Proceedings Paper:
Predoaia, Ionut orcid.org/0000-0002-2009-4054 and García-López, Pedro (2023) Leveraging Intra-Function Parallelism in Serverless Machine Learning. In: WoSC '23: Proceedings of the 9th International Workshop on Serverless Computing. 9th International Workshop on Serverless Computing, WoSC '23, 11-15 Dec 2023 WoSC '23 . ACM , ITA , 36–41.

https://doi.org/10.1145/3631295.3631399

Leveraging Intra-Function Parallelism in Serverless Machine Learning

Ionut Predoaia
University of York
York, United Kingdom
ionut.predoaia@york.ac.uk

Pedro García-López
Universitat Rovira i Virgili
Tarragona, Spain
pedro.garcia@urv.cat

Abstract

Running stateful machine learning algorithms with serverless architectures inherently induces overheads, as serverless functions are not directly network-addressable, hence one must rely on a remote storage service for storing the shared state. To hide the access latency to the remote storage, one can employ intra-function parallelism to take advantage of the multicore computing resources of the serverless functions. In this work, we port to serverless two stateful machine learning algorithms, *k*-means clustering and logistic regression, and then adopt intra-function parallelism to parallelize the execution of the serverless functions. Several experiments have demonstrated that intra-function parallelism delivers performance improvements in serverless machine learning. Improved performances of up to 68% have been achieved when running *k*-means on serverless functions that employ intra-function parallelism. We demonstrate with *k*-means and logistic regression that from a performance perspective it is preferable to execute a smaller number of multiple-vCPUs workers than a larger number of single-vCPU workers, due to decreased synchronization overheads.

CCS Concepts: • **Computing methodologies → Machine learning**; **Parallel computing methodologies**; • **Computer systems organization → Cloud computing**.

Keywords: Serverless, Machine Learning, Intra-Function Parallelism, Multicore Functions, Stateful, Lithops

ACM Reference Format:
Ionut Predoaia and Pedro García-López. 2023. Leveraging Intra-Function Parallelism in Serverless Machine Learning. In *9th International Workshop on Serverless Computing (WoSC '23), December 11–15, 2023, Bologna, Italy.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3631295.3631399

WoSC '23, December 11–15, 2023, Bologna, Italy
© 2023 Association for Computing Machinery.
ACM ISBN 979-8-4007-0455-0/23/12…$15.00
https://doi.org/10.1145/3631295.3631399

1 Introduction

Stateful machine learning algorithms running on serverless architectures have limitations regarding connectivity and the management of shared state. As serverless functions are not directly network-addressable, they cannot communicate with each other to share the state related to an iterative machine learning algorithm (e.g., the centroids in *k*-means clustering, the gradients in logistic regression). Consequently, one must rely on a remote storage service for storing the shared state, which can be used by the serverless functions to access and update the shared state, over the network. This inherently induces communication overheads, as multiple concurrent serverless functions must access and update the shared state at every iteration, and consequently, significant overheads can result in the case of running hundreds of iterations. Moreover, algorithms that rely on the Bulk Synchronous Parallel (BSP) synchronization protocol incur synchronization overheads, as the workers must wait for each other at the end of every iteration. For instance, the *k*-means clustering algorithm is highly parallelizable, however, it requires regular communication at the end of each iteration to update and retrieve the new centroids.

Cloud providers typically provide one virtual CPU (vCPU) per serverless function, however, a few cloud providers have upgraded their offering by providing multiple vCPUs per serverless function (e.g., AWS Lambda and Google Cloud Functions). Therefore, engineers can adopt a so-called intra-function parallelism pattern to leverage the multicore computing resources of a serverless function for parallelizing its execution via threads or processes. Note that the amount of memory determines the number of vCPUs available to a serverless function, thus an increase in memory is correlated with a proportional increase in the number of vCPUs.

