On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures

Daniel Barcelona-Pons
Universitat Rovira i Virgili
daniel.barcelona@urv.cat

Marc Sánchez-Artigas
Universitat Rovira i Virgili
marc.sanchez@urv.cat

Gerard París
Universitat Rovira i Virgili
gerard.paris@urv.cat

Pierre Sutra
Télécom SudParis
pierre.sutra@telecom-sudparis.eu

Pedro García-López
IBM T.J. Watson Research Center
Universitat Rovira i Virgili
pedro.garcia.lopez@ibm.com

Abstract

Serverless computing is an emerging paradigm that greatly simplifies the usage of cloud resources and suits well to many tasks. Most notably, Function-as-a-Service (FaaS) enables programmers to develop cloud applications as individual functions that can run and scale independently. Yet, due to the disaggregation of storage and compute resources in FaaS, applications that require fine-grained support for mutable state and synchronization, such as machine learning and scientific computing, are hard to build.

In this work, we present Crucial, a system to program highly-concurrent stateful applications with serverless architectures. Its programming model keeps the simplicity of FaaS and allows to port effortlessly multi-threaded algorithms to this new environment. Crucial is built upon the key insight that FaaS resembles to concurrent programming at the scale of a data center. As a consequence, a distributed shared memory layer is the right answer to the need for fine-grained state management and coordination in serverless. We validate our system with the help of micro-benchmarks and various applications. In particular, we implement two common machine learning algorithms: k-means clustering and logistic regression. For both cases, Crucial obtains superior or comparable performance to an equivalent Spark cluster.

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

Middleware '19, December 8–13, 2019, Davis, CA, USA

ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3361535

Keywords Serverless, FaaS, in-memory, stateful, synchronization

ACM Reference Format:
Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *20th International Middleware Conference (Middleware '19), December 8–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3361525.3361535

1 Introduction

With the emergence of serverless computing, the cloud has found a paradigm that removes much of the complexity of its usage by abstracting away the provisioning of compute resources. This fairly new model was started by services such as Google BigQuery [40] and AWS Glue [5], and evolved into Function-as-a-Service (FaaS) computing platforms, such as AWS Lambda, and Google's Cloud Functions, to name a few. With these platforms, a user-defined function and its dependencies are deployed to the cloud, where they are managed by the provider and executed on-demand.

Current practice shows that the FaaS model works well for applications that require a small amount of storage and memory due to the operational limits set by the cloud providers (see, for instance, AWS Lambda [3]). However, there are more limitations. While functions can initiate outgoing network connections, they cannot directly communicate between each other, and have little bandwidth compared to a regular virtual machine [9, 53]. This is because this model was originally designed to execute event-driven, stateless functions in response to user actions or changes in the storage tier (e.g., uploading a photo to Amazon S3 [2]). Despite these constraints, recent works have shown how this model can be exploited to process and transform large amounts of data [25, 42, 44], encode videos [15], execute linear algebra tasks [46], and perform Monte Carlo simulations with large amounts of parallelism [23].

The above research projects, such as PyWren [25, 44] and ExCamera [15], prove that FaaS platforms can be programmed to perform a wide variety of embarrassingly parallel computations. Yet, these tools face also fundamental challenges when used out-of-the-box for many popular tasks. Although the list is too long to recount here, convincing cases of these ill-suited applications are machine learning (ML) algorithms. Just an imperative implementation of *k*-means [33] raises several issues: first, the need to efficiently handle a globally-shared state at fine granularity (the cluster centroids); second, the problem to globally synchronize cloud functions, so that the algorithm can correctly proceed to the next iteration; and finally, the prerogative that the shared state survives system failures.

Current serverless systems do not address these issues effectively. First, due to the impossibility of function-to-function communication, the prevalent practice for sharing state across functions is to use remote storage. For instance, serverless frameworks, such as PyWren [25, 44] and numpywren [46], use highly-scalable object storage services to transfer state between functions. Since object storage is too slow to share short-lived intermediate state in serverless applications [31], some recent works use faster storage solutions. For instance, this has been the path taken by Locus [42], which proposes to combine fast, in-memory storage instances with slow storage to scale shuffling operations in MapReduce. However, with all the shared state transiting through storage, one of the major limitations of current serverless systems is the lack of support to handle mutable state at a fine granularity (e.g., to efficiently aggregate small granules of updates). Such a concern has been recognized in various works [9, 26], but this type of fast, enriched storage layer for serverless computing is not available today in the cloud, leaving fine-grained state sharing as an open issue.

Similarly, FaaS orchestration services (such as AWS Step Functions [4] or OpenWhisk Composer [16]) offer limited capabilities to coordinate serverless functions [17, 26]. For instance, there is no abstraction to signal a function when a condition is fulfilled, or for multiple functions to synchronize, e.g., in order to guarantee data consistency, or to ensure joint progress to the next stage of computation. Of course, such fine-grained coordination should be also low-latency to not significantly slow down the application. Existing standalone notification services, such as AWS SNS [8] and AWS SQS [18], add significant latency, sometimes hundreds of milliseconds. This lack of efficient cloud coordination tools means that each serverless framework needs to develop its own mechanisms. For instance, PyWren [25, 44] enforces the synchronization of map and reduce stages through object storage, while ExCamera [15] has built a notification system using a long-running VM-based rendezvous server. As of today, there is no general way to let multiple functions synchronize via abstractions hand-crafted by users, so that fine-grained coordination can be truly achieved.

1.1 Contributions

To overcome the aforementioned issues, we propose Crucial, a system for the development of stateful distributed applications with serverless architectures. To simplify the writing of an application, Crucial provides a thread abstraction that maps a thread to the invocation of a serverless function. To support fine-grained state management and coordination, our system builds a distributed shared object (DSO) layer on top of a low-latency in-memory data store. This layer provides out-of-the-box strong consistency guarantees, simplifying the semantics of global state mutation across serverless threads. Since global state is manipulated as remote objects, the interface for mutable state management becomes virtually unlimited, only constrained by the expressiveness of the programming language (Java in our case). The result is that Crucial can operate on small data granules, making it very easy to develop applications that have fine-grained state sharing needs. Crucial also leverages this layer to implement fine-grained coordination. For applications that require longer retention of in-memory state, Crucial ensures data durability through replication. To ensure the consistency of replicas, Crucial uses state machine replication (SMR), so that any acknowledged write can survive failures.

Most importantly, Crucial offers all of the above guarantees with almost no increase in the programming complexity of the serverless model. With the help of a few annotations and constructs, developers can run their single-machine, multi-threaded, stateful code in the cloud as serverless functions. Crucial's programming constructs enable developers to enforce atomic operations on shared state, as well as to finely synchronize functions at the application level, so that (imperative) implementations of popular algorithms such as *k*-means can be effortlessly ported to serverless platforms.

Our evaluation shows that, for representative applications that require fine-grained updates (e.g., *k*-means, logistic regression), Crucial can rival, and even outperform, Spark running on a dedicated cluster. We also establish that Crucial induces a very small overhead, and that it can effectively be used for fine-grained coordination (e.g., to solve the Santa Claus problem [51]).

