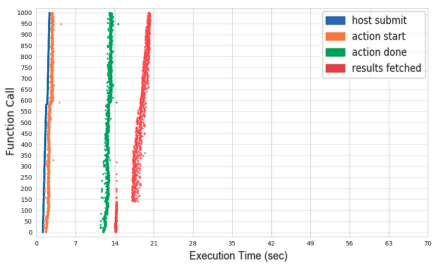
(a) Job example on PyWren (sleep of 10 seconds).



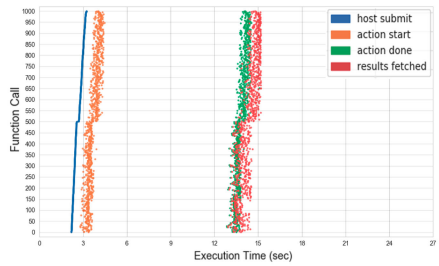
(b) Monitoring a Lithops job (sleep of 10 seconds) polling the storage backend in intervals of 2 seconds.



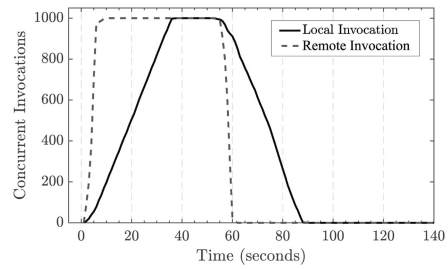
(c) Elasticity and Concurrency. Black lines show total concurrent functions. Each horizontal gray line represents a function execution.



(d) Job example on Lithops (sleep of 10 seconds).



(e) Monitoring a Lithops job (sleep of 10 seconds) using RabbitMQ to capture the termination events from the function activations.



(f) Local invoker vs remote invoker. The y-axis shows the total concurrent invocations for in each second represented by the x-axis.

Fig. 2. Evaluation of Lithops main components.

▷ **map()**. The second method is called `map()`. This method is used to run a UDF in parallel (map-style parallelism). This method is also non-blocking and it takes as main input the `map_function_code` and the `data` that the map function executors receive. Unlike the prior method, this one receives as input `data` a list of values. For each element of the input list, a separated Lithops worker is started to process the list in parallel. For example, to launch 3 function executors, the list must contain 3 elements as follows:

```
def my_map_function(x):
    return x + 7
input_data = [3, 6, 9]
lth = lithops.FunctionExecutor()
lth.map(my_map_function, input_data)
```

▷ **map_reduce()**. The third method of the API is used to run MapReduce flows, i.e., multiple map function executors (map phase), and one or multiple reduce function executors (reduce phase). This method is also non-blocking. It takes as input the `map_function_code`, the input `data` as a list of values, and the `reduce_function_code`. As in the prior method, it spawns one function executor for each value in the list.

▷ **wait()**. This method fulfills two important needs: 1) to monitor the status of concurrent function executions; and to 2) let users set synchronization points in their client code. This is the reason why this method is synchronous, i.e., the local client code is blocked until the call to `wait()` ends. Further, it provides a parameter to decide when to release the call and continue execution. There are three options: 1) '`Always`': it checks whether or not the results are available when `wait()` is invoked. If the case, it returns them. Otherwise, it resumes execution; 2) '`Any completed`': it resumes execution *upon termination of any function invocation*; and 3) '`All completed`': it waits until *all the results are available* in the storage backend. In all the cases, it returns two lists. The first list contains the futures that has been successfully completed. The second list reports the uncompleted ones. For more details about function monitoring, see Section 4.4.

▷ **get_result()**. In Lithops, the output of any UDF is never returned directly to the client, but saved to the storage backend. The reason is that the size of the output could be big, or just be *intermediate* state for a subsequent MapReduce job. Either way, it may be interesting for the user to retrieve the output data from the cloud. This need is fulfilled with the `get_result()` method. Internally, it calls `wait()` with the '`All completed`' option to ensure the completion of all concurrent function executions. When the blocking `wait()` invocation returns, the results are immediately downloaded in parallel.

Also, it adds new functionality like timeout support, and a CLI-based interface to cancel the retrieval of results, which also displays a progress bar to inform users about the % of completion of the `map()` and `map_reduce()` calls. Last but not least, this method is composition-aware: it transparently waits for an on-going function composition to complete, just returning the final result to users. See Section 4.5 for further details.

▷ **plot()**. This method is useful to display the execution trace of a parallel workflow. It outputs two plots onto the destination folder `dst` of the user's local filesystem. The first plot is a timeline diagram, which, among other things, records the invocation, activation and completion times of each function executor. Examples of this type of plot can be found in Figs. 2b and 2e. The second plot is a horizontal histogram which is very convenient to quantify the degree of

concurrency of the different function executors. An example of this plot is shown in Fig. 2c.

▷ **clean()**. This method allows to clean all the temporary data produced by Lithops once a job has ended. Usually, it is automatically called after an invocation to `get_result()`.

