λ-FL : Serverless Aggregation For Federated Learning

K. R. Jayaram, Vinod Muthusamy, Gegi Thomas, Ashish Verma and Marc Purcell

IBM Research, USA and Ireland

Abstract

Advances in federated learning (FL) algorithms, along with technologies like differential privacy and homomorphic encryption, have led to FL being increasingly adopted and used in many application domains. Traditional tree-based parallelization schemes can enable FL aggregation to scale to thousands of participants, but (i) waste a lot of resources due to the fact that training on participants, as opposed to aggregation, is the bottleneck in many FL jobs, (ii) require a lot of effort for fault tolerance and elastic scalability.

In this paper, we present a new architecture for FL aggregation, based on serverless technology/cloud functions. We describe how our design enables FL aggregation to be dynamically deployed only when necessary, elastically scaled to handle participant joins/leaves and is fault tolerant with minimal effort required on aggregator side. We also demonstrate that our prototype based on Ray (Moritz et al. 2018) scales to thousands of participants, and is able to achieve a $> 90\%$ reduction in resource utilization with minimal impact on aggregation latency.

Introduction

Federated Learning (FL) (Kairouz et al. 2019) provides a collaborative training mechanism, which allows multiple parties to build a joint machine learning model. FL allows parties to retain private data within their controlled domains. Only model updates are shared, typically, to a central aggregation server. Though there have been demonstrated data reconstruction (model inversion) and membership inference attacks on model updates (Zhao, Mopuri, and Bilen 2020; Zhu, Liu, and Han 2019; DLG git repository; iDLG git repository; IG git repository; Geiping et al. 2020), there have also been solutions proposed (and proven) to prevent information leakage from model updates, including homomorphic encryption e.g., (Jayaram et al. 2020; Aono et al. 2017), shuffling and differential privacy (Abadi et al. 2016) through the addition of noise to model updates. These security measures have made FL especially attractive for mutually distrusting/competing training participants as well as holders of sensitive data (e.g., health and financial data) seeking to preserve data privacy. Recently, FL has been applied to several real world problems on private data, e.g., Google Gboard (Bonawitz et al. 2019) and COVID CT-Scans (Dayan 2021) and has been shown to achieve significant increases in model utility.

Aggregation of model updates is arguably the most important component of an FL job. The increased adoption and use of FL has made apparent the lack of scalable, fault-tolerant and resource efficient FL aggregation platforms. Most existing FL platforms (IBM FL (Ludwig et al. 2020), FATE (Liu et al. 2021), NVFLARE (NVIDIA 2021)) are based on a client-server model with aggregators deployed in datacenters waiting for model updates. Often, training at the party takes much longer compared to model update fusion/aggregation, resulting in under-utilization and wastage of computing resources. This is a significant problem even in so-called "cross-silo" deployments, where the number of parties is small; parties have dedicated resources available for training and are actively participating. It is further compounded in "cross-device" deployments, where the number of parties is very large (> 1000), parties are highly intermittent and do not have dedicated resources for training. This results in aggregators having to wait for long periods of time for parties to finish local training and send model updates.

This paper proposes a flexible parameter aggregation architecture and mechanism for federated learning, that:

- supports both intermittent and active participants effectively,
- is scalable both with respect to participants – effective for cross-silo and cross-device deployments, and with respect to geography – single cloud or hybrid cloud or multicloud,
- is efficient, both in terms of resource utilization with support for automatic elastic scaling, and in terms of aggregation latency.

The core technical contribution of this paper is the design, implementation and evaluation of a new FL aggregation system – λ-FL, which has the following novel features:

1. λ-FL reduces state in aggregators and treats aggregators as serverless functions. In many existing FL jobs, every aggregator instance typically acts on a sequence of inputs and produces a single output. State, if present, is not local to the aggregator instance and may be shared by all aggregators. Such state is best left in an external store, and consequently aggregators can be completely

stateless and hence, serverless. This is the case with most federated learning algorithms.

2. λ-FL leverages current serverless technologies to deploy and tear down aggregator instances dynamically in response to participant model updates. There is no reason to keep aggregators deployed all the time and simply "awaiting input".
3. λ-FL is implemented using the popular Ray (Moritz et al. 2018) distributed computing platform, and can run arbitrary Python code in aggregation functions, and use accelerators if necessary.
4. λ-FL uses NO long lived network connections anywhere, and instead uses a message queue to record the presence of model updates. With serverless functions, aggregator instances (at all levels) can appear when there are model updates, and be torn down when their work is finished. Aggregator outputs go back to the message queue, and aggregators (functions) are not coupled to each other.
5. Increased reliability and fault tolerance by reducing state in aggregators, eliminating persistent network connections between aggregators, and through dynamic load balancing of participants.

FL Aggregation, Approaches and Problems

FL Aggregation and Rounds

The federated learning process between parties most commonly uses a single aggregator, which can quickly become a bottleneck and impact the scalability of the FL platform/job. An aggregator would coordinate the overall process, communicate with the parties, and integrate the results of the training process. Most typically, for neural networks, parties would run a local training process on their training data, share the weights of their model with the aggregator, which would then aggregate the weight vectors of all parties using a fusion algorithm. Then, the merged/aggregated model is sent back to all parties for the next round of training.