In summary, the present work makes the following contributions:

- We provide the first concrete evidence that stateful applications with fine-grained data sharing can be efficiently built using stateless serverless functions and a disaggregated shared object layer.
- We design Crucial, a system for the development and execution of serverless stateful distributed applications. Crucial supports fine-grained semantics for both mutable state and coordination. Moreover, the programming model of Crucial keeps the simplicity offered by current serverless architectures. These

properties make Crucial a great tool to easily move multi-threaded applications to the cloud.
- Using extensive evaluation of *k*-means and logistic regression over a 100GB dataset, we show that Crucial can lead to 18 – 40% performance improvement over Spark running on dedicated instances at similar cost. Crucial is also within 8% of the completion time of the Santa Claus problem running on a local machine.

The remaining of the paper is structured as follows: Section 2 puts this work in perspective with a general background. We explain Crucial's programming model in Section 3, and describe its design in Section 4. Section 5 covers some implementation details. The evaluation is presented in Section 6, where we validate Crucial through micro-benchmarks and assess its effectiveness for fine-grained state management and coordination in serverless environments. Finally, we review related work in Section 7 and close in Section 8.

2 Background

2.1 FaaS computing: *What fits?*

We contextualize FaaS computing with a description of AWS Lambda; although other platforms are equally well-suited for this purpose (e.g., Google Cloud Functions, Azure Functions, or Apache OpenWhisk).

AWS Lambda is a cloud service designed to run user-supplied cloud functions—or *Lambdas*—in response to events (e.g., file uploads, message arrivals, etc.), or explicit API calls (via HTTP requests). AWS Lambda, as other FaaS computing platforms, gives the favorable advantages of rapid provisioning, high elasticity and just-right cost: containers used for function deployment can be launched within a few seconds; they can quickly scale up or down to match demand; and the service charges for the duration of their execution at the granularity of milliseconds. All these properties make possible to run arbitrary workloads in the cloud with minimal overhead [25, 42, 44, 46].

However, due to their lightweight nature, cloud functions are also subject to stringent resource restrictions. For instance, AWS Lambda [3] imposes a 15 minute limit per function invocation and caps memory usage to 3GB. Similar limits are applied by other FaaS providers. In addition, while a user can execute functions concurrently, peer-to-peer communication is impossible between them. As a consequence, the linear scalability in function execution is in practice only achievable for embarrassingly parallel tasks [20, 26].

Function invocations can also fail for various reasons (e.g., it raises an exception, times out, or runs out of memory). When an error occurs, the AWS Lambda service may automatically retry the failed invocation.[1] This requires the developer to consider carefully such a behavior when designing FaaS applications.

[1]See: https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html

2.2 The dilemma of shared data

This work focuses on the case of stateful distributed applications with fine-grained updates and synchronization needs. Cloud functions are assumed to be *stateless*, and as such, they do not provide support for arbitrary mutable state. This stateless nature, together with the impossibility to communicate across functions, has encouraged the use of remote storage [26, 54].

So far, the prevalent choice has been to rely on disaggregated object storage such as Amazon S3 or Google Cloud Storage. Typically, object stores have high access latency (>10ms) and deliver either limited or costly I/O performance [26, 54]. Consequently, most serverless frameworks, like PyWren [25], only allow coarse-grained operations to shared data.

To alleviate the above problem, some recent works [39, 42, 46] make use of their own in-memory storage instances. This type of storage offers low-latency but it is not fault-tolerant and does not support convenient abstractions to synchronize cloud functions. In addition, these systems may not provide guarantees regarding data persistence and availability.

Another recurring problem is the need to ship data to code. Existing serverless frameworks access data using storage services that either offer a CRUD interface or provide limited sets of data types. As a consequence, data is repeatedly transported back and forth between the cloud functions and the storage layer. This negatively impacts performance (especially for large objects) and restrains concurrency on shared data.

2.3 An overview of Crucial

Crucial apprehends a simplified view of FaaS where cloud functions are seen as a set of "cloud threads" that communicate through shared state. To achieve this, our framework organizes mutable shared data in a layer of distributed shared objects (DSO). Cloud functions remotely call the methods of the objects to read/update them at fine granularity.

The DSO layer is implemented within a low-latency in-memory data store and deployed jointly with the serverless application. It delivers sub-millisecond latency—like other in-memory systems such as Redis (see Table 2)—and achieves even better throughput for complex, CPU-bound, concurrent operations (see Fig. 2a). Both properties, low-latency and high-throughput, make it an excellent substrate for mutable shared state and synchronization. Crucial also permits data to persist after the computation, ensuring their durability through replication.

Although the idea of distributed objects is not novel, to the best of our knowledge, it was never applied to serverless computing. Such an approach simplifies the programming of stateful applications atop serverless architectures and further closes the gap between cloud and conventional computing. The next two sections describe the programming model of Crucial and its internals.

Table 1. Programming abstractions

Abstraction	Description
`CloudThread`	Serverless functions are invoked like threads.
Shared objects	Linearizable (wait-free) distributed objects. `AtomicInt`, `AtomicLong`, `AtomicBoolean`, `AtomicByteArray`, `List`, `Map`, ...
Synchronization objects	Shared objects providing primitives for thread synchronization (e.g., `CyclicBarrier`, `Semaphore`, `Future`).
`@Shared`	User-defined shared object. Methods are run on the DSO servers, allowing fine-grained updates and aggregates (`.add()`, `.update()`, `.merge()`, ...).
Data persistence	Long-lived shared objects are replicated. Use `@Shared(persistence=true)` to activate it.

3 Using Crucial

3.1 Programming model

Crucial's programming model is object-based and can be integrated with any concurrent object-oriented language. As Java is the programming language supported in our implementation, the following description considers its jargon.

Overall, a Crucial program is strongly similar to a regular multi-threaded, object-oriented Java one, besides some additional annotations and constructs. Table 1 summarizes the key programming abstractions available to developers that are detailed hereafter.

Cloud threads To write a stateful application for serverless architectures, a programmer first builds its logic as a regular multi-threaded, object-oriented Java program. Then, two refinements are necessary to make it executable atop the FaaS model. First, each runnable object is associated with a `CloudThread`. An instance of this class hides the execution details of the remote cloud function to the developer. The second modification is to replace each mutable object shared between threads with its Crucial counterpart.

State handling Crucial already includes a library of base shared objects to support mutable shared data across cloud threads. This library consists of common objects such as integers, counters, maps, lists and arrays. These objects are *wait-free* and *linearizable* [34]. This means that each method invocation terminates after a finite amount of steps (despite concurrent accesses), and that concurrent method invocations behave as if they were executed by a single thread. Crucial also gives programmers the ability to craft their own custom shared objects by decorating them with the `@Shared` annotation. Annotated objects become globally accessible by any thread. Crucial refers to an object with a key crafted from the field's name of the encompassing object. The programmer can override this definition by explicitly writing `@Shared(key=k)`.

Data Persistence Shared objects in Crucial can be either *ephemeral* or *persistent*. By default, shared objects are ephemeral and they only exist during the application lifetime. Once the application finishes, they are discarded. Ephemeral objects can be lost, e.g., in the event of a server failure in the DSO layer, since the cost of making them fault-tolerant outweighs the benefits of their short-term availability [31]. Nonetheless, it is also possible to make them persistent with the annotation `@Shared(persistent=true)`. Annotated objects outlive the application lifetime and are only removed from storage by an explicit call.

```
1  public class PiEstimator implements Runnable{
2   private final static long ITERATIONS = 100_000_000;
3   private Random rand = new Random();
4   @Shared(key="counter")
5   crucial.AtomicLong counter = new crucial.AtomicLong(0);
6
7   public void run(){
8    long count = 0;
9    double x, y;
10   for (long i = 0L; i < ITERATIONS; i++) {
11    x = rand.nextDouble();
12    y = rand.nextDouble();
13    if (x * x + y * y <= 1.0) count++;
14   }
15   counter.addAndGet(count);
16  }
17 }
18
19 List<Thread> threads = new ArrayList<>(N_THREADS);
20 for (int i = 0; i < N_THREADS; i++) {
21  threads.add(new CloudThread(new PiEstimator()));
22 }
23 threads.forEach(Thread::start);
24 threads.forEach(Thread::join);
25 double output = 4.0 * counter.get() / (N_THREADS * ITERATIONS);
```

Listing 1. Monte Carlo simulation to approximate π.

