Big Data Platforms

Stragglers and serverless computing

Oren Ben Eliyahu

Yuval Barkan

February 16th, 2022

# Table of Contents

[Table of Contents 2](#_Toc94965030)

[1. Abstract: 4](#_Toc94965031)

[2. Motivation and background: 5](#_Toc94965032)

[3.1 Background – 5](#_Toc94965033)

[3.2 What are struggler tasks: 5](#_Toc94965034)

[**3.3** **The root cause their appear –** 5](#_Toc94965035)

[3.4 Examples – 6](#_Toc94965036)

[3.5 How different Big Data engines deals straggler tasks- 6](#_Toc94965037)

[3. Stragglers in serverless computing: 8](#_Toc94965038)

[4.1 Explain what is serverless computing 8](#_Toc94965039)

[4.2 The problem of straggler tasks in serverless paradigm 8](#_Toc94965040)

[4.3 Difference and similarity of the straggler tasks in serverless computing vs other cluster based Big Data engines 9](#_Toc94965041)

[4. Stragglers in jobs submitted by Lithops 10](#_Toc94965042)

[5.1 Explore and explain how Lithops submit it’s job and knows which tasks are finished and which tasks yet running: 10](#_Toc94965043)

[5.2 Explain how Lithops know when entire job is finished and how it waits for the running tasks to complete. 11](#_Toc94965044)

[5.3 Provide two different approaches to extend Lithops with mechanism to prevent jobs to be affected by straggler tasks: 12](#_Toc94965045)

[5.4 Address different aspects of complexity of implementation, costs, performs: 12](#_Toc94965046)

[5.5 Explain positive and negative effects of the approaches you propose: 12](#_Toc94965047)

[5. Prototype 12](#_Toc94965048)

[6. Next steps 12](#_Toc94965049)

[7. Conclusion 12](#_Toc94965050)

[8. Bibliography 12](#_Toc94965051)

# Abstract:

# 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.[[1]](#footnote-1)

## What are struggler tasks:

Straggler tasks are those that take longer to complete than the average task, which delays the system's performance because it takes longer than expected. In short, a straggler problem is detected when a few threads take longer than the rest to execute a given iteration. Each iteration proceeds 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.

A performance bottlenecks arise when some of the worker nodes run slowly. Nodes with these problems, also known as stragglers, can significantly slow down computation, as the slowest node may dictate the overall computation time.[[2]](#footnote-2)

## The root cause their appear –

Four potential resource bottlenecks that can throttle the performance of a given worker. 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 the majority of the tasks are from the type of IO, 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 via (1). If each worker processes a disjoint 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.[[3]](#footnote-3)

## 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.[[4]](#footnote-4)

* Apache Spark - given an application that will first run with a fraction of the input data set to predict possible stragglers and/or skewed task distribution problem in advance. Subsequently, if the model predicts the possibility of task straggler problem, we use our performance models to repartition the input data (or intermediate data if needed) by adjusting the partition number to either split a longer task (i.e., straggler) into multiple shorter tasks, or merge multiple tiny tasks into a larger one. On the other hand, if the model predicts the possibility of skewed task distribution problem, we tune the locality setting (i.e., spark.locality.wait) that controls the task creation on remote worker nodes to address possible skewed task distribution problem[[5]](#footnote-5)
* Hadoop - speculatively runs a backup copy of a slow task on a different machine. Besides fault-tolerance, speculative execution can expedite stragglers due to skew to a certain extent as the backup copy may run on a better performing machine.[[6]](#footnote-6)

# Stragglers in serverless computing:

## Explain what is 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, the server maintenance. Serverless computing provides a platform to efficiently develop and deploy applications to the market without having to manage any underlying infrastructure.[[7]](#footnote-7)

In addition, beyond the serverless architecture benefits, there are four main advantages:

* No servers – Do not need to manage infrastructure, everything is managed by the vendor.
* Security and High availability - The system will run your function in a variety of Availability Zones, to ensure survival in the event of a malfunction in one of them.
* Full adjustment of the infrastructure- The system will allow auto scaling according to the requirement of the application and the user will not need to do anything to configure it.
* Payment per use – In case the user did not run the code the user will not pay.