▷ **Storage API**. To chain MapReduce jobs without forcing the client machine to download big intermediate state from the cloud, Lithops provides the Storage API. This API makes it straightforward to operate the storage backend with calls similar to those of the Python boto3 library. To wit, one can use the Storage API to upload a file from our computer to a cloud storage service, and then read this file from a function spawned with the `call_async()` method:

```
from lithops import FunctionExecutor, Storage

BUCKET, KEY = 'my-bucket', 'test.txt'

def get_file(key, storage):
    return storage.get_object(bucket=BUCKET, key=KEY))

storage = Storage()
storage.put_object(bucket=BUCKET, key=KEY, body='Hi!')

with FunctionExecutor() as fexec:
    fut = fexec.call_async(get_file, KEY)
    print(fut.result())
```

4.3 Data Discovery and Partitioning

To provide an easy-to-use MapReduce execution platform, a key ingredient is a built-in data partitioner that abstracts users from this arduous, prone to error task. Of course, this support has been given to the `map()` and `map_reduce()` methods, along with a useful data discovery mechanism. In particular, the only action that a user has to do is to supply the list of object keys comprising the dataset. However, as a dataset may contain hundreds, or even thousands of files, it is instead possible to enumerate the name of the storage bucket(s) containing all the dataset objects. In the latter case, the framework is responsible for discovering all the objects in the bucket(s), and automatically partition them.

Once data discovery has terminated, the data partitioner enters the scene to seamlessly generate the partitions based on a configurable chunk size parameter. If no chunk size is specified, each object will be processed by a single executor. Otherwise, each data partition is automatically assigned to a function executor, which applies the map function to the data partition, and finally writes the output to the storage backend. The partitioner then executes the reduce function. The reduce function will wait for all the partial results before processing them.

Lithops supports more than one reducer. By default, the `map_reduce()` method uses one reducer for all the dataset partitions. Fortunately, it is possible to increase the number of reducers and make `map_reduce()` behave as a kind of Spark's `reduceByKey()` operator by setting the parameter `reducer_one_per_object=True`. When this parameter is enabled, all the data values for the same object key are processed by a separate reducer. This feature is very useful, since very often, it is necessary to produce a different result for every object in the dataset (e.g., see the real use case on `www.airbnb.com` in Section 6.5.3).

4.4 Monitoring

The monitoring subsystem has been devised to keep track of function executions as well as to establish synchronization points. The monitoring logic has been encapsulated into the `wait()` method, which admits three options to control when a call to this blocking method must resume execution (see Section 4.2 for details).

At a low level, the monitoring subsystem supports two ways to track the progress of function executions. These are:

- *Polling-based*, where the storage backend is *periodically* polled every x seconds to find out whether the Lithops workers have finished or not; and
- *Event-based*, where the function executors themselves signal their completion through a termination event.

We recall here that a function execution always outputs two different objects to the storage backend: a file named `result.pickle` that contains the results of the UDF as circumscribed by the `return` statement, and a metadata file called `status.json` that contains the status of the function execution, along with some statistics about the execution of the function (timestamps, etc).

▷ *Polling-based.* By default, Lithops has adopted the same monitoring mechanism as that in PyWren [3]: the periodic polling of the object storage BaaS to check the termination of the function executors. It works as follows: the Lithops client makes a HEAD request against the storage backend every x seconds to list all the available `status.json` objects. All the available files are then downloaded without further ado. In the worst case, this process is repeated until all the `status.json` files are downloaded to fulfill the '`All completed`' option.

This strategy has the major advantage that it does not require extra services for monitoring. However, it negatively impacts the job execution time due to the polling overhead: 1) although the *polling rate* is configurable (2 secs by default), periodic polling will surely increase the execution time. The magnitude of the increase depends on the number of HEAD requests. To see this, notice that if n HEAD requests were needed to ensure the completion of all UDFs, a worst-case overhead of $\mathcal{O}(n)$ seconds would be added to the execution time; and 2) a high IO overhead. Recall that each available `status.json` file will trigger a new download request to the storage backend. For a large number of executors (e.g., $>1,000$ function executors), this may become problematic, as the client has to download one small file per executor, i.e., the `status.json` file, which is bandwidth inefficient.

▷ *Event-based.* To address the above inefficiencies, Lithops supports event-based monitoring. Although this approach requires an extra service — specifically, `RabbitMQ`, it largely diminishes the monitoring overhead. It operates as follows: Lithops first creates a global queue, where all the function executors will later publish a termination event to signal the completion of each UDF. This event consists of the contents of the `status.json` file. Once all the function invocations have been spawned, the `wait()` method starts to listen to this queue and collect the events

as they are produced. This simple but efficient approach can decrease, in some cases, the monitoring overhead by around 99% compared with the polling-based approach. It is worth to mention that usually, this event-based service is deployed on a serverful virtual machine, so this component would not be serverless as the rest of Lithops components. The full evaluation and the results are in Section 6.2.

4.5 Composability

A novel feature of Lithops is function composition. It must be noted that although complex function composition still remains an open issue [5], [26], Lithops yet allows a certain level of composability. Function composability is achieved *programatically*, and not *declaratively* as in services like AWS Step Functions [27], where function composition is realized writing state machines in JSON text. A programmatic style is *a priori* more powerful as we have available all the control flow instructions from Python to compose workflows.