Synchronization Current serverless frameworks support only uncoordinated embarrassingly parallel operations, or bulk synchronous parallelism (BSP) [20, 26]. To provide fine-grained coordination of cloud threads, Crucial offers a number of primitives such as cyclic barriers and semaphores (see Table 1 and an example at line 2 in Listing 2). These coordination primitives are semantically equivalent to those in the standard `java.util.concurrent` library. They allow a coherent and flexible model of concurrency for serverless functions that is, as of today, non-existent.

3.2 Sample application

Listing 1 presents an application implemented with Crucial. This simple program is a multi-threaded Monte Carlo simulation that approximates the value of π. It draws a large number of random points and computes how many fall in the circle enclosed by the unit square. The ratio of points falling in the circle converges with the number of trials toward $\pi/4$ (line 25).

The application first defines a regular `Runnable` class that carries the estimation of π (lines 1-19). To parallelize its execution, lines 23-24 run a fork-join pattern using a set of `CloudThread` instances. The shared state of the application is a `counter` object (line 5). This counter maintains the total

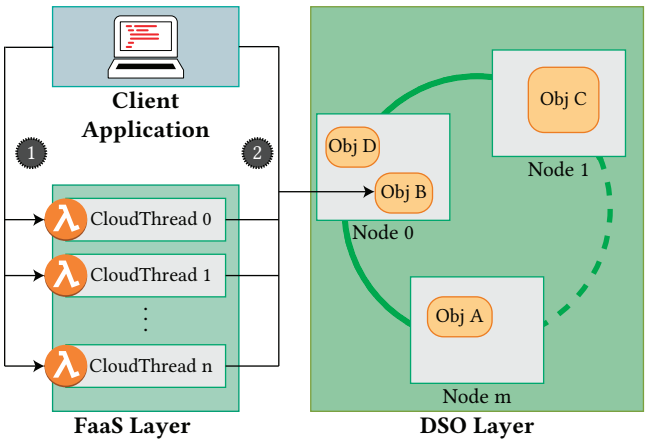

Figure 1. Overall architecture of Crucial. A client application would run a set of threads in FaaS functions, and all the threads would have access to the same state (client included).

number of points falling into the circle, which serves to approximate π. It is updated by the threads concurrently using the `addAndGet` method (line 15).

4 System Design

Figure 1 presents the overall architecture of Crucial. In what follows, we detail its components and describe the lifecycle of an application in our system. Crucial encompasses three main components: 1) the FaaS computing layer that runs the cloud threads; 2) the DSO layer that stores the shared objects; and 3) the client application. A client application differs from a regular JVM process on two aspects: threads are executed as serverless functions, and they access shared data using the DSO layer. In addition, Crucial may use object storage (such as Amazon S3) to store the immutable input data of the application (not modeled in Figure 1).

4.1 The distributed object layer

In Crucial, fine-grained updates to a data item are implemented as object methods. Internally, each object in the DSO layer is uniquely identified by a reference. Given an object of type T, the reference to this object is (T, k), where k is either the field's name of the encompassing object or the value of the parameter *key* in the annotation `@Shared(key=k)`. When a cloud thread accesses an object, it uses its reference to invoke remotely the appropriate method. Crucial constructs the DSO layer using consistent hashing [28], similarly to Cassandra [32]. Each storage node knows the full membership of the storage layer and thus the mapping from data to node. The location of a shared object o is then determined by hashing the reference (T, k) of o. This offers the following usual benefits: 1) no broadcast is necessary to locate an object; 2) disjoint-access parallelism [24] can be exploited; and 3) service interruption is minimal in the event of server addition and removal. The latter property is useful for persistent objects, as detailed next.

Persistence One interesting aspect of Crucial is that it can ensure durability of the shared state. This property is appealing, for instance, to support the different phases of machine learning workflows (training and inference). Objects marked as persistent are replicated *rf* (replication factor) times in the DSO layer. They reside in memory to ensure sub-millisecond read/write latency and can be passivated to stable storage using standard mechanisms (marshalling). When a cloud thread accesses a shared object, it contacts one of the server nodes. The operation is then forwarded to the actual replicas storing the object. Each replica executes the incoming call, and one of them sends the result back to the caller. Notice that for ephemeral—non-persistent—objects, *rf* is 1.

Consistency Crucial provides linearizable objects and developers can reason about interleavings as in the shared-memory case. This greatly simplifies the writing of stateful serverless applications. For persistent objects, consistency across replicas is maintained with the help of state machine replication (SMR) [45]. To handle membership changes, the DSO layer relies on a variation of view synchrony [10] View synchrony provides a totally-ordered set of views to the server nodes. In a given view, for some object x, the operations accessing x are sent using total order multicast. The replicas of x deliver these operations in a total order and apply them on their local copy of x according to this order. A distinct replica (primary) is in charge of sending back the result to the caller. When a membership change occurs, the nodes re-balance data according to the new view.

4.2 Fast aggregates through method call shipping

Crucial helps to alleviate perhaps one of the biggest downsides of FaaS platforms: its data-shipping architecture [20]. As functions are not network-addressable and run separate from data, applications are routinely left with no other choice but to "ship data to code". Fortunately, the DSO layer helps to resolve this design anti-pattern with minimal effort from the user side: it suffices to implement arbitrary computations as object methods. This feature is extremely useful for many applications that need to aggregate and combine *small granules of data* (e.g., machine learning tasks). As object methods are remotely executed on the DSO servers, applications can save significant communication resources. Without this property, each cloud function would need first to pull all the intermediate data from the remote storage service (e.g., S3) and then aggregate it locally (i.e., `AllReduce` operation). This would entail a communication cost of N^2 messages, where N is the number of functions. With Crucial, however, this complexity reduces to $O(N)$ messages. For instance, we exploited this feature in k-means clustering to calculate the

final centroids from their partial updates. The performance benefits are detailed in Section 6.2.

4.3 Execution lifecycle

The execution lifecycle of a Crucial application is similar to that of a multi-threaded Java application. Every time a `CloudThread` is started, a standard Java thread (i.e., instance of `java.lang.Thread`) is spawned in the client application with some extra logic. The basic role of this logic lies in calling a generic serverless function to execute the `Runnable` code attached to the `CloudThread`. During the execution of a cloud thread, each access to a shared object is mediated by a proxy. This proxy is created when a constructor is encountered in the code, and either the newly created object belongs to Crucial's library, or it is tagged `@Shared`.

The Java thread remains blocked until the call to the serverless function terminates. Such behavior gives cloud threads the appearance of conventional threads; minimizing code changes and allowing the use of the `join()` method in the application's master thread to establish synchronization points (e.g., fork/join pattern). It must be noted, however, that as cloud functions cannot be canceled or paused, the analogy is not complete. If any failure occurs to the remote cloud function, the error is propagated back to the client application for further processing.