An FL job thus proceeds over a number of fusion/synchronization rounds, determined by the batch size (B) used. Setting B to ∞ means that each party trains on its entire local dataset before model fusion. For each round, a model update generated by a party is often intended to be ephemeral, but must be durably stored by the aggregator until model fusion is complete, and the aggregated model is durably stored. Model updates may be retained by each party according to its local data retention policy, but the default behavior on the aggregator side is to delete the model updates once the fused model is durably stored. If required by legal or audit regulations, or for model explainability, aggregators may store model updates long term with the permission of parties. Durable storage means reliable replicated distributed storage (like Cloud Object Stores).

Cross-silo vs. Cross-device

Federated Learning is *typically* deployed in two scenarios: *cross-device* and *cross-silo*. The cross-device scenario involves a large number of parties (> 1000), but each party has a small number of data items, constrained compute capability, and limited energy reserve (e.g., mobile phones or IoT devices). They are highly unreliable/asynchronous and are expected to drop and join frequently. Examples include a large organization learning from data stored on employees' devices and a device manufacturer training a model from private data located on millions of its devices (e.g., Google Gboard (Bonawitz et al. 2019)). A trusted authority, which performs aggregation and orchestrates training, is typically present in a cross-device scenario.

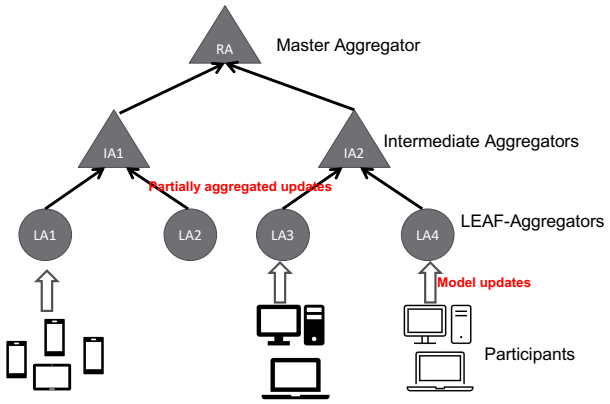

Figure 1: Hierarchical/Tree-based Aggregation

Contrarily, in the cross-silo scenario, the number of parties is small, but each party has extensive compute capabilities (with stable access to electric power and/or equipped with hardware accelerators) and large amounts of data. The parties have reliable participation throughout the entire federated learning training life-cycle, but are more susceptible to sensitive data leakage. Examples include multiple hospitals collaborating to train a tumor detection model on radiographs, multiple banks collaborating to train a credit card fraud detection model, etc. In cross-silo scenarios, there often exists no presumed central trusted authority. The deployments often involve hosting aggregation in public clouds, or alternatively one of the parties acting as, and providing infrastructure for aggregation.

In FL, aggregation is typically provided as a service by one or two organizations, which are trusted by participants to faithfully aggregate model updates. Although recent research has explored peer-to-peer (P2P) aggregation and using blockchain to publish model updates, such techniques, in practice require participants to trust more intermediaries. P2P aggregation involves model updates passing through and visible to multiple entities, requiring participants to trust all of them. Model updates posted to blockchain distributed ledgers are typically public to all participants. Hence, aggregation in a datacenter/cloud, as a software service provided by an organization, is the predominant deployment mode.

Associativity of Aggregation

Given the differences in use cases, deployment environments and performance expectations, different FL aggregation architectures are used for cross-silo and cross-device

FL.

In this scenario, given that the number of participants varies between FL jobs, and within a job (over time) as participants join and leave, horizontal scalability of FL aggregation software is vital. *Horizontally scalable* aggregation is only feasible if the aggregation operation is associative – assuming $\oplus$ denotes the aggregation of model updates (e.g., gradients) U_i, $\oplus$ is associative if $U_1 \oplus U_2 \oplus U_3 \oplus U_4 \equiv (U_1 \oplus U_2) \oplus (U_3 \oplus U_4)$. Associativity enables us to partition participants among aggregator instances, with each instance responsible for handling updates from a subset of participants. The outputs of these instances must be further aggregated. A common design pattern in parallel computing (Kumar 2002) is to use tree-based or hierarchical aggregation in such scenarios, with a tree topology connecting the aggregator instances. The output of each aggregator goes to its parent for further aggregation.

Hierarchical/Tree-based Aggregation

Establishing a tree-based aggregation topology starts by identifying the number of participants that can be comfortably handled by an aggregator instance. This is dependent on (i) size/hardware capability of the instance (CPU/RAM/GPU) and its network bandwidth. This matters irrespective of whether a single physical server or a virtual machine (VM) or a container hosts the instance, and (ii) the size of the model, which directly corresponds to the size of the model update and the memory/compute capabilities needed for aggregation. Assuming that each instance can handle k participants, a complete and balanced k-ary tree can be used; $\lceil \frac{n}{k} \rceil$ leaf aggregators are needed to handle n participants, the tree will have $O(\lceil \frac{n}{k} \rceil)$ nodes.