In commercial FaaS orchestration systems [26], the UDFs of a composition need to be first deployed to the platform before being called. On the contrary, Lithops simplifies this "boring" task to adding a few lines of boilerplate code (e.g., a call to the `FunctionExecutor()` class to get a function executor, followed by a call to `map()` for parallel execution). In other words, any regular Python function can be executed with Lithops without its prior deployment as a standalone function in the FaaS platform. This fact enables the *dynamic* and *parallel* composability of functions.

To wit, consider a simple parallel composition: a user invokes a first UDF, say `foo()`, via `call_async()`. While doing some processing, this UDF might dynamically create a list of 100 elements that would need further processing. To do so quickly, `foo()` could call the `map()` method to launch 100 parallel jobs to process them. This example illustrates that with a just few lines, it is possible to create a dynamic composition of functions:

```
import lithops

def add_seven(y)
    return y + 7

def foo(x):
   # do some processing
   rlist = random_list(x)
   lth = lithops.FunctionExecutor()
   return lth.map(add_seven, rlist)

lth = lithops.FunctionExecutor()
lth.call_async(foo, 100)
result = lth.get_result()
```

▷ *Sequences.* A common type of composition are sequences, which chain functions with one another. Each function acts on the data outputted by its predecessor in the chain. Such a composition can be realized by having each function call the successor function in the sequence through an execution method (`call_async()`, `map()` or `map_reduce()`) in its `return` statement. As a result, the list of futures generated by each invocation are returned back to the caller (e.g., see the listing above), eventually reaching the head of the chain. Finally, when the `get_result()` method receives this list of futures, it transparently starts to track all the functions in the sequence as it was a single-phase composition, letting the user program retrieve the result of the last function when the sequence finishes.

5 Elasticity and Concurrency

A high degree of elasticity (i.e., fast adaptation to workload changes) and concurrency (i.e., number of functions running concurrently) are critical to the success of Lithops, and any serverless parallel framework. Both properties are important to allow a quick execution of data-parallel tasks. Fortunately, FaaS platforms can provision a large amount of compute power quickly, and thus, perform "big data"-styled analysis with good performance.

With Lithops, it is straightforward to handle bursty and heavy workloads that require hundreds, or even thousands, of concurrent workers without waiting for machines to spin up. FaaS platforms like IBM Cloud Functions are based on containers which are fast to boot up. Consequently, function executors can be up within a sub-second range, plainly right after their corresponding invocations.

5.1 Proxied Function Invocation

When real testing Lithops, we identified some performance issues due to network latency. Concretely, we observed that a high network latency between the client and the compute backend can largely rise the total invocation time. To put it baldly, while the invocation of a *thousand* function executors from a low-latency network (i.e., within a datacenter) takes around 3 secs, this time could grow up to 40 secs (or more) in a high-latency network, despite leveraging threading to concurrently spawn the functions. Further, a higher latency turns into more invocation failures, which in turn increase the total invocation time due to invocation retries.

To overcome this issue, we designed the *proxied function invocation* mechanism, which can be enabled and disabled as needed by the user. It leverages the composability of Lithops to build a two-level invocation mechanism. In the first level, it spawns the *invoker* as a cloud function. In the second level, the *invoker* spins up the target number of function executors. Since the *invoker* is located inside the compute backend, the invocation latency is thus the lowest possible. This reduces both the total invocation time and the number of invocation failures. The *invoker* immediately returns the control to the Lithops client once it finishes all the invocations.

6 Experimental Evaluation

The evaluation of Lithops is by far extensive and covers the main components of its design. This includes monitoring, concurrency and elasticity, and proxied function spawning, as well as its practical performance in several real (scientific) programs, namely stock prediction, hyperparameter tuning, and tone analysis.

Experimental Setup. For all the experiments with Lithops, we have used the IBM Cloud services in the `us-east` region — i.e., Washington DC. As a baseline for our experiments, we have used a laptop with the following specs: Intel Core i5 (4 cores) with 16GB RAM and Ubuntu 20.04. It is

worth to notice that the focus of Lithops is to simplify the parallel execution of everyday tasks in the cloud, and not to compete with complex computing stacks running on warm clusters. In this sense, it is more interesting to assess the performance benefits for non-cloud users, who typically run programs at "laptop scale". Or to put it baldly, the benefits from shifting from the laptop to the cloud.

To test the effectiveness of proxied function invocation, we used another client machine (Intel Core i5-4 cores, 8GB RAM with Ubuntu 20.04), located in a remote network with high latency.

6.1 PyWren Versus Lithops

As a sanity check, we wanted to show the higher efficiency of Lithops compared to PyWren [3] via a representative test. More precisely, we made of use the `map()` operator, which is common in both systems, to run 1,000 executors in parallel, each running a 10-second sleep UDF with no computation.

Fig. 2a shows the results for PyWren. Fig. 2d reports the ones for Lithops. We set the same scale in both plots to ease the comparison. As can be seen in the figures, the invocation phase is 3x faster in Lithops, which finishes around second 14 compared with the 21s taken by PyWren. This difference can be explained by the action of the Lithops proxied, two-level invocation mechanism. The `get_result()` phase is also a few seconds shorter in Lithops due to our event-based termination detector compared with the polling-based one from PyWren.

6.2 Monitoring

The goal of this experiment is to compare the performance of both monitoring strategies: *polling-based* versus *event-based*. For this test, we used a simple UDF that sleeps for 10 seconds, which was run concurrently in 1,000 function executors for each monitoring strategy. For the polling-based strategy, we used IBM COS as the storage backend. For the event-based strategy, we used RabbitMQ manually deployed on a virtual machine instance (Ubuntu 20.04, 2 Cores, 4GB RAM) on the IBM Cloud, resulting in a hybrid, non-pure FaaS system.

The results of polling-based and event-based monitoring are depicted in Figs. 2b and 2e, respectively. As can be seen in the figures, the overall execution time decreases from 17 secs to 11.5 secs with event-based monitoring, which yields a reduction of $\approx 32\%$. This is because the monitoring time has been reduced from 6 secs to only 0.3 secs sthrough eventing, which represents an overhead reduction of 95%.

It should be noted that Fig. 2b reports the optimal results for polling-based monitoring. In practice, this test can easily take more time to complete, even surpassing 14 seconds. So, the improvement with event-based monitoring system will presumably be greater in real use cases. Indeed, the major limitation of polling-based monitoring is the fact that objects (i.e., `status.json` files) do not become visible (or listable) at the very moment they have been uploaded. This causes polling-based monitoring to induce significant overhead.

6.3 Elasticity and Concurrency

For this test, we employed a function that runs a compute-bound task for 60 seconds. More concretely, we first checked the correct interaction of all the cloud services involved in Lithops, and then measured its degree of elasticity and concurrency.

We varied the workload by increasing the number of concurrent UDF invocations, from 500 up to 2,000 function executors. Fig. 2c shows the results of the experiment. For every workload, the black line signals the concurrency level delivered at every time instant. The stacked horizontal gray lines represent the total time that each function invocation took to complete.

As it can be easily seen in this figure, some functions ran fast while others slow. Such variability is due to the internal operation of IBM Cloud Functions, the time to spawn all function executors, and the available resources in the cluster. However, we outline that for all the workloads, we obtained full concurrency, i.e., the black line met the target workload size in all the experiments.

Moreover, Fig. 2c verifies that IBM Cloud Functions met the elasticity objective at all times. We ran the experiment for 500, 1,000, 1,500 and 2,000 function invocations. In all scenarios, we observed that the system had no problems to allocate the additional new 500 function executors to keep up with the growing demand of UDF invocations posed by the client.

6.4 Proxied Function Invocation

We evaluated the efficacy of the Lithops proxied invocation mechanism by running two experiments of 1,000 functions each. For simplicity, all functions executed a compute-bound UDF of 50 seconds. In the first test, the Lithops client issued the 1,000 functions locally. In the second test, we enabled the proxied function invoker. The results in Fig. 2f visualize how from a high-latency network, it can take a considerable time to complete the invocation phase when all functions are invoked from a local machine. More specifically, 43% of the 88 seconds that took the first experiment were wasted on function invocation. In contrast, remote spawning was able to reduce the invocation time to just 6 seconds, representing a 6x improvement from our specific high-latency location.

6.5 Applications

We used Lithops to run various applications at scale in the IBM Cloud, each emitting MapReduce jobs.We describe and evaluate them next to show the high utility of Lithops.

6.5.1 *Monte Carlo-Based Stock Prediction*

Stock price forecasting provides a compelling guidance for making decisions in the financial markets today. To show the way that Lithops can handle financial data, we used Monte Carlo simulations to forecast the value of IBM stock prices. As input, we used IBM daily stock prices for years 2014, 2015, and 2016 [28], to make a future prediction of IBM stock prices for the next 1095 days. For the forecast model, we adopted a classic model based on Geometric Brownian Motion [29]. Due to the stochastic nature of this model, our general approach was to run a large number of Monte Carlo simulations of this model to make a correct prediction.

To execute Monte Carlo simulations in parallel, we used Lithops. We first encapsulated the logic to generate a number of forecasts, each predicting a number of days, into a

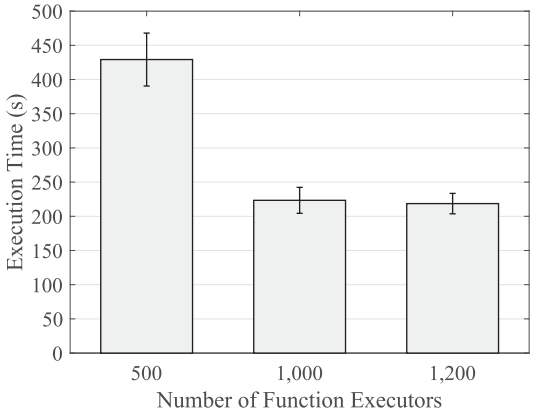


Fig. 3. Effect of the number of function executors for fixed work (100,000 forecasts) on the Monte Carlo-based daily stock price prediction job.