4.4 Fault tolerance

As detailed next, fault tolerance in Crucial is based on the disaggregation of the compute and storage layers. On the one hand, writes to the shared object layer can be made durable with the help of data replication. In such a case, Crucial tolerates the joint failure of up to $rf - 1$ servers.[2] On the other hand, Crucial offers the same fault-tolerance semantics in the compute layer as the underlying FaaS platform. In AWS Lambda, this means that any failed cloud thread can be re-started and re-executed with the exact same input. Thanks to the cloud thread abstraction, Crucial allows full control over the retry system. For instance, the user may configure how many retries are allowed and/or the time between them. If retries are permitted, the programmer should ensure that the re-execution is sound (e.g., it is idempotent). Fortunately, atomic writes in the DSO layer make this task easy to achieve. Considering the *k*-means example depicted in Listing 2 (or other iterative algorithms), it simply consists of sharing an iteration counter. When a thread fails and re-starts, it fetches the iteration counter and continues its execution from thereon.

[2]Synchronization objects (see Table 1) are not replicated. This is not an important issue due to their ephemeral nature.

5 Implementation

Crucial applications are written in Java and use Maven to manage dependencies and compilation. Cloud threads are defined by implementing a `Runnable` and executed with the abstraction from Table 1. Our system uses AWS Lambda as computation engine for the cloud threads. Lambda functions are deployed with the help of the `lambda-maven-plugin`[3] and invoked through the AWS Java SDK. To control the replay mechanism, our prototype uses synchronous invocations (`RequestResponse`).

When an AWS Lambda function is invoked, it receives in the payload the name of the user-defined `Runnable` and a set of parameters to initialize it. We use the Java reflection API to instantiate the classes and provide them the initialization values. Before executing the user code, our generic function establishes the connection to the DSO layer. Since our prototype only accepts `Runnable`, the return payload is empty unless an error occurs. In case of error, the system interprets it and re-throws an exception.

The DSO layer is written atop the Infinispan in-memory data grid [35] as a partial rewrite of the Creson project [49]. The code of the client and the server of this layer weigh 2.5k and 9.2k SLOC, respectively. The client prototype includes a small library of shared objects and the proxies to access them. To wave proxies in the code of the client application and the cloud functions, both are compiled with the help of AspectJ [29]. In the case of user-defined shared objects, the aspects are applied to annotated instance fields (see Section 3.1). Such objects must be serializable and contain an empty constructor for marshalling purposes. The `jar` package containing the objects is uploaded to the DSO servers. This package is then loaded dynamically, without having to restart the servers.

Synchronization objects (e.g., barriers, semaphores, futures) follow the structure of their Java counterparts. They rely on Java monitors, blocking upon a method call at a client, and using a combination of `wait()`/`notify()` at servers. For instance, the cyclic barrier is implemented thanks to an internal counter and a generation system. A new generation is started once the last process reaches the barrier.

SMR is implemented using the interceptors API of Infinispan.[4] It follows the visitor pattern as commonly found in storage systems. Infinispan [35] relies on JGroups [19] for total order multicast. The current implementation uses Skeen's algorithm [7].

In our current prototype, the deployment of the storage layer is explicitly managed (like, e.g., AWS ElastiCache service). Automatic provisioning of storage resources for serverless computing remains an open issue [9, 26], with just a couple works appearing very recently in this area [31, 42].

[3]https://github.com/SeanRoy/lambda-maven-plugin

[4]The interceptors API enables the execution of custom code in-between Infinispan's processing of data store operations.

Table 2. Average latency comparison – *1KB payload*

	PUT	GET
S3	34,868μs	23,072μs
Redis	232μs	229μs
Infinispan	228μs	207μs
Crucial	231μs	229μs
Crucial (rf = 2)	512μs	505μs

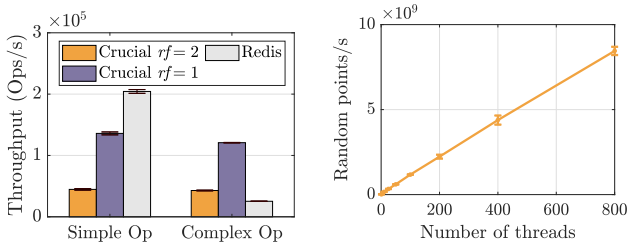

Figure 2. (a) Operations per second performed in Crucial (with and without replication) and Redis. The simple operation is a multiplication. The complex one is the sequential execution of 10k multiplications. Cloud threads access uniformly at random 800 different keys/objects. (b) Scalability of a Monte Carlo simulation to approximate π. Crucial reaches 8.4 billion random points per second with 800 threads.

6 Evaluation

This section presents experimental results assessing that our approach is not only feasible but also desirable for certain types of applications, e.g., machine learning (ML).

We first validate the general design of Crucial with a series of micro-benchmarks. Next, we show that our system based on fine-grained updates to shared mutable data outperforms Spark at comparable cost in two instances of ML problems. Further, we outline the benefits of Crucial when coordinating serverless functions The end of this section explains quantitatively how Crucial simplifies the programming of multi-threaded stateful applications over a serverless infrastructure.

Evaluation setup All the experiments are conducted in Amazon Web Services (AWS), within in a Virtual Private Cloud (VPC) located in the `us-east-1` region. Unless otherwise specified, we use `r5.2xlarge` EC2 instances for the DSO layer and the maximum resources available for AWS Lambda.[5]

6.1 Micro-benchmarks

First, we evaluate Crucial's performance across a range of micro-benchmarks.

6.1.1 Latency

Table 2 compares the latency to access a 1KB object sequentially in Crucial, Redis, Infinispan and S3. We chose Redis because it is a popular key-value store available on almost all cloud platforms, and it has been extensively used as storage substrate in prior serverless systems [25, 31, 42]. Each function performs 30k operations and we display the average latency of an access. In this test, Crucial exhibits a performance similar to other in-memory systems. In particular, it is an order of magnitude faster than S3. This table also depicts the effect of object replication. When data is replicated, SMR adds an extra round-trip, doubling the latency perceived at a client.

6.1.2 Throughput

Figure 2a compares the performance of Crucial and Redis for both simple and complex operations. In this experiment, 200 cloud threads access remotely 800 objects at random in closed loop. Each remote object consists of an integer with basic arithmetic operations. The experiment runs for 30s and we present the average performance. The storage layer consists of a two-node cluster for both Crucial (with and without replication), and Redis (2 shards with no replicas).

The key observation in Figure 2a is that Crucial is not sensitive to the complexity of the operation. Redis is 50% faster for base operations because its implementation is optimized and written in C. However, for complex operations, the performance of Crucial is almost five times better than Redis. Again, implementation-specific details are responsible for this behavior: while Redis is single-threaded—so concurrent calls (Lua script) run sequentially—, Crucial benefits from disjoint-access parallelism [24]. With replication, Crucial is 70% faster.

6.1.3 Parallelism

Our first application using Crucial is the Monte Carlo simulation presented in Listing 1. This base algorithm is embarrassingly parallel, relying only on a single shared object (a counter). We run the simulation with 1 to 800 cloud threads and track the total number of points computed by them each second. The results presented in Figure 2b show that our system scales linearly and that it exhibits a 512x speedup with 800 threads.

6.2 Fine-grained state management

This section shows that Crucial is efficient for parallel applications that access a shared state at fine grain. To this end, we describe the implementation of two machine learning algorithms in Crucial and compare them to a single machine solution and Spark.

