Serverless Data Analytics in the IBM Cloud

Josep Sampé*
Universitat Rovira i Virgili
Tarragona, Spain
josep.sampe@urv.cat

Gil Vernik
IBM
Haifa, Israel
gilv@il.ibm.com

Marc Sánchez-Artigas,
Pedro García-López
Universitat Rovira i Virgili
Tarragona, Spain
{marc.sanchez,pedro.garcia}@urv.cat

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. Inspired by this vision, this industry paper presents *IBM-PyWren*, which continues the pioneering work begun by PyWren in this field. It must be noted that *IBM-PyWren* is not, however, just a mere reimplementation of PyWren's API atop IBM Cloud Functions. Rather, it is must be viewed as an advanced extension of PyWren to run broader MapReduce jobs. We describe the design, innovative features (API extensions, data discovering & partitioning, composability, etc.) and performance of *IBM-PyWren*, along with the challenges encountered during its implementation.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Theory of computation** → *Distributed computing models*;

KEYWORDS

Distributed computing, Serverless computing, IBM Cloud Functions, IBM Cloud Object Storage, PyWren

ACM Reference Format:
Josep Sampé, Gil Vernik, and Marc Sánchez-Artigas, Pedro García-López. 2018. Serverless Data Analytics in the IBM Cloud. In *Middleware Industry '18: ACM/IFIP/USENIX Middleware Conference (Industrial Track), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3284028.3284029

1 INTRODUCTION

Serverless computing, also known as "*functions-as-a-service*" (FaaS), has become a hot topic in the cloud world today. 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 conferences and articles dedicated to this subject [2, 5, 6, 12, 17]. The shift from the server to the task level, the ability to scale to thousands of concurrent functions, and sub-second billing have spurred many users to embrace serverless computing for a variety of applications (e.g., microservices, IoT, ML), some of them somewhat far from the original intents of serverless architectures.

One of these apparently"anti-natural" intents has been the use of serverless computing for massive-scale distributed computing. In 2017, Eric Jonas et al. from RISELab@ Berkeley built a prototype called *PyWren* [12], and showed how a serverless execution model with stateless functions can substantially lower the barrier for non-expert users to leverage the cloud for massively parallel workloads. By skipping the need for complex cluster management and using a simple Pythonic futures-based interface, PyWren allows users' non-optimized code to run on thousands of cores using AWS Lambda.

Inspired by this new radical vision, and as joint work between IBM Research Haifa Lab and URV, we decided to build a serverless parallel framework similar to PyWren but for IBM Cloud. We called it "*IBM-PyWren*", because it takes the tack of PyWren by extending it with a number of new features, including a broader MapReduce support, automatic data discovery and data partitioning, dynamic function composability (e.g., sequences and nested parallelism), etc. In Table 1, we list the main differences and the new features added to PyWren. *IBM-PyWren*'s source code is available on PyWren's Github repository [13], maintained by RISELab@ Berkeley.

Table 1: Differences between PyWren [12] and IBM-PyWren.

	PyWren	*IBM-PyWren*
MapReduce	*Mapping portion; reducing is still experimental.*	*Broader support for MapReduce jobs. Also, it includes a* `reduceByKey`*-like operation to run one reducer per object key in IBM COS 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:* $f_3 = f_2 \circ f_1$., *nested parallelism (*`mergersort`*).*
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 (e.g., PyWren equipped with* `matplotlib`*).*
Remote function spawning	*Due to network overhead and AWS throttling, it may be slow to launch jobs with thousands of functions.*	*Faster; client calls a remote invoker function, which starts all functions in parallel within the cloud.*
Open source portability	*Adapted to work with Amazon Lambda*	*Can work with Apache OpenWhisk*

*This work was done as part of a research internship at IBM Haifa Research Labs.

Middleware Industry '18, December 10–14, 2018, Rennes, France

ACM ISBN 978-1-4503-6016-6/18/12…$15.00
https://doi.org/10.1145/3284028.3284029

Contributions. Here we present the design and implementation of *IBM-PyWren*, i.e., IBM's open-source serverless framework for data analytics [13], which has been successfully tested over IBM Cloud [11], i.e., IBM's cloud computing platform. We detail the possibilities that it provides for data analytics, alongside the challenges encountered along the way. Besides, we evaluate its performance, the elasticity and concurrency it supports, eventually showing the potential of this framework with a real use case example.