In this work, two stateful machine learning algorithms will be ported to serverless, *k*-means clustering and logistic regression. Furthermore, a technique will be proposed for adopting intra-function parallelism to parallelize the execution of serverless functions. We found that leveraging intra-function parallelism can compensate for the communication and synchronization overheads by achieving improved performances of up to 68%.

2 Related Work

Many research efforts have been carried out for addressing machine learning in the context of serverless computing. To mention a few, [2, 3, 8, 15] are examples of such works. Nevertheless, in the context of serverless machine learning, no prior works have adopted intra-function parallelism to leverage the multicore computing resources of serverless functions for parallelizing their execution. Moreover, it can be argued that in serverless computing, intra-function parallelism is poorly addressed as well in previous works that are outside the area of machine learning [11].

Note that intra-function parallelism is alternatively called intra-worker parallelism or intra-level parallelism within workers [5]. Moreover, serverless functions that have multiple vCPUs are called multicore functions [9], and in the context of AWS Lambda, they are called big lambdas [4].

Intra-function parallelism has been employed in serverless machine learning in [14], however, this work is not based on multicore functions but rather is based on IBM Cloud Functions, which are limited to 1 vCPU at most. The impact of leveraging intra-function parallelism for parallelizing serverless functions has been analyzed in [10], and significant cost savings have been achieved: 81% cost savings with AWS Lambda and 49% with Google Cloud Functions. Furthermore, intra-function parallelism has been adopted for decompression [12] and fog robotics algorithms [7]. Moreover, a technique has been presented in [16] for automating the resource configuration of functions, by optimizing the memory size of each function step in a workflow whilst taking into consideration intra-function parallelism.

3 Portage to Serverless

Lithops has been used for porting k-means and logistic regression to serverless. Lithops [13] is a multi-cloud distributed computing framework that enables engineers to run unmodified single-machine Python code at scale in the main serverless computing platforms, such as AWS, Google Cloud and IBM Cloud. With regard to shared state management, Lithops provides a Multiprocessing module containing abstractions [1] that enable sharing state among serverless functions. Note that the Lithops Multiprocessing API mimics the Python Multiprocessing API, however, the shared state abstractions store their state values in a remote memory data store, i.e., the Redis in-memory key-value database.

In our serverless implementations[1] of the stateful machine learning algorithms, the BSP synchronization protocol has been used, as it ensures the correctness of the results of the algorithms. However, the protocol has the limitation that it induces synchronization overheads, as the workers must wait for each other at the end of every iteration. A client machine executes a client program that launches several concurrent serverless functions, that are alternatively called workers, that share mutable state through disaggregated memory, i.e., Redis. Each worker operates on a partition of the data set and at each iteration exchanges partial results with the other workers via the shared state that is stored on a remote Redis node.

When porting stateful machine learning algorithms to serverless, one must first identify what data can be computed independently and concurrently by the workers. The workers perform parallel computations regarding data of interest (e.g., centroids) and then the partial results obtained by all workers are aggregated to attain global results. The partial results computed by the workers must be stored in the shared state, as they must be accessed at a later point for obtaining global results. Furthermore, the global results must be stored in the shared state, for the reason that they must be accessed by all workers at each iteration. In the case of the k-means algorithm, the partial results are represented by the clusters counters and totals, whereas the global results are the centroids that are computed by dividing the clusters totals by the counters. Moreover, in the case of the logistic regression algorithm, the partial results are represented by the partial gradients, whereas the global results are represented by the weights and the global gradients.

3.1 Serverless K-Means

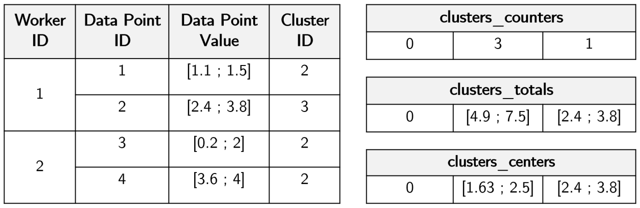
To compute the centroid of a cluster, one would need the sum of all data points assigned to the cluster, and the number of data points belonging to the cluster. As such, the core data objects in the shared state are the sum of all data points for each cluster and the number of data points from each cluster.