[5] 3008MB of memory at the time of writing.

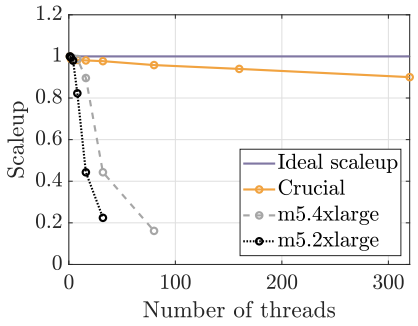

Figure 3. Scalability of the *k*-means clustering algorithm with VM threads versus FaaS with Crucial.

```
1  public class KMeans implements Runnable{
2   private CyclicBarrier barrier = new crucial.CyclicBarrier();
3   @Shared(key = "delta")
4   private GlobalDelta globalDelta = new GlobalDelta();
5   @Shared(key = "iterations")
6   private AtomicInteger globalIterCount = new AtomicInteger();
7   // Wraps a list of @Shared centroids
8   private GlobalCentroids centroids = new GlobalCentroids();
9
10  public void run(){
11   loadDatasetFragment();
12   int iterCount = globalIterCount.intValue();
13   do {
14    correctCentroids = globalCentroids.getCorrectCoordinates();
15    resetLocalStructures();
16    localDelta = computeClusters();
17    globalDelta.update(localDelta);
18    centroids.update(localCentroids, localSizes);
19    barrier.await();
20    globalIterCount.compareAndSet(iterCount, iterCount++);
21   } while (iterCount < maxIterations && !endCondition());
22  }
23 }
```

Listing 2. *k*-means implementation with Crucial.

6.2.1 A serverless *k*-means

Listing 2 details the code of a *k*-means clustering algorithm written with Crucial. This program computes *k* clusters from a set of points across a fixed number of iterations, or until some convergence criterion is met (line 21). The algorithm is iterative, with recurring synchronization points (line 19), and it uses a small mutable shared state. Listing 2 relies on shared objects for the convergence criterion (line 3), the centroids (line 8), and a synchronization object to coordinate the iterations (line 19). At each iteration, the algorithm needs to update both the centroids and the criterion. The corresponding method calls (at lines 14, 17 and 18) are executed remotely in the DSO layer.

Figure 3 compares the scalability of Crucial against two EC2 instances: `m5.2xlarge`, `m5.4xlarge`, with 8 and 16 cores respectively. In this experiment, the input increases proportionally to the number of threads. We measure the *scale-up* computed with respect to that fact: $scale\text{-}up = T_1/T_n$, where T_1 is the execution time of Listing 2 with one thread, and

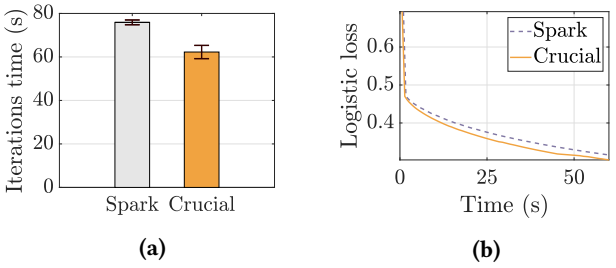

Figure 4. Comparison of Spark and Crucial implementations of Logistic Regression. (a) shows the average completion time of the iteration phase (100 iterations). (b) shows a comparison of the performance of both systems.

T_n when using n threads.[6] Accordingly, $scale\text{-}up = 1$ means a perfect linear scale-up, i.e., the increase in the number of threads keeps up with the increase in the workload size (top line in Figure 3). The scale-up is sublinear when $scale\text{-}up < 1$. Non-surprisingly, the single machine solution quickly degrades when the number of threads exceeds the number of cores. The solution using Crucial is within 10% of the optimum. For instance, with 160 cloud threads, the scale-up factor is ≈ 0.94. This lowers down to 0.9 for 320 threads due to the overhead of thread creation.

6.2.2 Comparison with Spark

We compare Crucial against Spark [55] using two machine learning algorithms: logistic regression and *k*-means. Both algorithms are iterative and share a modest amount of state that requires per-iteration updates. So they are a perfect fit to assess the efficiency of fine-grained updates in Crucial against a current state-of-the-art solution. To complement this analysis, we also run the *k*-means application with a modified version of Crucial that uses Redis for in-memory storage. Object methods are implemented in Redis with the help of Lua scripts.

Setup For this comparison, we provide equivalent CPU resources to all competitors. In detail, Crucial experiments are run with 80 concurrent AWS Lambda functions and one storage node. Each AWS Lambda function has 1792MB and 2048MB of memory for logistic regression and *k*-means, respectively. These values are chosen to have the optimal performance at the lowest cost (see Section 6.2.3).[7] Spark experiments are run in an Amazon EMR cluster with 1 master node and 10 `m5.2xlarge` worker nodes (*Core nodes* in EMR's terminology), each having 8 cores. The Spark executors are configured to utilize the maximum resources possible on

[6] In Figure 3, threads are AWS Lambda functions for Crucial, and standard Java threads for the EC2 instances.

[7] Starting with a configuration of 1792MB, an AWS Lambda function has the equivalent to 1 full vCPU (https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html). Also, with this assigned memory, the function uses a full Elastic Network Interface (ENI) in the VPC.

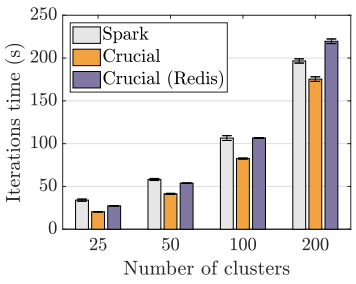

Figure 5. Average completion time of the iteration phase (10 iterations) of the *k*-means algorithm with varying number of clusters.

each node of the cluster. The DSO layer and Redis run on a `r5.2xlarge` EC2 instance.

Dataset The input is a 100GB dataset generated with spark-perf [11] that contains 55.6M elements. For the logistic regression use case, each element is labeled and contains 100 numeric features. For *k*-means, each element corresponds to a 100-dimensional point. The dataset has been split into 80 equal-size partitions to ensure that all partitions are small enough to fit into the function's memory. Each partition has been stored as an independent file in Amazon S3.

Logistic regression We evaluate a Crucial implementation of logistic regression against its equivalent counterpart in Spark's MLlib [37]: `LogisticRegressionWithSGD`. A key difference between the two implementations is the management of the shared state. At each iteration, Spark broadcasts the current weight coefficients, computes, and finally aggregates the sub-gradients in a MapReduce phase. In Crucial, the weight coefficients are shared objects. At each iteration, a cloud thread retrieves the current weights, computes the sub-gradients, updates the shared objects, and synchronizes with the other threads. Once all the partial results are uploaded to the DSO layer, the weights are recomputed and the threads proceed to the next iteration.

In Figure 4, we measure the running time of 100 iterations of the algorithm and the logistic loss after each iteration. Results show that the iterative phase is 18% faster in Crucial (62.3s) than Spark (75.9s), and thus the algorithm converges faster.[8] This gain is explained by the fact that Crucial aggregates and combines sub-gradients in the dedicated shared object layer. On the contrary, each iteration in Spark induces a reduce phase that is costly both in terms of communication and synchronization.

k-means We now compare the implementation of *k*-means described in Section 6.2.1 to the one in Spark's MLlib. For both algorithms, centroids are initially at random positions.