2 RELATED WORK

Our approach shares the same spirit of PyWren [12], which is to provide users with a simple interface to run their optimized, single-machine code in parallel, while offloading cluster management to the cloud. Concretely, PyWren enables the automatic conversion of Python functions into massively parallel `map` tasks running as AWS Lambda functions. IBM-PyWren enhances PyWren's functionality, and allows to execute a broader scope of MapReduce jobs over IBM Cloud Functions, with automatic data discovery and partitioning. In addition, it is able to manage simple workflows (e.g., sequential compositions) and optimize the spawning of thousands functions to 8 seconds (In PyWren, function invocation can take up to 20-30 seconds due to AWS throttling [12]). A detailed description of the new features are reported in Table 1.

Further, several recent attempts have been made to implement MapReduce with serverless technology. Works such as Qubole [16] and Flint [15] have tried to enable Apache Spark to run over AWS Lambda. For instance, Qubole creates an execution DAG on the client side and runs each task as a serverless function. In addition to the users' difficulty to learn Spark 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.

Other works have tried to implement a MapReduce serverless framework from scratch such as [3] and Lambada [18], which are still under active development. Many open issues remains in these systems, where data shuffling remains one of the biggest challenges in running MapReduce jobs over serverless architectures. Several proposals have been made on how to make shuffle more efficient by using Amazon ElasticCache, S3, Amazon SQS, among others.

A final word on commercial serverless platforms: Among the big cloud providers, only Microsoft and IBM provide open source solutions (IBM Cloud Functions are based on Apache OpenWhisk). The rest are proprietary solutions such AWS Lambda and Google Cloud Functions. Hence, it extremely hard to optimize, or even overtly impossible to implement tools like ours over these proprietary platforms [4]. This give us advantage over competitors. As all the software stack is open-source in IBM-PyWren, it is thus possible for practitioners to adapt the whole system to the evolving needs of their organizations.

3 ARCHITECTURE

The overall architecture of IBM-PyWren is depicted in Fig. 1. At a high level, it uses the IBM Cloud Functions service to run serverless actions as MapReduce tasks, compositions of functions, etc., and IBM Cloud Object Storage (IBM COS) to store all data, including intermediate results (e.g., output of a `map` task) on the server side.

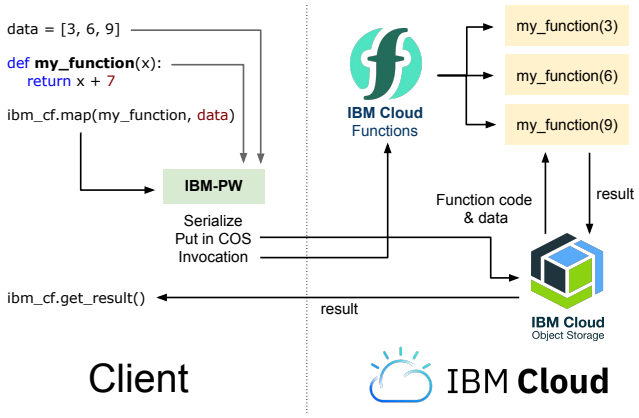

Figure 1: High-level IBM-PyWren execution flow example.

Very succinctly, IBM Cloud Functions, which is based on Apache OpenWhisk, is a FaaS platform for running functions in response to events. At the time of this writing[1], the IBM Cloud Functions service limits function execution to 600 seconds, 512MB of RAM per function execution, and a maximum 1,000 concurrent invocations, though the number of concurrent functions can be increased if needed. On the other hand, IBM COS is an unstructured data storage service designed for durability, resiliency and security.

Also, the platform includes a client, which can run on a laptop or in the cloud, responsible for taking user's Python code and relevant data, and saving them into IBM COS, among other things.

Fig. 1 illustrates the interactions occurring in a typical execution flow: 1) The client takes a user's code (e.g., `my_function()`) and input data (e.g., `data = [3, 6, 9]`), serializes them and finally stores them into IBM COS; 2) Then, it invokes the function (e.g., `my_function()`) through IBM Cloud Functions; 3) On the server side, the IBM Cloud Functions framework gets the user's code and the data from IBM COS, executes the function in parallel (one for each piece of data), storing the results back to COS; 4) Finally, the results are pulled by the client straightway when they are ready. This simplified architecture is enough to run full MapReduce jobs with acceptable performance.

