Graphical user interface, website

Description automatically generated with medium confidence

# Table of Contents

[Table of Contents 2](#_Toc95838930)

[1. Abstract 3](#_Toc95838931)

[2. Motivation and background 4](#_Toc95838932)

[2.1 Background 4](#_Toc95838933)

[2.2 What are struggler tasks 4](#_Toc95838934)

[2.3 The root cause their appear 4](#_Toc95838935)

[2.4 Example 5](#_Toc95838936)

[2.5 How different Big Data engines deals straggler tasks 5](#_Toc95838937)

[3. Stragglers in serverless computing 7](#_Toc95838938)

[3.1 Explain serverless computing 7](#_Toc95838939)

[3.2 The problem of straggler tasks in serverless paradigm 7](#_Toc95838940)

[3.3 Difference and similarity of the straggler tasks in serverless computing vs other cluster based Big Data engines 8](#_Toc95838941)

[4. Stragglers in jobs submitted by Lithops 9](#_Toc95838942)

[4.1 How Lithops submits its job and knows which tasks are finished and which tasks are still running 9](#_Toc95838943)

[4.2 How Lithops knows when entire job is finished and how it waits for the running tasks to complete 10](#_Toc95838944)

[4.3 Two different approaches to extend Lithops with mechanism to prevent jobs to be affected by straggler tasks 11](#_Toc95838945)

[4.4 Positive and negative effects of the approaches proposed, and addressing different aspects of complexity of implementation 11](#_Toc95838946)

[4.4.1 Positive effects 11](#_Toc95838947)

[4.4.2 Negative effects 12](#_Toc95838948)

[5. Prototype 13](#_Toc95838949)

[5.1 Algorithm 1: Lithops speculation mode 13](#_Toc95838950)

[5.2 Algorithm 2: Lithops recursive mode 13](#_Toc95838951)

[5.3 Algorithms' Explanation 13](#_Toc95838952)

[6. Next steps 14](#_Toc95838953)

[7. Conclusion 15](#_Toc95838954)

[8. Bibliography 16](#_Toc95838955)

# Abstract

In this project we present two algorithms for dealing with straggler tasks in Lithops. A straggler is a long-duration task that delays the completion of the entire job. The first algorithm is based on the speculation approach commonly used in big data engines. The second algorithm is based on the idea that solving multiple small problems is easier than solving a single large problem. For both approaches, we discuss their advantages and disadvantages, as well as the additional steps that need to be taken to implement them.

# Motivation and background

## Background

In 2014, Amazon Web Services (AWS) introduced the concept of serverless computing, an evolution of cloud computing. This computational model allows developers to write small pieces of cloud-native code-named functions instead of having to configure and manage servers, resources, and applications. (Parichehr Vahidinia, et al. 2020)

## What are struggler tasks

A straggler task is one that takes longer than average time to complete, which means it delays system performance due to its late completion. It is detected when a few threads take longer than the rest to execute a given iteration. In this case each iteration will proceed at the speed of the slowest thread due to the need to synchronize all threads. In addition, this problem increases with increasing parallelism: as the number of servers increases, the possibility of having a straggler increases as well. (Songze Li et al. 2017)

## The root cause their appear

Four potential resource bottlenecks that can throttle the performance of a given worker are CPU workloads, disk usage, memory limitations, and network bandwidth.

The CPU is the calculation power unit. CPU work depends on the task it is performing, for example, I/O operations require less power than computational operations. As a result, the type of operation matters when parallel operations are performed. If most of the operations are computational and use a lot of CPU power, some operations may take longer and cause delays in the system. When it comes to memory, there are operations that can be performed in parallel, however, there are operations that are performed in sequence (for example, I/O) so if most of the tasks are from the type of   
I/O, the thread execution may experience a delay waiting for each thread to be completed.

## Example

Diagram

Description automatically generated

The following figure illustrates the distributed computing model consisting of a master node and n workers. Every worker "i" store and process a subset of ri training examples locally, and then based on partial gradient calculation, they produce a message zi, which is then transmitted to the master node. Using these messages, the master computes the total gradient and updates the model. If each worker processes a disjointed subset of the examples, the master needs to gather all partial gradients from all the workers. Because of this, the run-time of each iteration of distributed GD is determined by the slowest worker (or straggler). When workers execute or communicate much slower than others, this phenomenon is known as the straggler effect, leading to significant delays in the execution of distributed computing tasks. (Li et al. 2017)

## How different Big Data engines deals straggler tasks

Big data engines, such as Hadoop, Spark, Flink, Heron and so on, are usually able to speculate stragglers, therefore, in order to shorten the execution time, they will run a copy of the task that will most likely be completed faster. (PHAN et al. 2019)