Figure 1 presents how the data objects from the shared state are computed, where the table from the left represents an example data set, and each table from the right represents a variable from the shared state. In the example, we will consider that the k-means algorithm is executed with $k = 3$ clusters using 2 workers. The data set is bi-dimensional, and it is split into two partitions, one for each worker. Each worker iterates the data points, assigns them to clusters and updates the shared state variables accordingly. To exemplify, the first worker assigns the first data point to the second cluster, therefore it increments by 1 the value of the second index of the `clusters_counters` array, and additionally, the first data point is summed to the value of the second index of the `clusters_totals` array. Three data points from the data set are assigned to the second cluster, therefore the value at index 2 of `clusters_counters` will become 3 by the end of the iteration. Furthermore, the value at index 2 of `clusters_totals` will be the sum of all data points assigned to the second cluster by the end of the iteration. To be specific, the data point [4.9; 7.5] represents the sum of all data points assigned to the second cluster, i.e., [1.1; 1.5], [0.2; 2], and [3.6; 4]. After both workers finish assigning the data points to clusters, they will then update the clusters counters and totals from the shared state. At

[1] https://github.com/neardata-eu/lithops-ml-big-lambdas

this point, all workers synchronize their execution via a synchronization barrier that is stored as a barrier object in the shared state. Then, one of the workers, by convention the first worker, will compute the values of the centroids. The first worker will fetch the final values of the iteration of `clusters_counters` and `clusters_totals`, and then compute the centroids by dividing the totals by the counters. Finally, the first worker will update the shared state variable `clusters_centers` by the computed centroids. At the beginning of the next iteration, all workers will retrieve the current centroids by fetching the value of `clusters_centers` from the shared state.

The output of k-means is the centroids and the labels. The centroids are stored in the shared state, i.e., remotely in the Redis node, therefore the serverless functions do not need to output the centroids, as they can be accessed by the client machine directly via the shared state. However, the labels (i.e., the cluster index to which each data point is assigned) represent data that is local to the serverless functions, and must be returned to the client machine that launched the serverless functions. Each serverless function outputs a set of partial labels, which will later be concatenated on the client machine to obtain the complete list of labels for the entire data set.



Worker ID	Data Point ID	Data Point Value	Cluster ID
1	1	[1.1 ; 1.5]	2
	2	[2.4 ; 3.8]	3
2	3	[0.2 ; 2]	2
	4	[3.6 ; 4]	2

clusters_counters		
0	3	1

clusters_totals		
0	[4.9 ; 7.5]	[2.4 ; 3.8]

clusters_centers		
0	[1.63 ; 2.5]	[2.4 ; 3.8]

Figure 1. Serverless K-Means - Shared State Variables

3.2 Adopting Intra-Function Parallelism

Intra-function parallelism can be employed to improve the performance of the stateful machine learning algorithms running on serverless architectures. Figure 2 illustrates the execution design of the algorithms that employ intra-function parallelism. With intra-function parallelism, the computation phase of the algorithms is carried out by inner workers, rather than workers. Each worker launches at every iteration a number of inner workers, where the number is dictated by the number of vCPUs of the serverless function. An inner worker represents a worker that is nested in another (parent) worker, and it is implemented as a child process. Each worker splits his data set partition into sub-partitions that are sent to the inner workers. The inner workers operate over the sub-partitions and run the computation phase of the iteration in parallel. When the inner workers have finished running the computation, they send their partial results to the (parent) workers, and then each (parent) worker aggregates the received partial results. The advantage of adopting intra-function parallelism is that a higher level of parallelism can be achieved with a fewer number of connection points to the Redis node, as only the workers access and update the shared state. This can potentially reduce synchronization overheads, as there are fewer connection points to the Redis node. Moreover, the additional vCPUs can be leveraged to read in parallel from object storage the data set partition associated with a serverless function.

Algorithm 1 contains the pseudocode of a generalized serverless implementation of logistic regression that employs intra-function parallelism. The algorithm takes as input X which represents the observations of the training set partition assigned to the worker, y which represents the true class labels of the observations from the training set partition, *worker_id* which represents the identifier of the worker, *max_iter* which represents the maximum number of iterations, *learning_rate* which specifies the step size, and *cpu_count* which specifies the level of parallelism adopted by the worker (i.e., either sequential or parallel via inner workers). The output of the algorithm is the global weights. The shared state is represented by all variables prefixed by "`global`", whereas the variables prefixed by "`local`" represent the local version of the variables from the shared state. The shared state is primarily comprised of the gradients and the weights, which have the initial values of 0. The iteration phase begins at line 1, and at line 2, the global weights are fetched from the shared state and stored in a local variable. At lines 3-4, the partial gradients are computed sequentially, whereas at lines 5-9, the partial gradients are computed in parallel by the inner workers. At line 6, the worker splits his data set partition into sub-partitions that will be used by the inner workers which are instantiated at line 7. At line 8, the inner workers are invoked and compute the partial gradients, which are retrieved and aggregated at line 9. At line 10, the global gradients from the shared state are incremented by the partial gradients computed by each worker. At line 11, all workers synchronize, and by this point, the final values for the current iteration of the gradients have been obtained. At lines 12-13, the first worker computes the new weights using the final global values of the gradients aggregated from all workers and the learning rate, and then updates the global weights with the computed weights. At line 14, the first worker resets the values of the global gradients to 0 for the next iteration. Finally, the barrier from line 15 waits for the first worker to update the global weights and gradients before continuing the execution of the other workers.

4 Experimental Evaluations

Several experiments have been carried out to evaluate the achieved performance improvement when leveraging intra-function parallelism in serverless functions. All experiments have been conducted in AWS within a Virtual Private Cloud (VPC) located in the *eu-west2* region. The serverless functions

Algorithm 1: Serverless Logistic Regression with Intra-Function Parallelism

```
Input: $X$, $y$, $worker\_id$, $max\_iter$, $learning\_rate$, $cpu\_count$
Output: $global\_weights$
for $iter \leftarrow 0$; $iter < max\_iter$; $iter \leftarrow iter + 1$ do
    $local\_weights \leftarrow global\_weights$
    if $cpu\_count == 1$ then
        $local\_gradients \leftarrow$ computeGradients($X$, $y$, $local\_weights$)
    else if $cpu\_count > 1$ then
        $partitions \leftarrow$ partitionDataset($X$, $y$, $cpu\_count$)
        $inner\_workers \leftarrow$ createInnerWorkers($partitions$, $local\_weights$)
        invoke($inner\_workers$)
        $local\_gradients \leftarrow inner\_workers$.getGradients()
    $global\_gradients \leftarrow global\_gradients + local\_gradients$
    $global\_barrier$.wait()
    if $worker\_id == 0$ then
        $global\_weights \leftarrow local\_weights - learning\_rate \cdot global\_gradients$
        $global\_gradients \leftarrow 0$
    $global\_barrier$.wait()
end
```

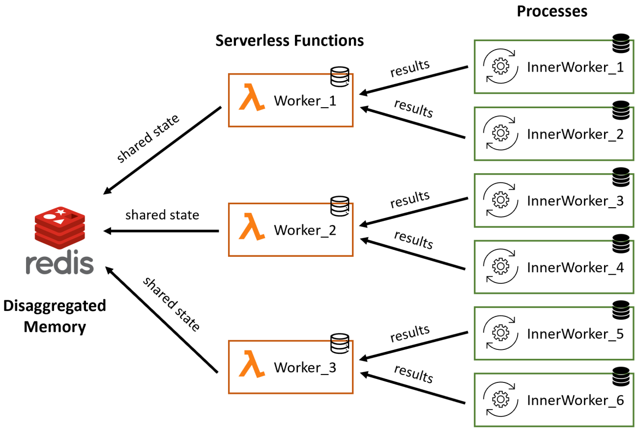


Figure 2. Adopting Intra-Function Parallelism

are running via AWS Lambda, and the data sets are stored in Amazon S3. For the client machine that launches the serverless implementations, a general-purpose EC2 instance of type *t2.2xlarge* is used, whereas a memory-optimized EC2 instance of type *r5.large* is used for the Redis node.

In the first experiment, the k-means algorithm has been executed with a data set of 8GB and with a growing number of concurrent serverless functions, from 50 up to 400, where each serverless function has 6 vCPUs allocated. In the first (baseline) instance, the k-means algorithm was executed without employing intra-function parallelism, i.e., without leveraging the multicore computing resources of each serverless function. In the second instance, the k-means algorithm was executed by employing intra-function parallelism, by using 2 vCPUs of each serverless function, i.e., making use of 2 inner workers for each worker. This procedure was repeated with 2 up to 6 inner workers. In the last instance, the k-means algorithm was executed by employing intra-function parallelism, by using all 6 vCPUs of each serverless function, i.e., making use of 6 inner workers for each worker. The performance of the algorithm when leveraging intra-function parallelism, i.e., when using 2 up to 6 inner workers, has been compared against the baseline, i.e., when not employing intra-function parallelism.

For the described experiment, Figure 3 presents the performance improvement gained by leveraging intra-function parallelism. The horizontal axis of the chart represents the number of workers used in the execution of the k-means algorithm, whereas the vertical axis represents the performance improvement obtained by using 2 to 6 inner workers compared to the case in which intra-function parallelism is not employed. For instance, the chart shows that when running the algorithm with 50 workers and 6 inner workers, a performance improvement of 68% has been obtained compared to when using 50 workers without any inner workers, i.e., without employing intra-function parallelism. Additionally, the chart shows that when running the algorithm with 400 workers, each containing 2 inner workers, a performance improvement of 20% has been obtained compared to when using 400 workers without any inner workers, i.e., without leveraging intra-function parallelism. The results show that one can achieve improved performances by up to 68% when leveraging intra-function parallelism. Note that the performance improvement is measured as a relative percentage to the baseline. It can be noticed that as the number of workers

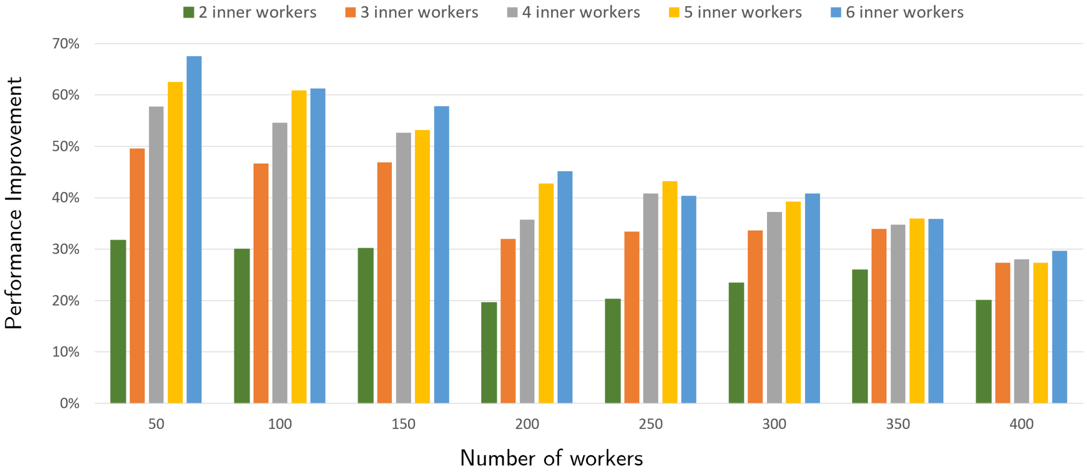


Figure 3. Serverless Intra-Function Parallelism Performance using 2 to 6 Inner Workers

increases, the performance gained by employing inner workers slowly decreases. Therefore, employing intra-function parallelism when the number of workers is very high may provide minimal performance improvements.

One may argue that the previous experiment does not carry out a fair evaluation. For example, executing 50 workers with 3 inner workers each does not yield the same level of parallelism as 50 workers without inner workers. In the case of executing 50 workers, each with 3 inner workers, 150 vCPUs are used, whereas in the case of using 50 workers without inner workers, only 50 vCPUs are used. Furthermore, one may argue that instead of using 50 workers with 3 inner workers each, one could instead simply use 150 workers without any inner workers. To this end, experiments have been carried out the evaluate the performance improvement obtained when leveraging intra-function parallelism, whilst maintaining the same level of parallelism. The aim is to determine whether it is better to use multiple workers, each with 1 vCPU, without inner workers, or a lesser number of workers with multiple vCPUs, each employing multiple inner workers. For example, it is to be determined whether a better performance can be obtained when invoking 50 serverless functions, each with 6 vCPUs, that leverage intra-function parallelism, compared to when invoking 300 serverless functions, each with only 1 vCPU, as in both cases a level of parallelism of 300 is achieved.

An experiment has been carried out, in which the *k*-means algorithm has been executed with a data set of 500MB and with various numbers of workers and inner workers, whilst maintaining an equivalent level of parallelism. As a baseline, the *k*-means algorithm has been executed with 300 workers, where each serverless function has 1 vCPU allocated with 1500MB of memory. In the baseline, the workers have been executed sequentially, without employing intra-function parallelism. Thus, a level of parallelism of 300 is achieved, as 300 vCPUs are used in the execution of the algorithm. Next, the *k*-means algorithm has been executed with 150 workers, where each serverless function has 2 vCPUs allocated. Intra-function parallelism has been employed, as each worker launched 2 inner workers. Similarly to the baseline, a level of parallelism of 300 is achieved, as 300 vCPUs are used in the execution of the algorithm. Moreover, the *k*-means algorithm has been executed with 100 workers with 3 inner workers each, 75 workers with 4 inner workers each, 60 workers with 5 inner workers each, and 50 workers with 6 inner workers each. Table 1 presents the performance improvement relative to the baseline. The memory of the serverless functions has been proportionally increased for each additional vCPU. The results show that a better performance is achieved with a lesser number of workers that have more inner workers. The execution of a smaller number of multiple-vCPUs workers is faster than a larger number of single-vCPU workers. After carrying out a breakdown of the execution times, it has been determined that the reason for the performance improvement is the decrease in synchronization overheads. Having many workers causes significant synchronization overheads due to having many connection points to the Redis node. Therefore, having a fewer number of connection points to the Redis node, decreases the synchronization overheads, and thus, the total execution times decrease. Note that the measured launch durations of big lambdas are longer by a maximum of 1 second compared to the baseline, therefore the influence over execution times is negligible.

To validate the previous experiment's results, the previous experiment has been repeated, but with the logistic regression algorithm, running for 90 iterations, with a large data set of 120GB. Table 1 presents the results of the experiment. Compared to the previous experiment running *k*-means, a similar performance improvement has been obtained. The reason for the performance improvement is the decrease in synchronization overheads, that are caused by having many workers, i.e., connection points to the Redis node.

Workers	Inner Workers	K-Means	Logistic Regression
150	2	46%	47%
100	3	62%	57%
75	4	68%	65%
60	5	71%	68%
50	6	74%	67%

Table 1. Serverless Intra-Function Parallelism Performance with Equivalent Level of Parallelism

5 Limitations

AWS Lambda allows only a maximum of 6 vCPUs per serverless function. A drawback is that the amount of memory determines the number of vCPUs available to a serverless function, as one may want to employ intra-function parallelism to leverage the multiple vCPUs, but may not need the additional amount of memory. Furthermore, as the cost of executing AWS Lambda functions depends on the allocated memory, one may have to spend more only to benefit from intra-function parallelism.

As the serverless implementations have been realized with Python, processes have been used to implement the inner workers, rather than threads, to avoid the limitation of the Python Global Interpreter Lock (GIL). However, AWS Lambda does not provide shared memory for processes. As inter-process shared memory is restricted, one has to rely on pipes for sending the output of the inner workers to the (parent) workers. This can be problematic, as large transfer overheads may be induced when the inner workers have a large output. For instance, when running k-means over large data sets significant transfer overheads are incurred, considering that at each iteration the inner workers send their resulting partial labels to the workers via pipes. This limitation can be mitigated with a CPython extension that performs multithreading without the Python GIL [6].

6 Conclusions and Future Work

In this paper, we demonstrated the performance benefits of leveraging intra-function parallelism in stateful machine learning algorithms running on serverless architectures. We ported two stateful machine learning algorithms to serverless, k-means clustering and logistic regression, and then adopted intra-function parallelism to parallelize the execution of the serverless functions. Several experiments have demonstrated that intra-function parallelism is beneficial to the performance of machine learning algorithms running on serverless architectures. Improved performances of up to 68% have been achieved by leveraging intra-function parallelism. Furthermore, we demonstrated that from a performance perspective, it is preferable to execute a smaller number of multiple-vCPUs workers than a larger number of single-vCPU workers, due to decreased synchronization overheads. In future work, it would be beneficial to validate the generality of our results with other machine learning algorithms. A different line of work could involve using Google Cloud Functions, as they support up to 8 vCPUs, or using threads for the implementation of inner workers.

References

[1] Aitor Arjona, Gerard Finol, and Pedro García López. 2023. Transparent serverless execution of Python multiprocessing applications. *Future Generation Computer Systems* 140 (2023), 436–449.

[2] Amine Barrak, Fabio Petrillo, and Fehmi Jaafar. 2022. Serverless on Machine Learning: A Systematic Mapping Study. *IEEE Access* (2022).

[3] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2018. A Case for Serverless Machine Learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, Vol. 2018. 2–8.

[4] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing*. 13–24.

[5] Germán T. Eizaguirre and Marc Sánchez-Artigas. 2023. A Seer Knows Best: Auto-tuned Object Storage Shuffling for Serverless Analytics. *J. Parallel and Distrib. Comput.* (2023), 104763.

[6] Python Software Foundation. 2023. *Python Multithreading without GIL*. [Online]. Available: https://github.com/colesbury/nogil.

[7] Jeffrey Ichnowski, William Lee, Victor Murta, Samuel Paradis, Ron Alterovitz, Joseph E. Gonzalez, Ion Stoica, and Ken Goldberg. 2020. Fog Robotics Algorithms for Distributed Motion Planning Using Lambda Serverless Computing. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 4232–4238.

[8] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the 2021 International Conference on Management of Data*. 857–871.

[9] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing*. 158–164.

[10] Michael Kiener, Mohak Chadha, and Michael Gerndt. 2021. Towards Demystifying Intra-Function Parallelism in Serverless Computing. In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*. 42–49.

[11] Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Chengzhong Xu. 2022. Serverless Computing: State-of-the-Art, Challenges and Opportunities. *IEEE Transactions on Services Computing* 16, 2 (2022), 1522–1539.

[12] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 115–130.

[13] Josep Sampé, Marc Sánchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro García-López. 2021. Outsourcing Data Processing Jobs With Lithops. *IEEE Transactions on Cloud Computing* (2021).

[14] Pablo Gimeno Sarroca and Marc Sánchez-Artigas. 2023. MLLess: Achieving Cost Efficiency in Serverless Machine Learning Training. *J. Parallel and Distrib. Comput.* (2023), 104764.

[15] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1288–1296.

[16] Zhaojie Wen, Yishuo Wang, and Fangming Liu. 2022. StepConf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 1868–1877.