\_\_

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.[[8]](#footnote-8)

## The problem of straggler tasks in serverless paradigm

Along with the benefits of a serverless computing method arises a problem that can be described as the potential for chain link failure (straggler). In fact, in the serverless computing each endpoint constitutes a link that is required so that the full chain will function. Failure/delayed of one of the components at the endpoints impairs the entire process. In addition, since the assumption is that the probability of each component to struggler is independent on each other the higher the number of components the greater the probability of failure.[[9]](#footnote-9)

\*\*It is common in serverless computing for a function to return nothing because functions can crash due to the lifetime constraint or for other reasons.

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

\*Speculative execution works by detecting workers that are running slowly, or will slow down in the future, and then assigning their jobs to new workers without shutting down the original job. The worker that finishes first submits its results. This has several drawbacks: constant monitoring of jobs is required, which is costly when the number of workers is large. Monitoring is especially difficult in serverless systems where worker management is done by the cloud provider and the user has no direct supervision over the workers. Moreover, it is often the case that a worker straggles only at the end of the job (say, while communicating the results). By the time the job is resubmitted, the additional communication and computational overhead would have decreased the overall efficiency of the system.[[10]](#footnote-10)

# Stragglers in jobs submitted by Lithops

## Explore and explain how Lithops submit it’s job and knows which tasks are finished and which tasks yet running:

Lithops is a Python multi-cloud agnostic serverless computing framework that can run Local Python code can be run 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[[11]](#footnote-11).

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:

* Set up the workers[[12]](#footnote-12) 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.

## Explain how Lithops know 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[[13]](#footnote-13): 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.

## Provide 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 you propose, 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.[[14]](#footnote-14)
  + 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

* Provide pseudo-code for the approaches you propose. Explain how it works.

# Next steps

* Suggest next steps to the solutions you proposed

# Conclusion

* Short conclusion of the work you did

# Bibliography

* Danish Khan, Kshiteej Mahajan, Rahul Godha, Yuvraj Patel, Empirical Study of Stragglers in Spark SQL and Spark Streaming, WISC, 2015

1. Cold Start in Serverless Computing: Current Trends and Mitigation Strategies, 2020, Parichehr Vahidinia, Bahar Farahani, Fereidoon Shams Aliee [↑](#footnote-ref-1)
2. 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 [↑](#footnote-ref-2)
3. 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 [↑](#footnote-ref-3)
4. A New Framework for Evaluating Straggler Detection Mechanisms in MapReduce, 2019, TIEN-DAT PHAN, GUILLAUME PALLEZ, SHADI IBRAHIM, PADMA RAGHAVAN [↑](#footnote-ref-4)
5. A Model Driven Approach towards Improving the Performance of Apache Spark Applications, 2019, Kewen Wang, Mohammad Maifi Hasan Khan, Nhan Nguyen and Swapna Gokhale [↑](#footnote-ref-5)
6. 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 [↑](#footnote-ref-6)
7. Serverless Computing— Where Are We Now, and Where Are We Heading, Davide Taibi, Josef Spillner, Konrad Wawruch, 2020 [↑](#footnote-ref-7)
8. Cloud Programming Simplified: A Berkeley View on Serverless Computing [↑](#footnote-ref-8)
9. Straggler Resilient Serverless Computing Based on Polar Codes, 2019, Burak Bartan and Mert Pilanci [↑](#footnote-ref-9)
10. 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 [↑](#footnote-ref-10)
11. <https://lithops-cloud.github.io/docs/index.html> [↑](#footnote-ref-11)
12. 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-12)
13. 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-13)
14. 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 [↑](#footnote-ref-14)