* Apache Spark - Suppose an application runs with a fraction of the input data set to predict possible stragglers and/or skewed task distribution problems in advance. As a result, if the model predicts the possibility of task straggler problems, it adjusts the partition number to divide either a longer task (i.e., straggler) into multiple shorter tasks or merge multiple tiny tasks into a larger one. When the model predicts the possibility of skewed task distribution, it adjusts the locality setting (spark.locality.wait) that controls the creation of tasks on the remote worker nodes to avoid skewed task distribution. (Wang et al. 2019)
* Hadoop – It performs a backup copy of a slow task on another machine. Furthermore, speculative execution can expedite stragglers due to skew to certain extent, since a backup copy can run on a better performing machine. (Guo, 2017)

# Stragglers in serverless computing

## Explain serverless computing

As the name implies, serverless computing is a computation without “caring” for the server itself. When developing in this way, the idea is to care only for the code writing and not worry about low-level details, which are, for example, server maintenance. A serverless computing platform enables efficient development and deployment of applications to the market without managing any infrastructure (Davide Taibi, Josef Spillner, Konrad Wawruch, 2020). Beyond the advantages of serverless architecture, the following are also significant:

* No servers – Everything is handled by the vendor, so no management is required.
* Security and High availability - The system runs the function in a variety of availability zones, thus ensuring survival if one fails.
* Full adjustment of the infrastructure- Scaling will be automatic based on the needs of the application and there will be no configuration required by the user.
* Pay per use – If the user doesn't run the code, no payment is due. It provides an interface that greatly simplifies cloud programming, and represents an evolution that parallels the transition from assembly language to high-level programming languages. (Eric Jonas, 2019)

## The problem of straggler tasks in serverless paradigm

Although serverless computing offers many benefits, it comes with a risk called chain link failure (straggler). As a matter of fact, in serverless computing, each endpoint is a link necessary for the chain to work. A failure or delay at the endpoints affects the entire process. Since the assumption is that the probability for each component to fail is independent of each other, the more components there are, the greater the probability of failure (Burak Bartan and Mert Pilanci, 2019).

## Difference and similarity of the straggler tasks in serverless computing vs other cluster based Big Data engines

Big data engines are characterized by the fact that there is some probability that there will be tasks that lag behind. As a result, there are several ways to handle these stragglers, like speculation mode, probability mode, and more. Contrary to serverless computing, which is not always functional for big data analysis and monitoring is especially difficult where worker management is handled by the cloud provider without any direct user involvement. (Dominic, 2020)

Big data engines and serverless are both fault tolerance each on its own way, the first, uses redundancy and the second invoke the function again with the same event payload. Both are managed to deal with stragglers and the user is not exposed to this process.

# Stragglers in jobs submitted by Lithops

## How Lithops submits its job and knows which tasks are finished and which tasks are still running

Lithops is a Python multi-cloud agnostic serverless computing framework that can run Local Python code at a massive scale on serverless computing platforms without modification. Using Lithops, the user does not have to worry about managing boilerplate code or deploying and running it. Lithops provides a simple and intuitive interface for handling data volumes by abstracting the underlying APIs for cloud storage[[1]](#footnote-1).

Map and reduce computations are each executed as separate jobs in Lithops.

There is an orchestrator component responsible for orchestrating all the computations in Lithops. It is instantiated prior to any use of Lithops. Its initialization includes these essential steps:

* Setup the workers[[2]](#footnote-2) by constructing docker images and defining serverless functions.
* Define a bucket in object storage (pre-configured) in which each job will store the job and input data prior to computation and results when computation is complete.
* Create an Invoker object and execute each job as independent worker tasks

In map job, a partitioner object first partitions the data.

Each worker should be able to process each partition independently.

In reduce jobs, Lithops provide two methods of execution:

* Reduce per object, where each object is processed by a reduce function, and the data is partitioned as one partition per storage object.
* Global reduce, where a single reduce function processes all data. Additionally, the reduce function is forced to wait for data before invocation since the output of the map jobs needs to be finished before the reduce job can run.

Both functions also construct jobs, described as follows:

* A job description is defined for the job.
* The partition map and the data processing function are serialized.
* The serialized partition map is stored in the object storage bucket.
* The serialized processing function and its module dependencies are stored in the same bucket

Lithops proceeds to execute the job by the following steps:

* The job is executed as independent and concurrent invocations from the beginning.
* An internal invoker creates a dictionary with all the data the invocation needs. It includes a copy of some of the job descriptions as well as some essential metadata regarding its job (e.g., call\_id, data length, log location output)
* Invocation proceeds to the computation part with a retry mechanism for the current call. that depends on the configured backend computation API
* When computation part completes, each invocation commits the result to object storage and returns a response of the computed result, which is used to wait for job completion and retrieve all results.