While a tree-based aggregation overlay is conceptually simple, it does involve significant implementation and deployment effort for fault tolerant aggregation. Typically, the aggregation service provider instantiates aggregator instances using virtual machines (VMs) or containers (e.g., Docker) and manages them using a cluster management system like Kubernetes. These instances are then arranged in the form of a tree, i.e., each instance is provided with the IP address/URL of its parent, expected number of child aggregators, credentials to authenticate itself to said parent and send aggregated model updates. Failure detection and recovery has to be implemented, and is typically done using heartbeats and timeouts, between each instance, its parents and children. Once faults happen, the service provider should typically take responsibility for recovering the instance, and communicating information about the recovered instance to its children for further communications. Things become complicated when an instance fails at the same time as one of its parent or child instances. In summary, the aggregation service provider has to maintain dedicated microservices to deploy, monitor and heal these aggregation overlays. Another issue, common in distributed software systems, that arises in this scenario is network partitions. Yet another factor, in geographically distributed settings, is that aggregator instances may live on different data centers and failure/partition detection and recovery can involve co-ordination between multiple cluster managers.

"Idle Waiting" in Hierarchical Aggregation

Even if some technologies like Kubernetes pods and service abstractions are able to simplify a few of these steps, a more serious problem with tree-based aggregation overlays is that aggregator instances are "always on" waiting for updates, and this is extremely wasteful in terms of resource utilization and monetary cost. To see why keeping aggregators alive is wasteful, it is useful to consider that parties in FL can participate *actively* or *intermittently*.

- *Active* participation means that parties have dedicated resources to the FL job, and will promptly respond to aggregator messages. That is, for every synchronization round, once the aggregator sends the updated model, the party starts the next local training round and sends a (local) model update as soon as it is done. Active participation does not mean specific types of optimization algorithms are used. Generally, active participation is only seen in small scale FL jobs, and more often in cross-silo settings.
- *Intermittent* participation means that for every FL round, each party trains at its convenience; this may be when connected to power in the case of mobile phones, or when (local) resource utilization from other computations is low, or when there are no pending jobs with higher priority. In these scenarios, the aggregator expects to hear from the participant *eventually* (typically over a several hours or maybe once a day). Generally, FL, at scale involves intermittent participants.

To handle FL jobs across thousands of parties, the aggregation service provider must support intermittent parties effectively. Given that, for every round, parties may send model updates over an extended time period (hours), aggregators spend the bulk of their time waiting – most aggregations of model updates are simple arithmetic operations taking seconds. This wastes resources and increases aggregation cost. A tree-based aggregation overlay compounds resource wastage and cost.

Re-configuring tree-based aggregation overlays is also difficult. This is needed, for example, when midway through a job, a hundred (or a thousand) participants decide to join. Supporting them would require reconfiguration at multiple levels of the aggregation overlay. Reconfigurations are also necessary to scale down the overlay when participants leave. Thus, elasticity of aggregation is hard to achieve in the hierarchical setting.

λ-FL : Design and Implementation

The key motivation behind our design is to reduce state in aggregators to decouple aggregator instances. This enables said instances to execute as serverless functions, which are spawned only when model updates arrive, and are torn down when parties are busy training (no updates available to aggregate). An aggregation function instance can be triggered once a specific number of model updates are available; or multiple instances can be triggered once the expected number of model updates for the current FL round are available. Once aggregation is complete and the fused model is sent

back to the parties, all aggregator functions exit, releasing resources.

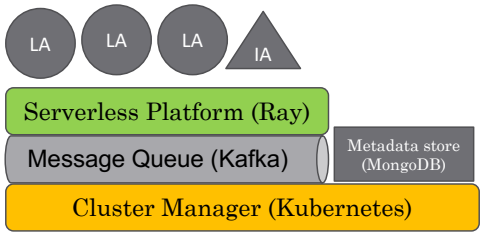

Figure 2: λ-FL System Architecture. Aggregators are executed as serverless functions.

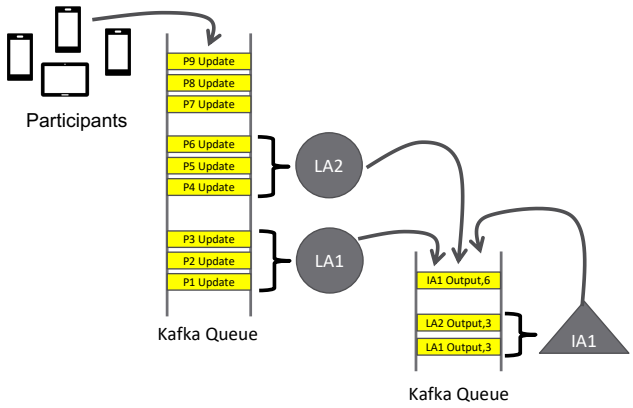

Figure 3: λ-FL – Illustration of stepwise serverless aggregation

Aggregator functions are executed in containers on a cluster managed by Kubernetes, which multiplexes multiple workloads and enables the cluster to be shared by multiple FL jobs and/or other workloads. Also, since there is no static topology, more (or less) aggregator functions can be spawned depending on the number of parties (model updates), thereby handling party joins/leaves effectively. The challenge in executing aggregation as serverless functions, which are ephemeral and have no stable storage, is managing FL job state. We have determined that it is possible to split any associative aggregation operation into the following two components:

1. A (leaf aggregation) function implementing logic to fuse raw model/gradient updates U_i from a group of k parties to generate a partially aggregated model update U_k. For example, in the case of FedSGD or FedAvg (McMahan et al. 2017a,b) where fusion involves simple gradient averaging of gradients, this function would take k_i gradient update vectors and return the sum S_i of these vectors along with k_i. That is, $U_{k_i} = (S_i, k_i)$
2. An (intermediate aggregation) function implementing logic to further aggregate partially aggregated model updates (U_k), in stages, to produce the final aggregated model update (U_F). In the case of FedSGD or FedAvg, this function would aggregate (add up) multiple (S_i, k_i). If all expected model updates have arrived from n parties, i.e., if $\sum_i k_i$ equals n, then the final fused gradient update is $\frac{1}{n} \sum_i S_i$.

We note that such decomposition is fundamental even for hierarchical aggregation to work; λ-FL takes associativity one step further. We also note that splitting aggregation into leaf and intermediate functions makes the logic simpler. It is also possible to have a single serverless function that can operate on both raw updates and partially fused updates; doing that will increase the complexity of the function. For this decoupling split to work, λ-FL employs the following components:

Party-Aggregator Communication This is done using a distributed message queue (Kafka). Kafka is a topic-based message queue offering standard publish/subscribe semantics. That is, each queue has a "name", and multiple distributed entities can write to (publish) and read from (subscribe to) it. Kafka enables us to set a replication level per queue, which ensures durability of messages between the aggregator instances and parties. For each FL job (with an identifier JobID, two queues are created at deployment time – JobID-Agg and JobID-Parties. Only the aggregator can publish to JobID-Agg and all parties subscribe to it. Any party can publish to JobID-Parties but only the aggregator can both publish to and read from it. This ensures that model updates sent to JobID-Parties are private and do not leak to other parties. When the job starts, the aggregator publishes the initial model on JobID-Agg; parties can then download the model and start training. At the end of each job round, parties publish their model updates to JobID-Parties. *Inter-Aggregator Communication,* is also handled using Kafka. Partially fused model updates are published by aggregation functions into Kafka, and can trigger further function invocations.

Aggregation Trigger. For serverless functions to execute, they must be triggered by some event. λ-FL provides several flexible and configurable triggers. The simplest one triggers an aggregation function for every k updates published to JobID-Parties. For FL jobs that use a parameter server for model updates, it is possible in λ-FL to implement the update logic as a serverless function and trigger it every time an update is published by a party. Complicated triggers involve the periodic execution of any valid Python code (also as a serverless function) which triggers aggregation.

Job Metadata Management. Metadata associated with the FL jobs like the number of parties (and the number of updates to expect), timeouts associated with a synchronization round (i.e., how long to wait before assuming that a party is not going to respond or has failed). These help determine when aggregation can terminate. This information is typically stored in external distributed stable storage in the datacenter like a key value store (like cloud object store, MongoDB, Cassandra, ETCD, etc.), in λ-FL as well as in existing single-aggregator and hierarchical systems. This aspect of FL aggregation does not change in λ-FL.

Durability. Aggregation checkpointing for fault tolerance determines how frequently the aggregator checkpoints its state to external stable storage. While this is needed for traditional FL platforms, λ-FL does not use checkpointing. If the execution of a serverless aggregation function fails, it is simply restarted. All aggregator state (updates from parties, partially fused models, etc) is durably stored in message queues. This aspect of λ-FL is vital to understanding λ-FL's resource usage; we observe that the resource overhead of using message queues is equal to that of checkpointing using cloud object stores in single/hierarchical aggregator schemes.

Implementation and Elastic Scaling

We implement λ-FL using the popular Ray (Moritz et al. 2018) distributed computing platform. Ray provides several abstractions, including powerful serverless functions (Ray remote functions). We explored a couple of alternate implementations, including KNative (Kubernetes 2021) and Apache Flink (Carbone et al. 2015), and settled on Ray because it provides arbitrarily long serverless functions, is well integrated with common Python libraries (numpy, scikit-learn, Tensorflow and PyTorch) and provides the freedom to use accelerators if necessary. Ray's internal message queue could have been used in lieu of Kafka, but we found Kafka to be more robust. Aggregation triggers are implemented using Ray, and support typical conditions on JobID-Parties (receipt of a certain number of messages, etc.), but are flexible enough to execute user functions that return booleans (whether aggregation should be triggered or not).

Our implementation using Ray executes on the Kubernetes cluster manager. Ray's elastic scaler can request additional Kubernetes pods to execute serverless functions, depending on how frequently aggregation is triggered. It is also aggressive about releasing unused pods when there are no model updates pending. When aggregation is triggered, groups of model updates are assigned to serverless function invocations. Each invocation is assigned 2 vCPUs and 4GB RAM (this is configurable). If there are insufficient pods to support all these invocations, Ray autoscales to request more Kubernetes pods. This also enables λ-FL to handle large scale party dropouts and joins effectively. Only the exact amount of compute required for aggregation is deployed – overheads to spawn tasks on Kubernetes pods and create new pods are minimal, as demonstrated in our empirical evaluation.

It is also vital to ensure that model updates are not consumed twice by aggregation functions. When aggregation is triggered for a model update in a Kafka queue, it as marked using a flag. The flag is released only after the output of the function is written to Kafka. If the aggregation function crashes, Ray restarts it, thereby guaranteeing "exactly once" processing and aggregation semantics.