Python function called `process_forecasts()`. Second, we implemented a function called `combine_forecasts()` to summarize the results of the forecasting function. That is, our idea was to execute multiple stock prediction forecasts (`process_forecasts()`), all of them running as separate function invocations. Once all the forecasts were completed, one single reducer (`combine_forecasts()`) was used to aggregate the results from all the invocations and generate the graph with the subsequent predictions.

With only three lines of additional code[4], we managed to perform thousands of Monte Carlo simulations distributed across thousands of concurrent function executors:

Listing 1. Monte Carlo simulations for daily stock price prediction

```
lth = lithops.FunctionExecutor(backend='ibm_cf')
lth.map_reduce (process_forecasts, data=range(1000),
                combine_forecasts)
result = lth.get_result()
```

Fig. 3 shows the execution time of stock price prediction for an increasing number of function executors and 100,000 forecasts. We ran the experiment five times and the collected results were averaged to produce the final plot. Concretely, each bar represents the mean execution time, while the error bars signal the 99^{th} percentile. As shown in the figure, the running time halves when doubling the number of function executors from 500 to 1k. For 1,200 executors, this reduction is not longer linear. The main reason is that some executors become stragglers during the experiment, thus delaying the execution of the single reducer in the system.

Because it was not feasible to make so many forecasts on our laptop, we repeated the same experiment but at smaller scale: predicting daily stock prices for 1,095 days by making only 1,000 forecasts. We used 100 function executors in this case. Again the experiment was run five times. Results are listed in Table 3. As can be seen in the table, we reduced the execution time from 2.56 minutes to just a few seconds, all with close to zero development effort (just three additional lines of code as shown in Listing 1).

4. See implementation details available at: https://github.com/pywren/pywren-ibm-cloud/blob/master/examples/monte_carlo/stock_prediction_monte_carlo_with_PyWren.ipynb

TABLE 3
Daily Stock Price Prediction for 1,095 Days

	Execution time (sec)	Confidence level (99%)
Laptop	154.209	$\pm$3.066
Lithops	20.613	$\pm$2.073

100 function executors were used in Lithops.

6.5.2 Hyperparameter Tuning

It is well-known that the performance of machine learning (ML) models very often depends on a relatively large variety of configuration options, the so-called *hyperparameters*. It is thus crucial that any ML system can effectively optimize its hyperparameters in the quest for better performance. While hyperparameters are model-specific, parameter sweeps are embarrassingly parallel (e.g., random and grid search, etc.), so hyperparameter tuning is indeed a good fit for serverless computing as discussed in [30], [31]. To judge the potential of Lithops for hyperamater tuning, we have targeted a different scenario other than neural networks as in the prior literature [30], [31]. More concretely, we have focused on text classification, i.e., a significant problem of Natural Language Processing (NLP), and more specifically, on how to choose the optimal set of hyperparameters for `fastText` algorithm [32], an state-of-the-art text classifier invented by Facebook. In particular, we demonstrate with this example how simple it is to find optimal hyperparameters at scale with Lithops. Indeed, we found it very easy to develop a small submodule on top of Lithops to perform hyperparameter tuning. In as few as ≈ 40 lines of Python code, this module parallelizes random search, but also the evaluation of each parameter set using k-fold cross-validation. Yet, what is more interesting is the fact that this task is almost fully automated. The user has only to specify the number of hyperparameter sets to sample and the value of k. From that minimal setup, the submodule launches parallel hyperparameter search accordingly.

Datasets. For the experiments, we utilized three datasets, all downloadable from `fastText` website[33]. These are: AG's News, DBPedia and Yelp Review Full, which contain 120k, 560k and 650k train samples labeled into 4, 14 and 5 classes, respectively. For complete details on the datasets, we kindly refer the reader to [34].

Experiment. Our evaluation pipeline consisted of two tests. For each dataset, we first conducted 5-fold cross-validation with all the `fastText` parameters set to default values in our laptop machine. We repeated this experiment ten times, and recorded the total execution time for each experimental run to establish a comprehensive baseline.

As a second experiment, we utilized the Lithops system to perform hyperparameter tuning. In particular, we chose to adjust three hyperparameters. As our goal is to quantify the ability of Lithops to speed up hyperparameter tuning, and not to find the best configuration, any other parameters could have been equally chosen. For reproducibility, Table 4 details them, along with their range of values. In particular, we performed a random search over this three-dimensional parameter space for an increasing number of trials. Thanks to Lithops, each trial of hyperparameters could be run in parallel with the rest; it sufficed a call to `map_reduce()`.

TABLE 4
Chosen `fastText` Hyperparameters

Name	Description	Range	Default
`lr`	learning rate	[0.01, 1]	0.1
`ws`	size of the context window	[3..7]	5
`epoch`	number of epochs	{1, 5, 10}	5

The reduce function was used to collect the precision value of each trial and return the candidate with the highest value.

For each trial, we again ran 5-fold cross-validation. But very interestingly, this time this task was also made parallel with a call to `map_reduce()`. That is, 5 function executors were launched to process the 5 folds in parallel by invoking `train_supervised()`, namely, the `fastText`'s method for supervised classification. The final reduce function was used to gather the precision value from each fold, and thus, produce an average precision value for the whole trial.

It is important to note here that this example shows how easy it is to handle (two-level) nested parallelism and simple compositions with Lithops.

Results. Fig. 4 illustrates the execution time for a growing number of hyperparameter sets h, where $h \in \{5, 10, 20, 40\}$. Error bars indicate the 99^{th} percentile. As can be seen in the figure, the execution time converges to a fixed point as the number of hyperparameter sets increases, irrespective of the dataset. This behavior demonstrates that Lithops scales out efficiently by launching function executors in proportion to the workload. Higher efficiency of Lithops for small values of h is explained by a lesser monitoring overhead, that is, a smaller number of function executors to keep track of. We also note that, although at first glance Lithops could appear to perform poorer than the user laptop, Lithops processed a heavier workload in all the cases as reported in Table 5, thus delivering a significant speedup for all datasets if the same workloads were to be executed on the laptop. The sublinear speedup is due to monitoring overhead (that includes result fetch), the time to spawn the function executors, and the I/O time for reading input data from IBM COS.

6.5.3 Sentiment Analysis

To conclude, we crafted a use case example to demonstrate how Lithops can help to process datasets stored in IBM COS. For this example, we used `www.airbnb.com` data

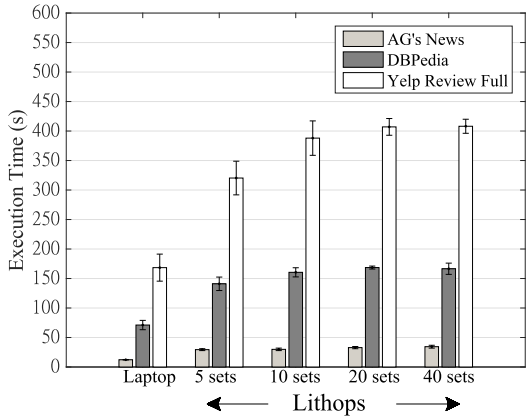


Fig. 4. Execution time against the number of searched hyperparameter sets $h \in \{5, 10, 20, 40\}$ in `fastText`, using 5-fold cross-validation.

TABLE 5
Relative Workload and Speedup Compared to the Laptop (Baseline) Varying the Number of Hyperparameters Sets $h \in \{5, 10, 20, 40\}$

h	Workload increase	Number of executors	Projected speedup		
			AG's News	DBPedia	Yelp Full
5	26.7x	31	11.15x	13.46x	14.04x
10	53.3x	71	21.94x	23.67x	23.18x
20	106.6x	141	39.97x	45.1x	44.24x
40	213.3x	281	76.27x	91.26x	88.2x

from various cities around the world, in conjunction with a tone analyzer, i.e., a linguistic analyzer to uncover emotional and language tones in written text. To make it more appealing, we plotted the results visually on a map.

Datasets. The data was retrieved from IBM Watson Studio Community [35] and then copied to an IBM COS bucket. In particular, there is a dataset per city, which contains all the apartment reviews written by the users. As some cities are more "touristy" than others, the dataset size varies from city to city. The full dataset is made of 33 cities, with a total size of 1.9GB and 3,695,107 comments.

Experiment. We first tested how much time the experiment takes without the concurrency of Lithops. To this aim, we built a Jupyter notebook in IBM Watson Studio to process sequentially all the cities. For the hardware configuration of the VM, we borrowed the same specs of our laptop: 4vCPU with 16GB of RAM. With this setup, it took 1 hour and 26 minutes to serially process all the 3,695,107 comments and render the 33 city maps, which is a significant time burden, especially for impatient users.

Next, we redid the same experiment but with the aid of Lithops. Essentially, we made some cosmetic changes to the code to execute the experiment via the `map_reduce()` call. On the one hand, we created another Jupyter notebook in IBM Watson Studio with the same hardware configuration as in the prior test. On the other hand, we set up the Lithops IBM CF runtime to use 1GB of RAM.

Remember that it is possible to call `map_reduce()` with a specific chunk size. The chunk size determines the final concurrency, so we played out with different chunk sizes to

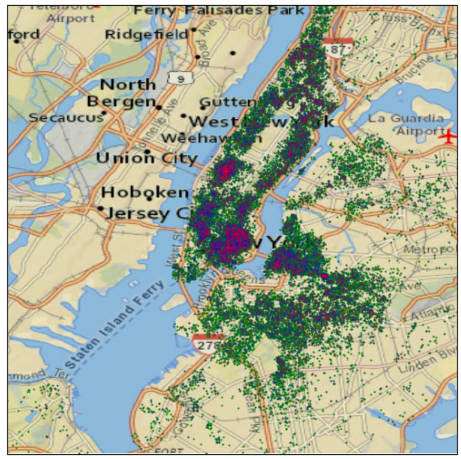


Fig. 5. Tone analysis of the Airbnb reviews of New York City. Green, blue and red points stand for good, neutral, and bad comments, resp.

TABLE 6
Performance of Tone Analysis of Airbnb reviews for Different Chunk Sizes

Chunk size (MB)	Number of executors	Execution time (sec)	Speedup
——-	——-	5160	Baseline
64	47	313.87	16.43x
32	72	196.24	26.29x
16	129	95.48	54.04x
8	242	59.77	86.33x
4	471	35.01	147.38x
2	923	25.58	201.72x

The results are better than in our preliminary work [10] as they have been expressly re-run for this work.

understand how it affects the total execution time. Also, we set `reducer_one_per_object=True` to have a dedicated reducer per city dataset. That is, each reducer collected the partial results from its corresponding city and rendered the final map. An example of a map is depicted in Fig. 5. In this case, it represents the tone analysis of the comments of the City of New York. Each point in the map represents the location of the apartment, and the color of the point signals the tone of the comments.

Results. In Table 6, we report the results of this experiment. A first key observation to be made is that Lithops achieved excellent speedups, greater than 100x. This proves the huge benefit of Lithops to non-cloud users, who can readily leverage the large number of CPU cycles available in the cloud, with no need to struggle with hardware management and specialized stacks such as Spark and MPI. In practice, although Lithops exhibits some overhead, users would not care as much about parallel efficiency, but more about the savings in compute times with close to zero devops cost.

Notice that the number of function executors does not duplicate when halving the chunk size. This occurs because partitioning takes place within each dataset file. Either way, the achieved parallel execution time is proportional to the number of function executors, growing between 16.43x and 201.72x for chunks of 64MB and 2MB, respectively.

7 Conclusion

In this work, we have proposed Lithops, a novel serverless platform for executing parallel tasks *à la MapReduce* on the most popular cloud providers (e.g., IBM, AWS and Google). With close to zero overhead to cater for "untrained" users of the cloud, Lithops comes along with a useful set of features, including automated data discovery & partitioning, nested composability, seamless integration with Jupyter notebooks, etc. We have thoroughly described all these valuable assets. Using several scientific programs, namely, stock prediction, hyperparameter tuning, and tone analysis, we have assessed Lithops performance. Compared with a *commodity* machine, Lithops has yielded important speedups, even larger than 100x, without the need of a *warm cluster running continuously* in the cloud.

Acknowledgments

The authors would like to thank Ohad Zohar for his valuable collaboration in the experiments.

References

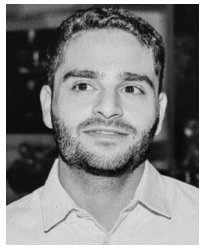
[1] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau , and R. H. Arpaci-Dusseau , "Serverless computation with openLambda," in *Proc. USENIX Workshop Hot Topics Cloud Comput.*, 2016, pp. 33–39.
[2] S. Fouladi *et al.*, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 363—376.
[3] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: distributed computing for the 99%," in *Proc. ACM Symp. Cloud Comput.*, 2017, pp. 445–451.
[4] J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París, "Data-driven serverless functions for object storage," in *Proc. ACM/IFIP/USENIX Middleware Conf.*, 2017, pp. 121–133.
[5] I. Baldini *et al.*, "Serverless computing: Current trends and open problems," in *Proc. Res. Adv. Cloud Comput.*, 2017, pp. 1–20.
[6] V. Shankar *et al.*, "numpywren: Serverless linear algebra," *CoRR, abs/1810.09679*, 2018.
[7] S. Fouladi *et al.*, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *Proc. USENIX Annu. Technical Conf.*, Jul. 2019, pp. 475–488.
[8] S. Fouladi *et al.*, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 363–376.
[9] "Lithops," [Online]. Available: https://github.com/lithops-cloud/lithops
[10] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the ibm cloud," in *Proc. ACM/IFIP Middleware Conf. Ind.*, 2018, pp. 1–7.
[11] T. White, *Hadoop: The Definitive Guide*, 1st ed. Newton, MA, USA: O'Reilly Media, Inc., 2009.
[12] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2012, p. 2.
[13] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 193–206.
[14] *AWS*, "Elasticache for redis," [Online]. Available: https://aws.amazon.com/elasticache/redis/
[15] M. Sánchez-Artigas, G. T. Eizaguirre, G. Vernik, L. Stuart, and P. García-López, "Primula: A practical shuffle/sort operator for serverless computing," in *Proc. Int. Middleware Conf. Ind. Track*, 2020, pp. 31–37.
[16] *Qubole*, "Spark on lambda," [Online]. Available: https://github.com/qubole/spark-on-lambda
[17] Y. Kim and J. Lin, "Serverless data analytics with flint," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2018, pp. 451–455.
[18] J. M. Hellerstein *et al.*, "Serverless computing: One step forward, two steps back," in *Proc. Biennial Conf. Innovative Data Syst. Res.*, 2019.
[19] B. Congdon, [Online]. Available: "Corral a MapReduce framework," https://github.com/bcongdon/corral
[20] J. Spillner, "Lambada," [Online]. Available: https://gitlab.com/josefspillner/lambada
[21] "Pulsar functions overview," [Online]. Available: https://pulsar.apache.org/docs/en/functions-overview/
[22] D. Barcelona-Pons , M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the FaaS track: Building stateful distributed applications with serverless architectures," in *Proc. Int. Middleware Conf.*, 2019, pp. 41–54.
[23] *Python*, "concurrent.futures," [Online]. Available: https://docs.python.org/3/library/concurrent.futures.html
[24] *jupyter*, "Python notebooks," [Online]. Available: https://jupyter.org/
[25] *IBM*, "Watson studio." Accessed: Nov. 25, 2021. [Online]. Available: https://www.ibm.com/cloud/watson-studio
[26] P. García-López, M. Sánchez-Artigas, G. París, D. Barcelona, Á. Ruiz, and D. Arroyo, "Comparison of FaaS orchestration systems," in *Proc. Int. Workshop Serverless Comput.*, 2018, pp. 148–153.
[27] *Amazon*, "Step functions," [Online]. Available: https://aws.amazon.com/step-functions/
[28] *IBM*, "Historical stock data," [Online]. Available: https://www.nasdaq.com/market-activity/stocks/ibm/historical
[29] P. Glasserman, *Monte Carlo Methods in Financial Engineering*. New York, NY, USA: Springer, 2004.
[30] M. Zhang, C. Krintz, M. Mock, and R. Wolski, "Seneca: Fast and low cost hyperparameter search for machine learning models," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2019, pp. 404–408.