## How Lithops knows when entire job is finished and how it waits for the running tasks to complete

Eventually, Lithops detects job completion in one of two following configurable techniques:

* RabbitMQ[[3]](#footnote-3): A unique RabbitMQ topic is defined for each job, combining the executor id and job id. Once each worker completes its invocation, it posts a notification message on that topic, the orchestrator consumes a number of messages on that topic and compare it with the expected total calls that determine completion of the job.
* Object Storage Polling: each invocation persists its computation results in a specific object. The orchestrator is repeatedly, once per fixed period, polls the executor’s bucket for status objects of a subset of invocations that have still not completed. This allows control of resource usage and eventual detection of all calls.

## Two different approaches to extend Lithops with mechanism to prevent jobs to be affected by straggler tasks

Lithops handle straggler tasks by retry mechanism for each invocation. In addition, it specifies a timeout threshold, when the system reaches this threshold, it stops the run and returns a timeout error. This mode reduces the availability and fault tolerance of the system.

Throughout this section, solutions will be presented to improve availability and fault tolerance, ensuring that the system will continue to run even in the event of a failure (in our case, straggler).

In both approaches presented in this section the initial steps are the same.

First, the user must specify the maximum number of acceptable straggler tasks. In the next step, a median time will be calculated using the execution plan, and a straggler task threshold will be set. A task that passes this threshold is defined as a straggler task.

* Lithops speculation mode: Concurrently, another worker will perform the same task. The first worker to return a result will be considered, and the other will be stopped. Therefore, after 75% of the tasks are completed, the median of the completion time is used to identify the stragglers. If a task is 1.5 times slower than the median, it should be considered for speculation. (Danish Khan, Kshiteej Mahajan, Rahul Godha, Yuvraj Patel, 2015).
* Lithops recursive mode: Each straggler task will be partitioned into smaller partitions, and another run of the job will be performed on each partition independently. Once each partition returns its results to its master, it will return the final result to RabbitMQ or Object Storage to recognize that the overall task has been completed.

## Positive and negative effects of the approaches proposed, and addressing different aspects of complexity of implementation

## Positive effects

* + Increasing Availability - As straggler tasks are execute by additional workers, the number of late tasks will be reduced. Therefore, availability will increase and, in the worst case, remain the same.
  + Reducing running time - allocating potential straggler tasks to other workers, allowing parallel computation of the same tasks, and returning the first completed results.
  + Both methods are applicable, need to adjust the existing code and add a StragglerManger class that holds the id of the task and the worker ids.

## Negative effects

* + Constant monitoring of jobs is required, which is costly when the number of workers is large, in the approaches proposed the number of tasks will most likely increase.
  + It is challenging to monitor serverless systems, where the cloud provider manages the workers and Lithops does not have direct access. Moreover, the number of tasks will increase due to the proposed solution.
  + A worker often straggles only at the end of the job (while communicating the results). By the time the job is resubmitted, the additional communication and computational overhead might negatively impact the overall system efficiency (Vipul, 2020)
  + It will increase the cost of the total job:
    - In the Lithops speculation mode, the cost would increase by the number of stragglers, for example, if the execution consisted of N tasks and K tasks were stragglers (K<N) the new cost will be N+K.
    - In the Lithops recursive mode (limited to one level of recursion), the pricing will be defined as N+NK, when N is the number of partitioned tasks and K is the number of straggler tasks. This mode increases the cost more than the speculation mode, however the task will be solved in a higher confidence.

# Prototype

## Algorithm 1: Lithops speculation mode

1. Receiving a job (map/reduce/ map\_reduce)
2. If job equals to map\_reduce than: Divide the job into map and reduce jobs.
3. Send job information to FunctionExecutor
4. Run the job using FunctionInvoker instance.
5. While the job is not finished do:
6. If call\_id not in ResponseFuture:
7. Send call\_id to ResponseFuture.
8. If len(ResponseFuture)= total\_calls/2 than:
9. Calculate the median time of execution
10. If call execution time is greater than median\*1.5 than:
11. FunctionInvoker assign additional worker to the same call.
12. End

## Algorithm 2: Lithops recursive mode

1. Receiving a job (map/reduce/ map\_reduce)
2. If job equals to map\_reduce than: Divide the job into map and reduce jobs.
3. Send job information to FunctionExecutor
4. Run the job using FunctionInvoker instance.
5. While the job is not finished do:
6. If call\_id not in ResponseFuture:
7. Send call\_id to ResponseFuture.
8. If len(ResponseFuture)= total\_calls/2 than:
9. Calculate the median time of execution
10. If call execution time is greater than median\*1.5 than:
11. Send call information to FunctionExecutor
12. End

## Algorithms' Explanation

Each proposed algorithm from section 4.3 begins the same way, except that the straggler calls are handled differently. In the first algorithm, the call is sent to another worker, whereas in the second algorithm, the call is split into sub-calls using the FunctionExceutor.

# Next steps

* Suggest next steps to the solutions you proposed
* In the proposed solutions, the determination of the straggler threshold is set by the user. Although this method has advantages such as controlling and configuring the straggler threshold to various assignment purposes, it is also a disadvantage because the user has to be proficient in how the code works. A refinement of the method for spotting data-driven threshold stragglers can be made in the future instead of a hard threshold.
* A cost-benefit analysis is required for both proposed solutions. A variety of factors need to be considered, including runtime, storage, CPU, the ability to implement the solution, and the cause of the straggler(reference). In practice, the one that yields the best cost-benefit ratio should be selected.
* Upon accepting the solution, the next step is to redesign the system's architecture. It includes adding functions and objects, building relationships, and adding the actual features.

# Conclusion

In this research project, we explored the issue of stragglers in big data engines, serverless computing, and Lithops in particular. Straggler is a task that takes longer than the average task and therefore delays the completion of the whole job. In our literature review, we found that the solutions offered today are mainly associated with big data engines, rather than serverless computing. This explains why Spark and similar engines are often used in big-data analysis as opposed to serverless computing.

In comparison to big data engines, Lithops does not deal with stragglers. To solve the straggler problem in a new manner, we have proposed two algorithms based on professional literature. The first algorithm is based on the speculation approach used in big data engines. The second algorithm is a recursive mode in Lithops, which is based on the idea that solving several small problems is easier than solving a large one.

# Bibliography

* <https://github.com/lithops-cloud/lithops>
* <https://lithops-cloud.github.io/docs/source/design.html>
* <https://arxiv.org/search/?query=straggler+serverless&searchtype=all&source=header>
* <https://lithops-cloud.github.io/docs/index.html>
* Danish Khan, Kshiteej Mahajan, Rahul Godha, Yuvraj Patel, Empirical Study of Stragglers in Spark SQL and Spark Streaming, WISC, 2015
* Cold Start in Serverless Computing: Current Trends and Mitigation Strategies, 2020, Parichehr Vahidinia, Bahar Farahani, Fereidoon Shams Aliee
* Near-Optimal Straggler Mitigation for Distributed Gradient Methods, 2017 , Songze Li∗ , Seyed Mohammadreza Mousavi Kalan∗ , A. Salman Avestimehr, and Mahdi Soltanolkotabi University of Southern California
* A New Framework for Evaluating Straggler Detection Mechanisms in MapReduce, 2019, TIEN-DAT PHAN, GUILLAUME PALLEZ, SHADI IBRAHIM, PADMA RAGHAVAN
* A Model Driven Approach towards Improving the Performance of Apache Spark Applications, 2019, Kewen Wang, Mohammad Maifi Hasan Khan, Nhan Nguyen and Swapna Gokhale
* Moving Hadoop into the Cloud with Flexible Slot Management and Speculative Execution,2017, Yanfei Guo, Member, IEEE, Jia Rao, Member, IEEE, Changjun Jiang, Member, IEEE, and Xiaobo Zhou, Senior Member, IEEE
* Serverless Computing— Where Are We Now, and Where Are We Heading, Davide Taibi, Josef Spillner, Konrad Wawruch, 2020
* Cloud Programming Simplified: A Berkeley View on Serverless Computing
* Straggler Resilient Serverless Computing Based on Polar Codes, 2019, Burak Bartan and Mert Pilanci
* Serverless Straggler Mitigation using Local Error-Correcting Codes,2020 Vipul Gupta, Dominic Carrano , Yaoqing Yang, Vaishaal Shankar, Thomas Courtade and Kannan Ramchandran Department of EECS, UC Berkeley
* Open-source message-broker software initially implemented the Advanced Message Queuing Protocol and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol, MQ Telemetry Transport, and other protocols.

1. <https://lithops-cloud.github.io/docs/index.html> [↑](#footnote-ref-1)
2. A worker performs one unit of computation (e.g., processing one dataset chunk or one object) within a more extensive job of Lithops [↑](#footnote-ref-2)
3. Open-source message-broker software initially implemented the Advanced Message Queuing Protocol and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol, MQ Telemetry Transport, and other protocols. [↑](#footnote-ref-3)