Evaluation

In this section, we evaluate the efficacy of using serverless functions for FL aggregation, in λ-FL, by examining whether (1) serverless functions increase the latency of an FL job, as perceived by a participant, and (2) serverless functions decrease the resources needed for aggregation and cost.

Metrics

When compared to a static hierarchical deployment of aggregator instances, serverless functions are dynamically instantiated in response to model updates. Deployment of serverless functions takes a small amount of time (< 100 milliseconds) and elastic scaling of a cluster in response to bursty model update can also take 1-2 seconds. Consequently, the overhead of aggregation in λ-FL will usually manifest in the form of increased *aggregation latency*. Given that aggregation depends on whether the expected number of model updates are available, we define *aggregation latency* as the time elapsed between the reception of the last model update and the availability of the aggregated/fused model/-gradient. It is measured for each FL synchronization round, and the reported numbers in the paper are averaged over all the rounds of the FL job. We want aggregation latency to be as low as possible.

The benefit of using serverless technologies is to improve resource utilization and prevent "idle waiting" for model updates. We execute both hierarchical aggregation and λ-FL using containers on Kubernetes pods in our datacenter, and measure the number of *container seconds* used by an FL job from start to finish. This includes all the resources used by the ancillary services, including MongoDB (for metadata), Kafka and Cloud Object Store. Measuring *container seconds* helps us use publicly available pricing from cloud providers like Microsoft Azure to project the monetary cost of aggregation, in both cases, and project cost savings.

Experimental Setup

Aggregation was executed on a Kubernetes cluster on CPUs, using Docker containers. Each container (hierarchical or serverless) was equipped with 2 vCPUs (2.2 Ghz, Intel Xeon 4210) and 4 GB RAM. For hierarchical aggregation, each instance was encapsulated using the Kubernetes service abstraction. Parties were emulated, and distributed over four datacenters (different from the aggregation datacenter) to emulate geographic distribution. Each party was also executed inside Docker containers (2 vCPUs and 4 GB RAM) on Kubernetes, and these containers had dedicated resources. We actually had parties running training to emulate realistic federated learning, as opposed to using, e.g., Tensorflow Federated simulator.

We used two FL jobs (i) training EfficientNet-B7 (Tan and Le 2019) on CIFAR-100 (Krizhevsky 2009), and (ii) VGG16 (Das et al. 2018) on RVL-CDIP (Harley, Ufkes, and Derpanis 2015) document classification dataset. Our goal here is to measure latency and resource utilization and not invent new FL optimization algorithms. Hence, we partitioned the datasets randomly and uniformly among the parties to create an IID scenario. We used Federated SGD for both the jobs. Each job was executed for 50 synchronization rounds.

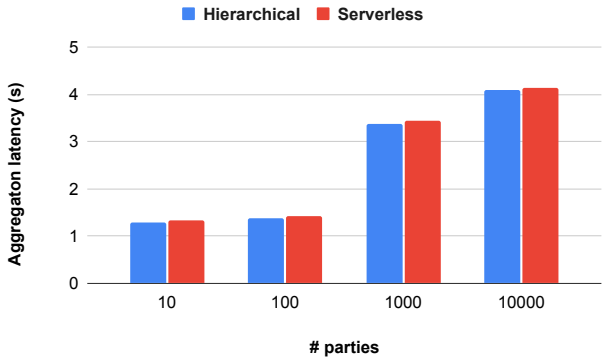

Figure 4: Aggregation Latency (s) – time taken for aggregation to finish after the last model update is available. EfficientNet-B7 and CIFAR100.

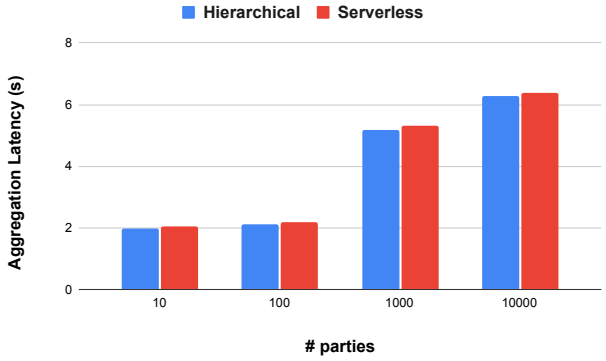

Figure 5: Aggregation Latency (s) – time taken for aggregation to finish after the last model update is available. VGG16 on RVL-CDIP.

Aggregation Latency

First, we consider a scenario where the number of parties remains constant throughout the FL job, for all synchronization rounds. We measure aggregation latency every synchronization round, and Figures 4 and 5 plot the average of these measurements. Figures 4 and 5 illustrate how aggregation latency varies as the number of parties increases; we see an increase in latency, but the trend is sub-linear for both hierarchical and serverless aggregation (latency increases by up to $\approx$ 4 $\times$ when the number of parties increases 1000$\times$). This is expected, due to the data parallelism in hierarchical and serverless aggregation. We also observe that the aggregation latency of λ-FL is very close to that of hierarchical aggregation, and decreases as the absolute value of the latency increases. It is 0.98% for EfficentNet/CIFAR100 and 1.85% for VGG16/RVL-CDIP at 10,000 parties. Thus, we observe that the runtime overhead of using serverless functions is minimal.

Next, we illustrate how λ-FL can handle parties joining in the middle of the job with minimal impact on aggregation latency. For this, we consider a single synchronization round, and increase the number of parties by 20%. Tables 1 and 2 illustrate the aggregation latency when 20% more parties send model updates at the end of the synchronization round. Serverless aggregation needs no overlay reconfiguration, while hierarchical aggregation needs to add more aggregator instances and reconfigure the tree. This manifests as a significant increase in aggregation latency (2.47$\times$ to 3.62$\times$). This is due to the fact that the number of serverless function invocations depends on the aggregation workload, and partially aggregated updates have to be only stored in message queues. However, with a tree overlay, new aggregator nodes have to be instantiated and the topology changed.

# parties	Hierarchical(s)	Serverless(s)	$\frac{Hierarchical}{Serverless}$
100	4.58	1.57	2.92x
1000	12.46	4.34	2.87x
10000	15.59	4.82	3.23x

Table 1: Effect of 20% party joins on aggregation latency (seconds). EfficientNet-B7 and CIFAR100.

# parties	Hierarchical (s)	Serverless (s)	$\frac{Hierarchical}{Serverless}$
100	10.59	4.29	2.47x
1000	17.6	6.45	2.73x
10000	26.82	7.4	3.62x

Table 2: Effect of 20% party joins on aggregation latency (seconds). VGG16 on RVL-CDIP.

Resource Utilization & Cost

Tables 3 and 4 measure the total resource utilization in terms of container seconds for both FL jobs with active participants. These tables illustrate the real benefits of using serverless aggregation, with $> 85\%$ resource and cost savings for the EfficientNet-B7/CIFAR100 job and $> 90\%$ savings for VGG16/RVL-CDIP. We also observe that, while compute resources needed for aggregation increase with the number of participants for both hierarchical and serverless aggregation, the amount of resource and cost savings remains fairly consistent. We use Microsoft Azure's container pricing for illustrative purposes only; pricing is similar for other cloud providers.

We stress that the experiments in Tables 3 and 4 are *conservative*; they assume active participation. That is, parties have dedicated resources to the FL job, parties do not fail in the middle of training, and training on parties for each round starts immediately after a global model is published by the aggregator. In realistic scenarios, parties (e.g., cell phones or laptops or edge devices) perform many functions other than model training, have other tasks to do and can only be expected to respond over a period of time (response timeout). Depending on the deployment scenario, this can be anywhere from several minutes to hours. Table 5 illustrates resource and cost savings when response timeout is set to *a*

# parties	Total container seconds Hierarchical	Serverless	Cost/container/s	Proj. Total cost (USD) Hierarchical	Serverless	Cost Savings (%)
10	1723	228	0.0002692	0.46	0.06	86.96%
100	2653	351	0.0002692	0.71	0.09	87.32%
1000	22340	2951	0.0002692	6.01	0.79	86.86%
10000	298900	40849	0.0002692	80.46	11	86.33%

Table 3: Resource usage and projected cost. EfficientNet-B7 on CIFAR100. Active Participants. Container cost is in USD, obtained from Microsoft Azure's public pricing table (Microsoft 2021).

# parties	Total container seconds Hierarchical	Serverless	Cost/container/s	Proj. Total cost (USD) Hierarchical	Serverless	Cost Savings (%)
10	2002	166	0.0002692	0.54	0.04	91.7
100	3016	229	0.0002692	0.81	0.06	92.41
1000	25254	1937	0.0002692	6.8	0.52	92.33
10000	337782	28543	0.0002692	90.93	7.68	91.55

Table 4: Resource usage and projected cost. VGG16 on RVL-CDIP. Active Participants. Container cost is in USD, obtained from Microsoft Azure's public pricing table (Microsoft 2021).

modest 10 minutes per aggregation round. Thus, our experiments reinforce our confidence that serverless aggregation can lead to significant resource and cost savings with minimal overhead.

Related Work

Parallelzing FL aggregation using a hierarchical topology has been explored by (Bonawitz et al. 2019), though the design pattern was introduced by and early work on datacenter parallel computing (Kumar 2002). While (Bonawitz et al. 2019) uses hierarchical aggregation, its programming model is different from λ-FL. Its primary goal is scalability and consequently, it deploys long lived actors instead of serverless functions. λ-FL aims to make FL aggregation resource efficient, elastic in addition to being scalable; and use off-the-shelf open source software like Ray, Kafka and Kubernetes.

Another closely related concurrent work is FedLess (Grafberger et al. 2021), which predominantly uses serverless functions for the training side (party side) of FL. FedLess is able to use popular serverless technologies like AWS Lambda, Azure functions and Openwhisk to enable clients/parties on cloud platforms perform local training and reports interesting results on using FaaS/serverless instead of IaaS (dedicated VMs and containers) to implement the party side of FL. It also has the ability to run a single aggregator as a cloud function, but does not have the ability to parallelize aggregation, and does not seem to scale beyond 200 parties (with 25 parties updating per FL round, per (Grafberger et al. 2021)). Our work in λ-FL has the primary goal of parallelizing and scaling FL aggregation. Fedless (Grafberger et al. 2021) also does not adapt aggregation based on party behavior, and it is unclear whether parties on the edge (phones/tablets) can train using FedLess.

A number of ML frameworks – Siren (Wang, Niu, and Li 2019), Cirrus (Carreira et al. 2019) and the work by LambdaML (Jiang et al. 2021) use serverless functions for centralized (not federated) ML and DL training. Siren (Wang, Niu, and Li 2019) allows users to train models (ML, DL and RL) in the cloud using serverless functions with the goal to reduce programmer burden involved in using traditional ML frameworks and cluster management technologies for large scale ML jobs. It also contains optimization algorithms to tune training performance and reduce training cost using serverless functions. Cirrus (Carreira et al. 2019) goes further, supporting end-to-end centralized ML training workflows and hyperparameter tuning using serverless functions. LambdaML (Jiang et al. 2021) analyzes the cost-performance trade-offs between IaaS and serverless for datacenter/cloud hosted centralized ML training. LambdaML supports various ML and DL optimization algorithms, and can execute purely using serverless functions or optimize cost using a hybrid serverless/IaaS strategy. λ-FL differs from Siren, Cirrus and LambdaML in significant ways – Distributed ML (in Siren, Cirrus and LambdaML) is different from FL. Distributed ML involves centralizing data at a data center or cloud service and performing training at a central location. In contrast, with FL, data never leaves a participant. FL's privacy guarantees are much stronger and trust requirements much lower than that of distributed ML.

The term "serverless" has also been used to refer to peer-to-peer (P2P) federated learning, as in (Chadha, Jindal, and Gerndt 2020; Niu et al. 2020; Hegedüs, Danner, and Jelasity 2019). In such systems, aggregation happens over a WAN overlay and not in a datacenter. The first step involves establishing the overlay network, by following existing technologies like publish/subscribe overlays, peer discovery, etc (Eugster et al. 2003; Hirzel et al. 2014). The next step involves establishing a spanning tree over the P2P overlay, routing

# parties	Total container seconds Hierarchical	Serverless	Cost/container/s	Proj. Total cost (USD) Hierarchical	Serverless	Cost Savings (%)
10	33043	250	0.0002692	8.9	0.07	99.21%
100	33015	385	0.0002692	8.89	0.1	98.88%
1000	480036	2973	0.0002692	129.23	0.8	99.38%
10000	4500038	40870	0.0002692	1211.41	11	99.09%

Table 5: Resource usage and projected cost. VGG16 on RVL-CDIP. Intermittent participants updating over a 10 minute interval for every synchronization round. Container cost is in USD, obtained from Microsoft Azure's public pricing table (Microsoft 2021).

updates along the spanning tree and aggregating at each node on the tree. Gossip based learning, (Hegedüs, Danner, and Jelasity 2019) does not construct overlays but uses gossip-based broadcast algorithms to deliver and aggregate model updates in a decentralized manner. While these techniques are scalable and (in the case of gossip algorithms) fault tolerant, they do require either (i) that the model be revealed to more entities during routing, or (ii) homomorphic encryption (Jayaram et al. 2020) which can be challenging both from a key agreement and model size explosion standpoints, or (iii) differential privacy (Abadi et al. 2016) which reduces model accuracy in the absence of careful hyperparameter tuning.

Conclusions and Future Work

In this paper, we have presented λ-FL, a system for serverless aggregation in federated learning. We have described the predominant way of parallelizing aggregation using a tree topology and examined its shortcomings. We have demonstrated how serverless/cloud functions can be used to effectively parallelize and scale aggregation while eliminating resource wastage and significantly reducing costs. Our experiments show that the overhead of using serverless for aggregation is minimal, but resource and cost savings are substantial. We also demonstrate that serverless aggregation can effectively handle changes in the number of participants in the FL job.

We are currently working to extend this work in a number of directions: (i) increasing the dependability and integrity of aggregation using trusted execution environments (TEEs), (ii) effectively supporting multi-cloud environments by using service mesh (like Istio) to find the best aggregator function to route a model update to, and (iii) extending serverless aggregation to more optimization algorithms, especially those that employ compression strategies like sketching.

References

Abadi, M.; Chu, A.; Goodfellow, I.; McMahan, H. B.; Mironov, I.; Talwar, K.; and Zhang, L. 2016. Deep Learning with Differential Privacy. In *CCS '16*.

Aono, Y.; Hayashi, T.; Wang, L.; Moriai, S.; et al. 2017. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 13(5): 1333–1345.

Bonawitz, K.; Eichner, H.; Grieskamp, W.; Huba, D.; Ingerman, A.; Ivanov, V.; Kiddon, C.; Konečnỳ, J.; Mazzocchi, S.; McMahan, H. B.; et al. 2019. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*.

Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; and Tzoumas, K. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4): 28–38.

Carreira, J.; Fonseca, P.; Tumanov, A.; Zhang, A.; and Katz, R. 2019. Cirrus: A Serverless Framework for End-to-End ML Workflows. In *SoCC '19*, 13–24. New York, NY, USA: ACM.

Chadha, M.; Jindal, A.; and Gerndt, M. 2020. Towards Federated Learning Using FaaS Fabric. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, 49–54. New York, NY, USA: Association for Computing Machinery. ISBN 9781450382045.

Das, A.; Roy, S.; Bhattacharya, U.; and Parui, S. K. 2018. Document Image Classification with Intra-Domain Transfer Learning and Stacked Generalization of Deep Convolutional Neural Networks. arXiv:1801.09321.

Dayan, I. e. a. 2021. Federated learning for predicting clinical outcomes in patients with COVID-19. *Nature Medicine*.

DLG git repository. 2020. Deep Leakage From Gradients. https://github.com/mit-han-lab/dlg.

Eugster, P. T.; Felber, P. A.; Guerraoui, R.; and Kermarrec, A.-M. 2003. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 114–131.

Geiping, J.; Bauermeister, H.; Dröge, H.; and Moeller, M. 2020. Inverting Gradients–How easy is it to break privacy in federated learning? *arXiv preprint arXiv:2003.14053*.

Grafberger, A.; Chadha, M.; Jindal, A.; Gu, J.; and Gerndt, M. 2021. FedLess: Secure and Scalable Federated Learning Using Serverless Computing. arXiv:2111.03396.

Harley, A. W.; Ufkes, A.; and Derpanis, K. G. 2015. Evaluation of Deep Convolutional Nets for Document Image Classification and Retrieval. In *International Conference on Document Analysis and Recognition*, 991–995. IEEE.

Hegedüs, I.; Danner, G.; and Jelasity, M. 2019. Gossip Learning as a Decentralized Alternative to Federated Learning. In *DAIS*.

Hirzel, M.; Soulé, R.; Schneider, S.; Gedik, B.; and Grimm, R. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.*, 46(4).

iDLG git repository. 2020. Improved Deep Leakage from Gradients. https://github.com/PatrickZH/Improved-Deep-Leakage-from-Gradients.

IG git repository. 2020. Inverting Gradients - How easy is it to break Privacy in Federated Learning? https://github.com/JonasGeiping/invertinggradients.

Jayaram, K. R.; Verma, A.; Verma, A.; Thomas, G.; and Sutcher-Shepard, C. 2020. MYSTIKO: Cloud-Mediated, Private, Federated Gradient Descent. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 201–210. IEEE.

Jiang, J.; Gan, S.; Liu, Y.; Wang, F.; Alonso, G.; Klimovic, A.; Singla, A.; Wu, W.; and Zhang, C. 2021. Towards Demystifying Serverless Machine Learning Training. In *ACM SIGMOD*.

Kairouz, P.; McMahan, H. B.; Avent, B.; Bellet, A.; Bennis, M.; Bhagoji, A. N.; Bonawitz, K.; Charles, Z.; Cormode, G.; Cummings, R.; et al. 2019. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*.

Krizhevsky, A. 2009. Learning Multiple Layers of Features from Tiny Images. 32–33.

Kubernetes. 2021. Enterprise-grade Serverless on your own terms. https://knative.dev/docs/.

Kumar, V. 2002. *Introduction to Parallel Computing*. USA: Addison-Wesley Longman Publishing Co., Inc., 2nd edition. ISBN 0201648652.

Liu, Y.; Fan, T.; Chen, T.; Xu, Q.; and Yang, Q. 2021. FATE: An Industrial Grade Platform for Collaborative Learning With Data Protection. *Journal of Machine Learning Research*, 22(226): 1–6.

Ludwig, H.; Baracaldo, N.; Thomas, G.; Zhou, Y.; Anwar, A.; Rajamoni, S.; Ong, Y.; Radhakrishnan, J.; Verma, A.; Sinn, M.; Purcell, M.; Rawat, A.; Minh, T.; Holohan, N.; Chakraborty, S.; Whitherspoon, S.; Steuer, D.; Wynter, L.; Hassan, H.; Laguna, S.; Yurochkin, M.; Agarwal, M.; Chuba, E.; and Abay, A. 2020. IBM Federated Learning: an Enterprise Framework White Paper V0.1. *arXiv preprint arXiv:2007.10987*.

McMahan, B.; Moore, E.; Ramage, D.; Hampson, S.; and y Arcas, B. A. 2017a. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, 1273–1282. PMLR.

McMahan, H. B.; Ramage, D.; Talwar, K.; and Zhang, L. 2017b. Learning differentially private recurrent language models. *arXiv preprint arXiv:1710.06963*.

Microsoft. 2021. Azure Container Instances pricing. https://azure.microsoft.com/en-us/pricing/details/container-instances/.

Moritz, P.; Nishihara, R.; Wang, S.; Tumanov, A.; Liaw, R.; Liang, E.; Elibol, M.; Yang, Z.; Paul, W.; Jordan, M. I.; and Stoica, I. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*. USENIX Association.

Niu, C.; Wu, F.; Tang, S.; Hua, L.; Jia, R.; Lv, C.; Wu, Z.; and Chen, G. 2020. Billion-Scale Federated Learning on Mobile Clients: A Submodel Design with Tunable Privacy. In *ACM MobiCom '20*.

NVIDIA. 2021. NVIDIA Federated Learning Application Runtime Environment. https://github.com/NVIDIA/NVFlare.

Tan, M.; and Le, Q. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, 6105–6114. PMLR.

Wang, H.; Niu, D.; and Li, B. 2019. Distributed Machine Learning with a Serverless Architecture. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 1288–1296.

Zhao, B.; Mopuri, K. R.; and Bilen, H. 2020. iDLG: Improved Deep Leakage from Gradients. *arXiv preprint arXiv:2001.02610*.

Zhu, L.; Liu, Z.; and Han, S. 2019. Deep leakage from gradients. In *Advances in Neural Information Processing Systems*, 14774–14784.