3.1 Runtime

Non-surprisingly, IBM-PyWren runtime is built upon IBM Cloud Functions runtime, which is based on Docker images. The default Python runtime is called *python-jessie:3* [10], which includes the most common python packages [7], so that data scientists can start using IBM-PyWren right away.

An important feature, that is different from the other serverless MapReduce frameworks based on AWS Lambda, is that IBM allows users to create their own custom runtimes. That is, a user can build a Docker image with the required packages, either Python libraries or system libraries and binaries, and then upload the image to the Docker hub registry, where the IBM Cloud Functions service will get it to run the functions.

[1] IBM Cloud Functions is evolving rapidly: default values can change while this paper is undergoing review.

IBM Cloud functions gets the self-built Docker container image from the Docker registry the first time you invoke a function with the specific runtime. Then, the Docker container is cached in an internal registry.

Further, this approach of storing the self-built runtimes in the public Docker hub registry enables the possibility to share them among other users. For example, a user may create a Docker image with the package `matplotlib`, a library to create 2D figures, and share that image with other colleagues via the Docker hub registry, thus avoiding them the overhead to create the same runtime with this specific library.

4 PROGRAMMING MODEL

One core principle behind IBM-PyWren is programming simplicity. As in the pioneering work of Jonas et al. [12], our focus 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 such as the `map_reduce()` method to offer a broader MapReduce support.

Also, we have devoted extra efforts to integrate IBM-PyWren with other tools (e.g., Python notebooks such as Jupyter [14]), which are very popular environments for the scientific community. Python notebooks are interactive computational environments, in which you can combine code execution, rich text, mathematics, plots and rich media. To this aim, IBM Cloud contains a service called IBM Watson Studio [8], that, among other things, allows to create and execute notebooks in the cloud, where IBM-PyWren can be very easily imported to run embarrassingly parallel jobs.

4.1 Executor

The first citizen object in IBM-PyWren is an *executor*. This object allows to perform calls to IBM-PyWren's API to run parallel tasks. The standard way to get everything set up is to import the module `pywren_ibm_cloud`, and call the function `ibm_cf_executor()` to get an instance of the executor:

```
exec = pw.ibm_cf_executor()
```

Once an instance of the executor is created, an executor ID is generated in order to track function invocations and the results stored in IBM COS. Each executor instance will generate a unique executor ID. The executor also loads the configuration required to grant access to both IBM services (e.g., account details). By default, the executor uses the runtime specified in the configuration file. However, it is possible to specify the desired runtime to use when a executor instance is created. This property allows to use different runtimes in different executor instances in the same client's code:

```
exec = pw.ibm_cf_executor(runtime='matplotlib')
```

4.2 Application Programming Interface (API)

The executor instance provides access to all the available methods in the API. The API is comprehensive enough for novice users to execute algorithms out of the box, but simple enough for experts to easily tune the system by adjusting important knobs and switches (parameters). The API is described in Table 2. There are three main methods to execute the user's code trough IBM Cloud Functions in addition to PyWren's `map()` function. Also, there are two different methods to get the results: `wait()` and `get_result()`.

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.	when to unlock, list of *futures*
`get_result()`	Sync.	*None*

When any of the computing methods (i.e., `call_async()`, `map()`, and `map_reduce()`) are called, both the code to run as a function and the input data are first serialized and stored in IBM COS. Next, the platform transparently interacts with the IBM Cloud Functions service to execute the tasks. This requires unserializing the code and data, and run the code as a function executor. Once execution concludes, the results and some metadata about the status of the invocations, such as execution times, are stored back in IBM COS.

To cater for average users, the API has been utterly simplified, so that the typical behavior of calling a computing method (e.g., `map()`) followed by a call to `get_result()` to retrieve the results does not require dealing with complex artifacts. Indeed, the readiness of the results is internally managed by IBM-PyWren (see the listing in the description of the `map()` method). To deliver a finer control, all the three computing methods, however, also return *future*[2] objects to track the status of the executors and get the results when available.

Let us review succinctly the main methods of the API

▹ **`call_async()`**. The first method is used to run asynchronously just one function in the cloud. The result are held in IBM COS. This method is non-blocking, i.e., the sequential execution of the local code continues without waiting for the results. The parameters of this method are the `function_code` and the input `data` that the function executor receives.