[8]This includes neither the provisioning time of the Spark cluster, nor the time to load and parse the dataset from S3 (the same for both systems). FaaS cold starts are also excluded due to a global barrier before measurement.

Table 3. Monetary costs of the experiments

		Total time (s)	Total cost ($)	Iterations cost ($)
k-means ($k = 25$)	Spark	168	0.246	0.050
	Crucial	87	0.244	0.057
k-means ($k = 200$)	Spark	330	0.484	0.288
	Crucial	234	0.657	0.492
Logistic regression	Spark	192	0.282	0.111
	Crucial	122	0.302	0.154

Figure 5 shows the completion time of 10 iterations of the clustering algorithm. In this figure, we consider different values of k to assess the effectiveness of Crucial when the size of the shared state varies. With $k = 25$, Crucial completes the 10 iterations 40% faster (20.4s) than Spark (34s). The time gap is less noticeable with more clusters because the synchronization portion of each iteration is less representative as the number of clusters increases. That is, the iteration time becomes increasingly dominated by computation. As in the logistic regression experiment, Crucial benefits from computing centroids in the DSO layer, while Spark requires an expensive reduce phase at each iteration. We also see in Figure 5 that the implementation that uses Redis as the storage tier is always slower than Crucial. This aligns with the results of Section 6.1.2.

6.2.3 A note on costs

Although one may argue that the programming simplicity of serverless computing justifies its higher cost [25], running an application serverless should not significantly exceed the cost of running it with other cloud appliances (e.g., VMs).

Table 3 offers a cost comparison of Spark and Crucial based on the above experiments. The first two columns list the time and cost of the entire experiments, including the time of loading and parsing the input data, but without considering provisioning times. The last column lists the costs that can be attributed to the iterative phase of each algorithm. To compare fairly the two approaches, we consider the pricing for on-demand instances and ignore AWS's free tier.

With the current pricing policy of AWS [3], the cost per second of the Crucial setup is always higher than the Spark one: 0.25 and 0.28 cents per second for 1792MB and 2018MB function memory, respectively, against 0.15 cents per second. Thus, when computation dominates the running time, as in the *k*-means clustering with $k = 200$, the cost of using Crucial is logically higher. This difference is erased in experiments during which Crucial is substantially faster than Spark (e.g., for *k*-means clustering with $k = 25$). To give a proper picture of this cost comparison, there are two additional remarks to make here. First, the solution provided

with Crucial using 80 concurrent AWS Lambda functions employs a larger aggregated bandwidth from S3 than the solution with Spark. This reduces the cost difference between the two approaches. Secondly, Crucial users only need to pay for the execution time of their functions, rather than the time the cluster remains active. This includes the bootstrapping of the cluster as well as the necessary trial-and-error processes found, for instance, in machine learning training or hyper-parameter tuning [52].[9]

6.3 Fine-grained synchronization

This section focuses on assessing the capabilities of Crucial to coordinate serverless functions.

6.3.1 Synchronizing a map phase

Many algorithms require synchronization at various stages of computation. In MapReduce, synchronization happens during the shuffle between the map and reduce phases. Starting the reduce phase requires to wait that all the appropriate data is output in the map phase. This is a costly operation, even if the reduce phase is short.

When data is small and the reduction operation simple, aggregating the output of the map phase directly in the storage layer is faster [12]. The DSO layer of Crucial allows to implement such an approach. To attest this fact, we compare different techniques to synchronize at the end of a map phase. *(i)* the original solution in PyWren, based on S3; *(ii)* the same mechanism but above an in-memory key-value data store (Infinispan); *(iii)* using Amazon SQS, as proposed in some recent works (e.g., [30]); and *(iv)* two techniques based on the `Future` object available in Crucial. The first one outputs one object per cloud thread and runs a reduce phase. The second aggregates all the results directly in the DSO layer and simply skips the reduce phase (auto-reduce).

We compare the above five techniques using a simple MapReduce scheme where we run back-to-back the Monte-Carlo simulation in Listing 1. The experiment employs 100 cloud threads, each doing 100M iterations. During a run, we measure the time spent in synchronizing the threads. On average, this time accounts for 23% of the total time spent in a run.

Figure 6 presents the results of this comparison. The solution based on Amazon SQS is the slowest. It employs a polling mechanism that actively reads messages from the remote queue. Using Amazon S3 is also slow, and the approach presents a high variability, some experiments being much more slower than others. This is explained by the combination of high access latency, eventual consistency, and a polling-based mechanism. Infinispan is faster, but being still based on polling, the approach induces noticeable overheads.

[9]Provisioning the 11-machine EMR cluster takes 2 minutes (not billed) and bootstrapping requires an extra 4 minutes. A Crucial storage instance starts in 30 seconds.

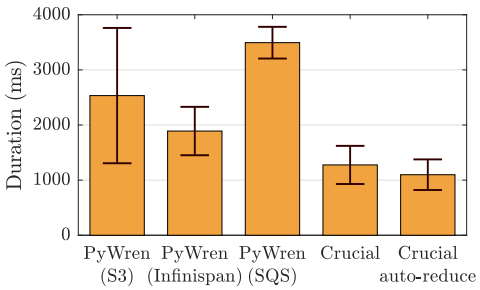

Figure 6. Synchronizing a map phase in MapReduce with PyWren, Amazon SQS and Crucial.

The solution based on `Future` objects allow to immediately respond when the results come up. This reduces the number of connections necessary to fetch the result and thus translates into better synchronization times. When the results of each map phase is aggregated directly in the storage layer, the proposed solution based on Crucial achieves even better performance, being twice faster than the polling-solution using S3.

6.3.2 Synchronization objects

This section evaluates the capabilities of the barrier object available in Crucial. In Figure 7a, we present a comparison against a solution using Amazon SNS and SQS. Figure 7b details a performance breakdown of the proposed synchronization primitive when executing iterative tasks.

In this first experiment, the cloud threads execute consecutive short computations (1s each) in lock step. Figure 7a reports the average time spent in waiting the barrier for a thread. The results show that the barrier primitive of Crucial is scalable: With 320 cloud threads, our solution is one order of magnitude faster than an approach based on the standard toolkit available in AWS. With 1800 cloud threads (not presented in Figure 7a), the barrier is passed after waiting 68ms on average.

Figure 7b further details the performance of the barrier for iterative tasks that need to fetch input from an object store. This figure provides a breakdown of the time spent in each phase of the task (Invocation, S3 read, Compute and Sync) for a selection of 2 cloud threads (out of 10). We report the breakdown for two approaches. The first one launches a new stage of cloud threads (a0 and a1) for each iteration and does not use the barrier primitive. The second approach launches a single stage of cloud threads (b0 and b1) that run all the iterations and use the barrier primitive for synchronization. In the first approach, input data must be fetched from Amazon S3 at each iteration, while in the second approach the threads only need to fetch it once, resulting in a lower total execution time. In addition, Figure 7b shows that the overall time spent in synchronizing cloud threads with a barrier is low. This lower overhead comes from the fact that