[31] L. Feng, P. Kudva, D. D. Silva, and J. Hu, "Exploring serverless computing for neural network training," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2018, pp. 334–341.
[32] E. Grave, T. Mikolov, A. Joulin, and P. Bojanowski, "Bag of tricks for efficient text classification," in *Proc. Conf. Eur. Chapter Assoc. Comput. Linguistics*, 2017, pp. 427–431.
[33] *Fasttext*, "Supervised models," [Online]. Available: https://fasttext.cc/docs/en/supervised-models.html
[34] X. Zhang, J. Zhao, and Y. LeCun , "Character-level convolutional networks for text classification," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 649–657.
[35] *IBM*, "Watson studio gallery," [Online]. Available: https://dataplatform.cloud.ibm.com/gallery

Josep Sampé received the PhD degree in computer engineering from Universitat Rovira i Virgili, Spain, in 2018. At the end of his PhD studies, he became a key contributor to the Lithops core during an internship with IBM Research Haifa. He is currently a postdoctoral fellow with the Universitat Rovira i Virgili. He has authored or coauthored several articles in important conferences such as IMC, Middleware, and USENIX FAST, among others, and has participated in several European H2020 research projects such as IOStack and CloudButton. His research interests include distributed systems, cloud computing, data analytics, and software engineering.

Marc Sánchez-Artigas received the PhD degree in computer science from the Universitat Pompeu Fabra, Spain, in 2009. During his PhD studies, he worked with École Polytechnique Fédérale de Lausanne (EPFL). In the same year he joined the Universitat Rovira i Virgili, where he currently works as an associate professor. He has authored or coauthored more than 80 articles in important venues such as IEEE P2P, Middleware, ICDCS, and USENIX FAST. He is currently involved in the coordination of several Spanish projects in cloud computing. He has participated in several European research projects such as FP7 CloudSpaces, H2020 IOStack, and H2020 CloudButton. He was the recipient of Best Paper Award from IEEE LCN'07 and the Best Dataset Award from ACM IMC'15.

Gil Vernik received the PhD degree from the University of Haifa in 2008. He held a postdoctoral position in Germany. He is currently a senior architect, technical and a team leader with IBM Research Haifa. He is an expert in Big Data, serverless computing, HPC over cloud computing, object storage, and general distributed execution frameworks. He is very active in open source projects, where he has led projects such as Stocator and Lithops. He is also an expert in web technologies and architectures, data reduction techniques, and both relational and non-relational databases. He holds more than 12 years position as a lecturer with the University of Haifa, where he has taught several courses on Big Data and cloud computing within the CS Department.



Ido Yehekzel received the BSc degree in computer engineering with Technion - Israel Institute of Technology, Haifa. His research interests include parallel and distributed computing, and machine learning. He is one of the active contributors to Lithops.

Pedro García-López (Member, IEEE) is currently a full professor with the Universitat Rovira i Virgili. He leads the CloudLab research group and has coordinated three large european research projects in the last few years: FP7 CloudSpaces from 2013 to 2015, H2020 IOStack from 2015 to 2017, and H2020 CloudButton from 2019 to 2021. He has authored or coauthored more than 100 papers on journals and prestigious conferences such as Middleware, ICDCS, USENIX FAST, ICDE, and IMC. His research interests include distributed systems, cloud storage, software architectures, and middleware. He has participated in scientific committees as Steering Committee member of IEEE P2P, General Chair of IEEE P2P, PC member of IEEE P2P, CCGRID, CLOSER, or WETICE.