▹ **`map()`**. The second method is called `map()`. This method is used to run multiple function executors. 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, one function executor is initiated to process them 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]
exec = pw.ibm_cf_executor()
exec.map(my_map_function, input_data)
result = exec.get_result()
```

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

▹ **`map_reduce()`**. The third method is used to execute 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()`**. On the client side, the API offers two different methods to pull the results from IBM COS. The first one is called `wait()`. It is synchronous, i.e., the local user code is blocked until the call to `wait()` ends. It provides a configurable parameter to decide when to release the call and continue the execution. A user might decide to unlock the method in three different circumstances: 1) '`Always`': it checks whether or not the results are available on the invocation of `wait()`. If so, it returns them. Otherwise, it resumes execution; 2) '`Any completed`': it resumes execution upon termination of any function invocation. The available results are given as a response to the call; and 3) '`All completed`': it waits until all the results are available in IBM COS. It returns a 2-tuple of lists: the first list with the futures that completed and the second with the uncompleted ones.

▹ **`get_result()`**. This method supersedes the `get_all_results()` method in PyWren's API to collect the results from IBM COS when a parallel task has finished (e.g., `map()`, `map_reduce()`, etc.). It adds new functionality such as timeout support, keyboard interruption to cancel the retrieval of results, and a progress bar to inform users about the % of task completion. 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.

4.3 Data discovery and partitioning in `map_reduce()`

To provide good MapReduce support, an important key ingredient is to provide a built-in data partitioner in the platform to abstract users from this arduous, prone to error task. Of course, this support has been given to the `map_reduce()` method, along with a useful data discovery mechanism.

In particular, the only action that a user must do is to supply the list of object keys that compose the dataset. However, as a dataset may contain hundreds, or even thousands of files, it is possible to specify the name of the IBM COS bucket(s) that contain all the objects in the dataset instead. In the latter case, the framework is responsible for discovering all the objects in the bucket(s), and partition them.

The data discovery process is automatically started when a bucket is specified instead of a list of objects. It consists of a HEAD request over each bucket to obtain the necessary information to create the required data for the execution.

Once the data discovery process has ended, the data partitioner is initiated to produce the data partitions based on a configurable chunk size parameter. 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 IBM COS service. The partitioner then executes the reduce function. The reduce function will wait for all the partial results before processing them.

Additionally, IBM-PyWren allows to have results computed by multiple reducers. By default, the `map_reduce()` method uses a single reducer for all the partitions of the dataset. 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 values for the same object key in IBM COS are combined in 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 in Section 6.3).

4.4 Composability

A novel feature of IBM-PyWren is *function composition*. Although complex function composition still remains an open issue [2], IBM-PyWren yet allows a certain level of composability. It is worthy to note that function composability is achieved *programatically*, and not *declaratively* like in AWS Step Functions [1], where function composition is realized writing state machines in declarative JSON. A programmatic approach is *a priori* more powerful, since we have available all the control flow instructions from Python to compose workflows.

In commercial FaaS orchestration systems [**?**], functions that are part of a composition need to be first deployed to the platform before being called. In contrast, IBM-PyWren reduces this "boring" task to adding two lines of code (e.g., a call to the `ibm_cf_executor()` method to get an executor, followed by a call to the `map()` method for parallel processing). That is, any regular Python function can be run with IBM-PyWren without its predeployment as a standalone function in the serverless platform. This fact enables the *dynamic* and *parallel* composability of functions.

To illustrate, consider a simple *parallel* composition of functions: A user invokes a first function, say `foo()`, via `call_async()`. While doing some processing, this function might dynamically generate 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 few lines, it is possible to create a dynamic composition of functions:

```
def add_seven(y)
    return y + 7

def foo():
 # do some processing
 rlist = random_list(100)
 exec = pw.ibm_cf_executor()
 return exec.map(add_seven, rlist)

exec = pw.ibm_cf_executor()
exec.call_async(foo)
result = exec.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 functions call the next function in the sequence via the `call_async()` method. Finally, the user receives the result of the last function in the sequence.