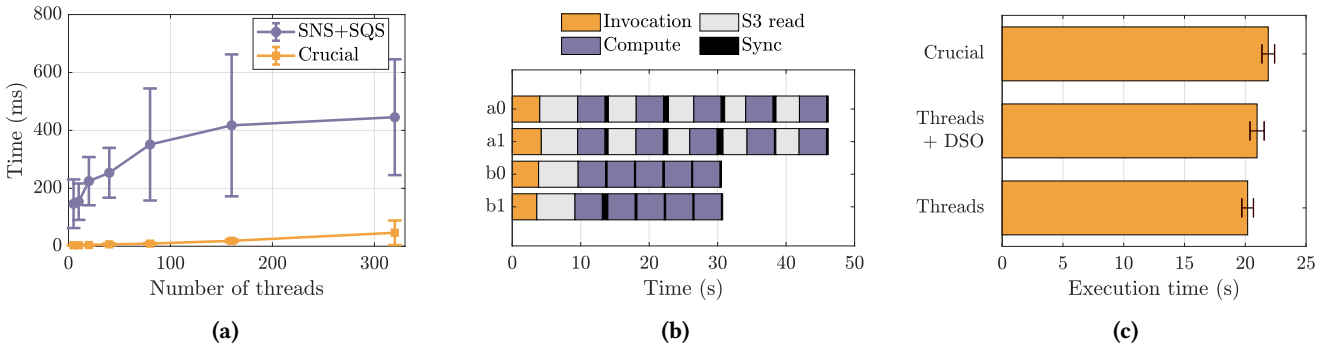

Figure 7. (a) Average time threads spend waiting on a barrier. (b) Performance breakdown of an iterative task using either multiple stages (a0/a1), or a single stage with a Crucial barrier (b0/b1). (c) Execution times of the Santa Claus problem on a single-machine vs. Crucial.

function invocations and S3 accesses are not in the critical path (which induces a high variability as seen in Figure 6).

6.3.3 A concurrency problem

Crucial can also be used for complex task coordination. To demonstrate this feature, we consider the Santa Claus problem [51]. This problem is a concurrent programming exercise in the vein of the dining philosophers, where processes need to coordinate in order to make progress. Common solutions employ semaphores and barriers, while others use actors [6].

In our case, we create a set of synchronization objects to implement the logic of the problem. The entities are cloud threads that communicate through Crucial's shared objects. These objects represent groups and gates, which allow entities to coordinate by joining or passing through them. In essence, the objects act like barriers and semaphores in a distributed way.

We implemented three solutions to the Santa Claus problem. The first solution uses plain old Java objects (POJO), where objects are monitors and the entities are threads. Our second solution is a refinement of this base approach, where shared objects are stored in the DSO layer. The conversion is straightforward using the API presented in Section 3. In particular, the code of the objects used in the POJO solution is not changed. Only the `@Shared` annotation is required. The last refinement consists in using cloud threads instead of Java ones—leveraging the `CloudThread` abstraction.

We consider an instance of the problem with 10 elves, 9 reindeer, and Santa, and run the simulation for 15 *toy deliveries* (epochs of the problem). We take the average time to complete the problem for each solution and plot the results in Figure 7c.

Storing the objects in Crucial induces an overhead of 8%. This low penalty shows that the synchronization model of Crucial is efficient. When cloud threads are used in the last solution, there is almost no difference in the completion time. The difference in Figure 7c is due to the remote calls to the

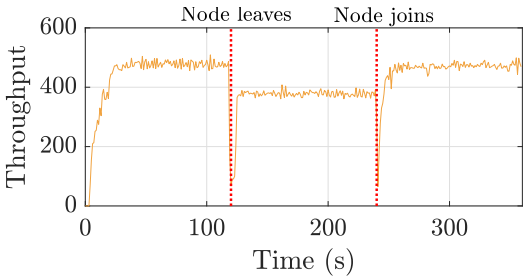

Figure 8. Inferences per second performed on a k-means model during 6 minutes. Up to 100 concurrent FaaS functions connecting to the shared model on up to 3 DSO nodes with $rf = 2$. Note the FaaS cold start at the beginning.

FaaS infrastructure to start the computation (for consistency, we do not include cold starts, which add 1 to 2 seconds of invocation delay). Overall, these results assess that our approach fits well for this kind of application.

6.4 Persistent state

To assess the durability of Crucial's replicated objects, we carry out a base experiment using our k-means code.

Figure 8 shows a 6-minute evolution of the number of completed inferences per second using the k-means model trained with our system. The model remains stored in a cluster of 3 DSO nodes with $rf = 2$. The inferences are performed using 100 cloud threads. Each inference performs a read of all objects in the model (200 centroids) and several distance computations.

In this experiment, at 120s and 240s, we crash and add, respectively, a storage node to the DSO layer. Figure 8 shows that our system is elastic and resilient to storage node failures. Indeed, changes in the membership of the DSO layer affect performance but do not block the system. The (abrupt) removal of a node lowers performance by 30%. The initial

Table 4. Lines of code changed in each application to move it to FaaS with Crucial.

Application	Total lines	Changed lines
Monte Carlo	44	2
Logistic Regression	430	10
k-means	329	8
Santa Claus problem	255	15

throughput of the system (490 inferences per second) is restored 20s after a new storage node is added.

6.5 Programming Simplicity

Table 4 details the modifications that were necessary to port each application to Crucial. The difference between single-machine, parallel code and its serverless counterpart is small. Even for complex programs, such as the logistic regression detailed in Section 6.2.2, changes account for less than 3%. Starting from a conventional object-oriented program, Crucial requires a handful of changes to port it to a FaaS platform. We believe that this smooth transitioning can help everyday programmers to start leveraging the benefits of serverless computing.

7 Related Work

Serverless computing is an emerging paradigm for the cloud. With its simplicity and high scalability, it has seduced both industry [3, 38, 41] and academia [21]. Some recent works in this area focus on enhancing its performance by reducing the cost of isolation [1] and improving startup time [36]. However, existing systems fundamentally lack support for mutable shared state and coordination [20, 26]. In Section 1, we have already discussed them and only a brief recap is provided here.

Coordination Services like Step Functions [4], Durable Functions [38] and Composer [16] orchestrate workflows using state machines. They provide a limited form of coordination among serverless functions and are not designed for highly-parallel concurrent tasks [17]. Coordination kernels such as ZooKeeper [22] can be used to synchronize serverless functions. However, their expressivity is limited and they do not support partial replication [13, 27]. Many serverless frameworks [15, 25, 44, 46] offer BSP-style programming patterns. They mainly differ by the way they synchronize the map operator. While some of them use storage [25, 44, 46], other systems, such as ExCamera [15], implement their own notification systems using a VM-based rendezvous server. Ray [39] is a recent framework to build distributed applications by combining stateless functions and actors that may synchronize with the help of futures. It achieves high-scalability with a bottom-up distributed scheduler and fault-tolerance using a chain-replicated key-value store. Its architecture is based on cluster provisioning and does not fit the serverless model.

Mutable data A number of research works [25, 44, 46] opt to write shared data to slow, highly-scalable storage. To hide latency, they perform coarse-grained accesses. The work of Pu et al. [42] combines Redis with slow storage to scale the shuffling phase in MapReduce. Pocket [31] focuses on the scalability and the cost-efficiency to access ephemeral data for serverless analytics. None of these works meet the requirements for fast, fine-grained updates to shared mutable state necessary in stateful applications.

Mutable shared state can be abstracted in various ways. Crucial targets the simplicity of serverless computing for general stateful applications. It choses to represent state as objects, and keeps the well-understood semantics of linearizability. Existing storage systems such as Memcached [14], Redis [43], or Infinispan [35] cannot readily be used in the DSO layer. They either provide too low-level abstractions or require server-side scripting. We show this problem in Section 6.2.2. Crucial borrows the concept of callable objects from Creson [49]. It simplifies its usage (`@Shared` annotation), provides control over data persistence and offers a broad suite of synchronization primitives. Some systems [47, 48, 50] rely instead on weak consistency, trading ease of programming for performance. The study of other consistency models in Crucial is left for future work.