▹ **Nested parallelism.** For instance, recursive algorithms, where each level of recursion has parallelism such as `mergesort`, can also be executed in parallel with IBM-PyWren. These algorithms can be actually parallelized at any recursion level. However, most of these algorithms operate on a single element, or just a few, in the deepest recursion levels, thereby finishing quickly. In order to amortize the overhead of function spawning, it is better off to execute part of the tree of recursive calls within each function, instead of mapping each recursive call to a new function. For example, the `mergesort` algorithm creates a complete binary tree of depth $O(\lg N)$ to sort an array of N numbers. That is, sorting 1,000 numbers generates a tree of 10 levels. In this case, it would be wiser to spawn a new parallel function, for example, every 5 recursive iterations of the binary tree, in order to minimize the overhead.

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 vital to the success of IBM-PyWren, and any serverless parallel cloud framework. Both properties are important to allow a quick execution of data-parallel tasks. Fortunately, IBM Cloud Functions can provision a large amount of compute power quickly, and thus, perform "big data"-styled analysis with good performance.

With IBM-PyWren, it is now easy to handle bursty workloads that require thousands of concurrent function executors without, waiting for machines to spin up. Serverless platforms such as IBM Cloud Functions are based on containers which are fast to boot up. Consequently, function executors can be up within a sub-second range, right after their corresponding invocations.

5.1 Massive Function Spawning

When playing out with IBM-PyWren, we encountered some issues in the function invocation phase due to network latency. When the client is located in a remote network (e.g., WAN), we observed that a high network latency between the client and the data center can significantly impact the total invocation time. For instance, while the invocation of 1,000 functions from a low-latency network (e.g., IBM Internal Network) takes around 8s, invocation time can grow up to 40s in a high-latency network, despite leveraging threading to concurrently spawn the functions. Moreover, a higher latency also turns into more invocation failures, which further increase the total invocation time due to the retries in the failed invocations.

To overcome these issues, we designed the *massive function spawning* mechanism, which can be enabled and disabled as needed by the user. Based on the composition feature of IBM-PyWren (see Section 4.4), this mechanism consists of launching one function from the client, i.e., the *remote invoker function*, and use it to spawn the target number of function executors. As the invoker function is located inside the IBM Cloud Functions cluster, the invocation latency is the lowest possible, thus reducing the total invocation time and the number of invocation failures. Unfortunately, with this mechanism the invocation time only diminished to around 20s, which is not good enough compared with the 8s of IBM's cloud internal network.

Experimenting with our framework and IBM Cloud Functions, we inferred that the best way to make the invocations faster was to optimize for the level of concurrency that IBM Cloud Functions itself allows. To reduce significantly the invocation time, we finally improved the remote invoker function. The final approach was to make groups of 100 invocations and execute them at the same time with different remote invoker functions. For example, pretend that it is necessary to execute 1,000 functions. In this case, the *massive function spawning* mechanism would internally arrange 10 groups of 100 invocations, which would be processed by 10 remote invoker functions separately.

With this mechanism, we obtained times like if the client was located in a low-latency network, i.e., 8 seconds. The full evaluation and the results are in Section 6.1.

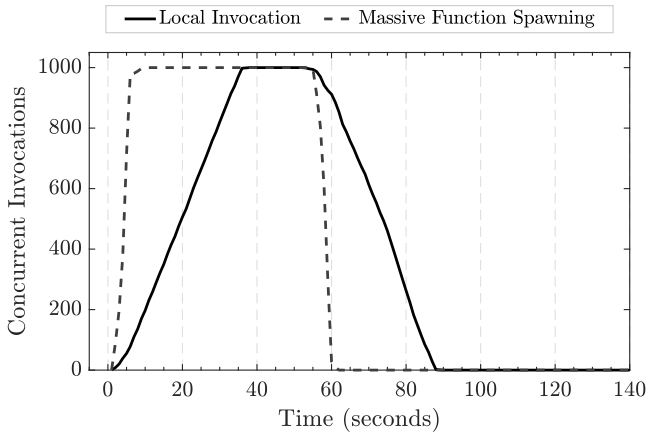

Figure 2: Local invocation vs Massive Function Spawning. The 'y' axes shows the total concurrent invocations in each moment of the experiment represented by the 'x' axes.

6 EXPERIMENTAL EVALUATION

We evaluated IBM-PyWren in different aspects: concurrency and elasticity, its massive function spawning mechanism, and finally, its performance in a real MapReduce scenario. Regarding IBM Cloud services we used the US-south region (Dallas, Texas). We used as a client machine an Intel Core i5, 8GB RAM with Ubuntu 16.04, located in a remote network with high latency.

6.1 Massive Function Spawning

We evaluated massive spawning by running two tests that realized 1,000 function invocations. Each function performed an arbitrary compute-bound task of 50-seconds duration. In the first test, IBM-PyWren's client issued the 1,000 functions locally, whereas in the second text, we enabled massive spawning. The results depicted in Fig. 2 show how from a high-latency network, it took 38 seconds to complete the invocation phase when all functions are invoked from the local machine (i.e., the whole experiment completed in 88 seconds). In contrast, by using the massive function spawning mechanism, it took only 8 seconds to have the 1,000 functions up and running (i.e., the total experiment lasted for 58 seconds). So, we obtained 5X faster invocation times with this mechanism.

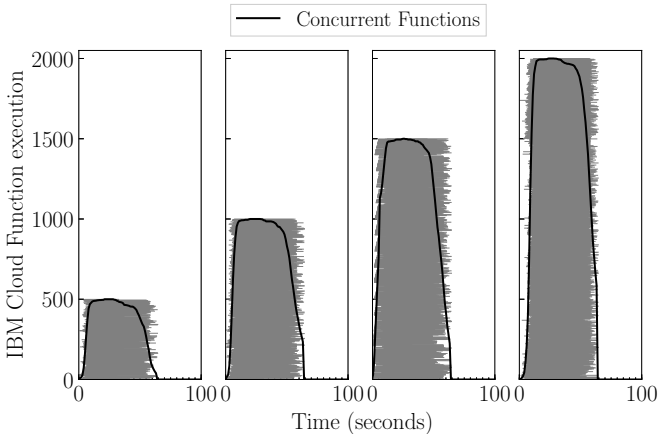

Figure 3: Elasticity and Concurrency. Black lines show total concurrent functions in each moment of the experiment. Each horizontal gray line represents a function execution.

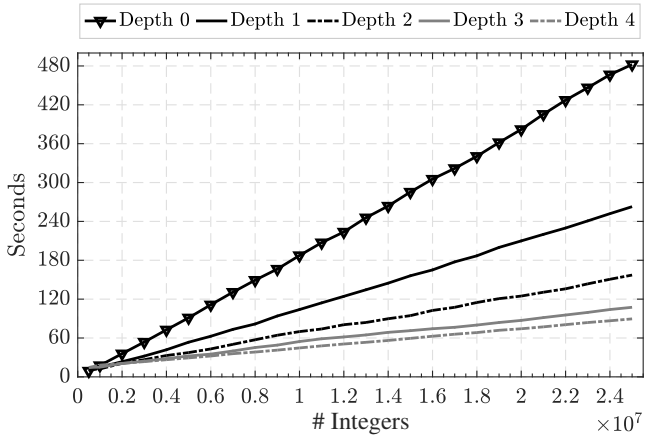

Figure 4: Dynamic composition (`mergesort` algorithm). Execution time for sorting an array as function of the depth of function recursion tree.

6.2 Elasticity and Concurrency

For this experiment, we employed a function that runs a compute-bound task for around 60 seconds. To reduce invocation overhead, we enabled the massive function spawning mechanism. Here, we verified the correct interaction of all the services involved in our framework, and finally we measured the degree of IBM-PyWren's elasticity and concurrency.

We varied the workload by increasing the number of concurrent function invocations, from 500 up to 2,000 function executors. Fig. 3 shows the results of the experiment. For every workload, the black line indicates the concurrency level delivered at every instant of time. 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.

Regarding elasticity, Fig. 3 shows 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. And in none of them, the system had problems to spawn the additional new 500 function executors necessary to deal with the growing demand of function invocations posed by the IBM-PyWren's client.

6.3 Dynamic Composability

As an example of dynamic composition, we ran the `mergesort` algorithm over different arrays of N integer numbers. As stated in Section 4.4, it is not efficient to spawn a new function for each recursive iteration. For this reason, we refactored the `mergesort` algorithm to process multiple recursive iterations per function. Note that as a result of this optimization, the binary tree of function invocations became much shorter than the original `mergesort` recursion tree. Indeed, to control the number of recursive iterations per parallel function, we made use of the depth d of the resultant function tree. For example, for sorting 1M numbers with depth $d = 1$, our code spawned a new function every 500K recursive iterations. This means that the parent (root) function spawned 2 parallel functions, with each child function performing 500K iterations. In the case of depth $d = 2$, the root function spawned 2 new child functions with 500K iterations assigned to each one. Further, every grandchild function was assigned 250K iterations from its corresponding parent. Given a depth d, the total number of recursive iterations to be processed by level-d functions in parallel was of $N/2^d$, respectively.

We varied the workload by increasing the total numbers to sort, N (i.e., array length). We ran this test with lengths N between 500K and 25M integer numbers. To benefit from our parallel composition, we tested different tree function depths. Fig. **??** shows the result of the experiment for depths $d = 0, 1, \ldots, 4$. Each line reports the total execution time in seconds (y-axis) needed to order N integers (x-axis).

This figure shows how the sort time increases linearly in all cases. A greater function depth d is better when the workload size is higher. This is because the overheads related to launching a new level of parallel functions became significant in comparison to `mergesort` time due to the smaller number of iterations per function. Moreover, we can see how the major improvements came from depths up to $d = 3$. Beyond that, the degree of improvement was lower due to the associated IBM-PyWren overheads needed to spawn new functions. In any case, computational-intensive workloads can benefit of this kind of composition by recursively invoking the function itself, thus taking advantage of nested parallelism.

6.4 Real MapReduce Job

We ran a use case example to demonstrate how IBM-PyWren can help to process datasets stored in IBM COS. For this example, we used `www.airbnb.com` data from various cities around the world, in conjunction with a tone analyzer — linguistic analysis to detect emotional and language tones in written text. Then, we plotted the results visually on a map.

To proceed for the analysis, we obtained the datasets from IBM Watson Studio Community [9], and we copied them to an IBM COS bucket. Each dataset represents a city, and it contains the reviews of the apartments wrote by the users. The full dataset is composed of 33 cities. Each city dataset has variable size. The total dataset size is of 1.9GB with a total of 3,695,107 comments.

We first tested how much time the experiment takes without the concurrency of IBM-PyWren. We created a Python notebook in the IBM Watson Studio platform, and processed each city sequentially, one after another. The hardware configuration of the VM for this experiment was 4vCPU with 16GB of RAM. It took 1 hour and 26 minutes to process all the 3,695,107 comments and render the 33 city maps sequentially.

Next, we did the same to evaluate the benefits of our serverless-styled approach. We used the `map_reduce()` method and enabled massive function spawning to speed up the invocation phase. More specifically, we created another Python notebook in IBM Watson Studio with the same hardware configuration as in the prior test. As seen in Section 4.3, it is possible to configure the `map_reduce()` method with a chunk size to partition the datasets. The chunk size determines the final concurrency, so we played out with different chunk sizes to 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 city and rendered the final map. An example of a map is depicted in Fig. 4. 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.

Table 3: MapReduce job execution results. Airbnb example execution times, number of concurrent function executors for the different chunk sizes, and obtained speedups.

Chunk Size	Concurrency	Exec. Time	Speedup
No / Sequential	0 executors	5160 seconds	0 (base)
64MB	47 executors	471 seconds	10.95x
32MB	72 executors	297 seconds	17.37x
16MB	129 executors	181 seconds	28.51x
8MB	242 executors	112 seconds	46.07x
4MB	471 executors	63 seconds	81.90x
2MB	923 executors	38 seconds	135.79x

We executed this test for the following chunk sizes: 64MB, 32MB, 16MB, 8Mb, 4MB and 2MB, respectively. Table 3 shows the results of the experiments. Compared with the sequential execution, IBM-PyWren achieved excellent speedups, greater than 100X. This shows how significantly IBM Cloud Functions can improve the times to process large datasets. Despite there is some overhead (e.g., for an speedup of 10.95X, it is necessary to use 47 executors), users do not care whether the parallel implementation is indeed 100% efficient, because they can get large speedups without the need to manage a cluster.

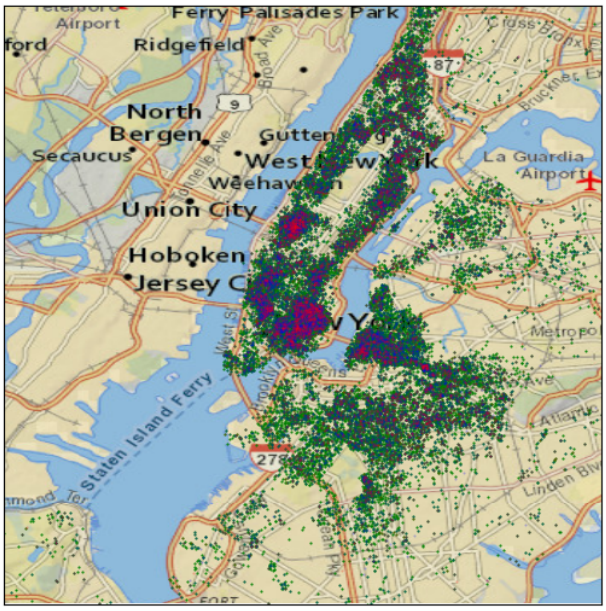

Figure 5: Tone analysis of the airbnb reviews of the city of New York. Green points are good comments, blue points are neutral comments and red points are bad comments.

Note that the number of concurrent function executors does not increase linearly by 2X, as we decrease the chunk size to the half. This is because data partitioning is done for each separate object of the dataset. Despite this, the achieved parallel execution time was directly proportional to the number of function executors, growing between 10.95X and 135.79X for chunks of 64MB and 2MB, resp.

7 CONCLUSIONS

In this research, we have presented IBM-PyWren, a novel serverless platform for executing massively parallel tasks in the IBM Cloud. As new features, it includes a broader support for MapReduce jobs, data discovery and partitioning, dynamic composability, simple integration with Python notebooks, etc. We have described all the new features and evaluated the performance of IBM-PyWren in the IBM Cloud, including a real MapReduce use case. The results are encouraging and are a new proof of the potentials of serverless platforms for distributed computing with speedups > 100X.

ACKNOWLEDGMENTS

This work has been partially supported by the Spanish government TIN2016-77836-C2-1-R.

REFERENCES

[1] Amazon. 2016. Step Functions. https://aws.amazon.com/step-functions/.
[2] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen J. Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric M. Rabbah, Aleksander Slominski, and Philippe Suter. 2017. Serverless Computing: Current Trends and Open Problems. *CoRR* abs/1706.03178 (2017). arXiv:1706.03178 http://arxiv.org/abs/1706.03178
[3] Ben Congdon. 2018. Corral a MapReduce framework. https://github.com/bcongdon/corral.
[4] Qifan Pu Eric Jonas. 2018. PyWren Alternative Clouds. https://github.com/pywren/alternative_clouds.

[5] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*.

[6] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16)*.

[7] IBM. 2018. OpenWhisk Python Envirnoment. https://console.bluemix.net/docs/openwhisk/openwhisk_reference.html.

[8] IBM. 2018. Watson Studio. https://dataplatform.ibm.com.

[9] IBM. 2018. Watson Studio Community. https://dataplatform.cloud.ibm.com/community.

[10] Carlos Santana (IBM). 2018. Using the New Python 3 Runtime and IBM Cloud Services in your Serverless Apps. https://www.ibm.com/blogs/bluemix/2018/02/using-new-python-3-runtime-ibm-cloud-services-serverless-apps/.

[11] Gil Vernik (IBM) and Josep Sampé. 2018. Process large data sets at massive scale with PyWren over IBM Cloud Functions. https://www.ibm.com/blogs/bluemix/2018/04/process-large-data-sets-massive-scale-pywren-ibm-cloud-functions/.

[12] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: distributed computing for the 99%. In *2017 ACM Symposium on Cloud Computing (SoCC'17)*. ACM, 445–451.

[13] Gil Vernik (IBM) Josep Sampé. 2018. PyWren for IBM Cloud. https://github.com/pywren/pywren-ibm-cloud.

[14] jupyter. 2018. Python Notebooks. https://jupyter.org/.

[15] Youngbin Kim and Jimmy Lin. 2018. Serverless Data Analytics with Flint. *arXiv preprint arXiv:1803.06354* (2018).

[16] Qubole. 2018. Spark on Lambda. https://github.com/qubole/spark-on-lambda.

[17] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. 2017. Data-driven Serverless Functions for Object Storage. In *18th ACM/IFIP/USENIX Middleware Conference (Middleware'17)*. 121–133.

[18] Josef Spillner. 2018. Lambada. https://gitlab.com/josefspillner/lambada.