8 Conclusion

This paper presents Crucial, a system to program highly-concurrent stateful applications on top of a Function-as-a-Service platform. Crucial is built using an efficient disaggregated in-memory data store, and it can be used to construct demanding serverless applications that require fine-grained support for mutable state and synchronization. In particular, we show that Crucial achieves superior or comparable performance to Spark for two common machine learning algorithms. In both cases, less than 3% of our code differs from a conventional solution using plain old Java objects. We also assess experimentally that Crucial rivals in performance with a single-machine, multi-threaded implementation to solve a complex coordination problem, despite the unavoidable overhead imposed by the serverless architectural constraints. Abiding by the simplicity of imperative programming, we believe that Crucial can help broadening the horizon of serverless computing to unexplored domains.

Acknowledgments

This work has been partially supported by the EU Horizon 2020 programme under grant agreement No 825184 and by the Spanish Government through project TIN2016-77836-C2-1-R. Marc Sánchez-Artigas is a Serra Húnter Fellow.

References

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-performance Serverless Computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 923–935. http://dl.acm.org/citation.cfm?id=3277355.3277444

[2] Amazon. 2008. https://aws.amazon.com/s3. retrieved Aug. 2019.

[3] Amazon. 2014. AWS Lambda. https://docs.aws.amazon.com/lambda. retrieved Aug. 2019.

[4] Amazon. 2016. AWS Step Functions. https://aws.amazon.com/step-functions. retrieved Aug. 2019.

[5] Amazon. 2017. AWS Glue. https://aws.amazon.com/glue/. retrieved Aug. 2019.

[6] Mordechai Ben-Ari. 2001. How to Solve the Santa Claus Problem. *Concurrency: Practice and Experience* 10 (2001). https://doi.org/10.1002/(SICI)1096-9128(199805)10:63.0.CO;2-2

[7] Kenneth P. Birman and Thomas A. Joseph. 1987. Reliable Communication in the Presence of Failures. *ACM Transactions on Computers Systems* 5, 1 (Jan. 1987), 47–76. https://doi.org/10.1145/7351.7478

[8] Stephen Blum. 2014. Amazon SNS vs PubNub: Differences for Pub/Sub. https://www.pubnub.com/blog/2014-08-21-amazon-sns-pubnub-differences-pubsub/.

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

[10] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. 2001. Group Communication Specifications: A Comprehensive Study. *ACM Comput. Surv.* 33, 4 (2001), 427–469.

[11] Databricks. 2014. spark-perf. https://github.com/databricks/spark-perf.

[12] David J. DeWitt and Michael Stonebraker. 2008. MapReduce: A major step backwards. DatabaseColumn Blog. http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html.

[13] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. 2015. Extensible Distributed Coordination. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/2741948.2741954

[14] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5–.

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

[16] The Apache Software Foundation. 2017. OpenWhisk Composer. https://github.com/apache/openwhisk-composer. retrieved Aug. 2019.

[17] Pedro García López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. 2018. Comparison of FaaS Orchestration Systems. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 148–153.

[18] Simson L. Garfinkel. 2007. *An Evaluation of Amazon's Grid Computing Services: EC2, S3, and SQS.* Technical Report TR-08-07. Harvard Computer Science Group. http://nrs.harvard.edu/urn-3:HUL.InstRepos:24829568

[19] Red Hat. 2015. Reliable group communication with JGroups. http://jgroups.org/manual/#TOA.

[20] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[21] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with openLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'16)*. USENIX Association, Berkeley, CA, USA, 33–39. http://dl.acm.org/citation.cfm?id=3027041.3027047

[22] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 1.

[23] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2017. Serving deep learning models in a serverless platform. *CoRR* abs/1710.08460 (2017). arXiv:1710.08460 http://arxiv.org/abs/1710.08460

[24] Amos Israeli and Lihu Rappoport. 1994. Disjoint-access-parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing (PODC'94)*. 151–160. https://doi.org/10.1145/197917.198079

[25] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. https://doi.org/10.1145/3127479.3128601

[26] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[27] Babak Kalantari and André Schiper. 2013. *14th International Conference Distributed Computing and Networking*. Springer Berlin Heidelberg, Chapter Addressing the ZooKeeper Synchronization Inefficiency.

[28] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *29th Annual ACM Symposium on Theory of Computing (STOC)*.

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP)*.

[30] Youngbin Kim and Jimmy Lin. 2018. Serverless Data Analytics with Flint. *CoRR* abs/1803.06354 (2018). http://arxiv.org/abs/1803.06354

[31] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[32] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010).

[33] S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (March 1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

[34] Nancy A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[35] Francesco Marchioni and Manik Surtani. 2012. *Infinispan Data Grid Platform.* Packt Publishing Ltd.

[36] G. McGrath, J. Short, S. Ennis, B. Judson, and P. Brenner. 2016. Cloud Event Programming Paradigms: Applications and Analysis. In *2016*

IEEE 9th International Conference on Cloud Computing (CLOUD). 400–406. https://doi.org/10.1109/CLOUD.2016.0060

[37] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7. http://jmlr.org/papers/v17/15-237.html

[38] Microsoft. 2016. Azure Durable Functions. https://functions.azure.com. retrieved Aug. 2019.

[39] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 561–577. http://dl.acm.org/citation.cfm?id=3291168.3291210

[40] Google Cloud Platform. retrieved Aug. 2019. BigQuery. https://cloud.google.com/bigquery/.

[41] Google Cloud Platform. retrieved Aug. 2019. Cloud Functions. https://cloud.google.com/functions/.

[42] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[43] Redis. 2009. https://redis.io/. retrieved Aug. 2019.

[44] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless Data Analytics in the IBM Cloud. In *Proceedings of the 19th International Middleware Conference Industry (Middleware '18)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/3284028.3284029

[45] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.* 22, 4 (1990), 299–319. https://doi.org/10.1145/98163.98167

[46] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. *CoRR* abs/1810.09679 (2018). arXiv:1810.09679 http://arxiv.org/abs/1810.09679

[47] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Convergent and Commutative Replicated Data Types. *Bulletin of the European Association for Theoretical Computer Science (EATCS)* (June 2011).

[48] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Symp. on Operating Systems Principles (SOSP '11)*. New York, NY, USA, 385–400. https://doi.org/10.1145/2043556.2043592

[49] Pierre Sutra, Etienne Riviere, Cristian Cotes, Marc Sánchez-Artigas, Pedro García-López, Emmanuel Bernard, William Burns, and Galder Zamarreno. 2017. CRESON: Callable and Replicated Shared Objects over NoSQL. In *37th IEEE International Conference on Distributed Computing Systems (ICDCS'17)*. https://doi.org/10.1109/ICDCS.2017.239

[50] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. ACM SIGOPS, ACM Press, Copper Mountain, CO, USA, 172–182. http://www.acm.org/pubs/articles/proceedings/ops/224056/p172-terry/p172-terry.pdf.

[51] John A. Trono. 1994. A new exercise in concurrency. *SIGCSE Bulletin* 26, 3 (1994), 8–10. https://doi.org/10.1145/187387.187391

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

[53] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[54] Chenggang Wu. 2019. The State of Serverless Computing. https://www.infoq.com/presentations/state-serverless-computing/ Presentation at QCon New York 2019.

[55] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia