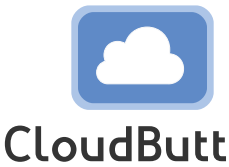


HORIZON 2020 FRAMEWORK PROGRAMME

CloudButton

(grant agreement No 825184)

Serverless Data Analytics Platform

D5.3 CloudButton Toolkit Reference Implementation

Due date of deliverable: 15-12-2021
Actual submission date: 01-07-2022

Start date of project: 01-01-2019 Duration: 36 months

Summary of the document

Document Type	Report
Dissemination level	Public
State	v1.0
Number of pages	62
WP/Task related to this document	WP5 / T5.3
WP/Task responsible	IMP
Leader	Peter Pietzuch (IMP)
Technical Manager	Pedro García (URV)
Quality Manager	Pierre Sutra (IMT)
Author(s)	Simon Shillaker (IMP), Bo Zhao (IMP), Peter Pietzuch (IMP), Carlos Segarra (IMP), Huanzhou Zhu (IMP), Vasily Sartakov (IMP), Guo Li (IMP), Daniel Barcelona (URV), Josep Sampé (URV), Pol Roca (URV), Marc Sánchez-Artigas (URV)
Partner(s) Contributing	IMP, URV
Document ID	CloudButton_D5.3_Public.pdf
Abstract	High-level programming models are essential to realising access transparency and full transparency, thus unlocking the full potential of any serverless cloud platform. However, existing serverless platforms lack flexible programming abstractions, making it challenging for users to build new applications or port existing ones. To address this problem, CloudButton has developed multiple programming models for serverless, which adapt familiar abstractions from big data computing, operating systems and high-performance computing (HPC) to the serverless environment. The developed programming models are: (1) map/reduce and multi-processing with Lithops; (2) multi-threaded serverless programming with CRUCIAL; (3) FaasMP, automatic FaaS-ification of existing OpenMP code for transparency; and (4) FaasMPI, transparent execution of MPI applications on serverless.
Keywords	serverless, function-as-a-service, machine learning, MPI, OpenMP, HPC

History of changes

Version	Date	Author	Summary of changes
0.1	2020-06-18	Daniel Barcelona	Crucial API
0.2	2020-06-22	Simon Shillaker	FAASM: DDOs, FaasMPI and FaasMP
0.3	2020-06-29	Peter Pietzuch	Add more detailed material
0.4	2020-07-02	Simon Shillaker	Summary, abstract and tidying up
0.5	2020-07-02	Peter Pietzuch	Additional clean-up
0.6	2020-07-10	Josep Sampé, Pol Roca	Python APIs and examples
0.7	2020-07-13	Simon Shillaker	Integrating feedback
1.0	2020-07-23	Peter Pietzuch	More review feedback
1.1	2022-05-24	Carlos Segarra, Guo Li	Update section on FaasMPI
1.1	2022-05-27	Bo Zhang	Update introduction and document structure
1.1	2022-05-27	Simon Shillaker	Update background section
1.1	2022-05-30	Huanzhou Zhu, Vasily Sartakov	Updates and apply review feedback
1.1	2022-06-02	Daniel Barcelona	Update Crucial text
2.0	2022-06-20	Carlos Segarra	Updates after in-person discussions
2.0	2022-06-21	Marc Sánchez	Updates Lithops text
2.1	2022-06-30	Gerard Finol, Pedro Garcia Lopez	Updates to global figure, transparent migration of multiprocessing applications.
2.2	2022-07-05	Carlos Segarra	Update introduction
2.2	2022-07-10	Peter Pietzuch	Final version

Table of Contents

Executive summary

Building distributed stateful applications is hard. Users must manage coordination between workers and efficient distribution of data, while scaling underlying system resources. *Serverless computing* provides an easy way to provision and scale resources, but writing applications for this environment remains challenging. This is down to the lack of *high-level programming models*. Existing work on such programming models in big data and machine learning systems is extensive, ranging from RDDs in Spark [1], to tensors in TensorFlow [2] and a plethora of variations on MapReduce [3, 4, 5]. Similar work in serverless is scant, with only a small number of use-case specific approaches that are tightly coupled to the underlying systems [6, 7, 8, 9].

To address this, we present several high-level *serverless programming models* in the context of CloudButton. Not only do we provide powerful, easy-to-use abstractions, but do so without introducing new concepts. Instead we adapt familiar principles such as *multi-threading* and *multiprocessing*, the ubiquitous *MapReduce* paradigm, and the two most popular HPC frameworks, *OpenMP* and *MPI*. This ensures that CloudButton achieves its transparency requirements, as described in D2.3.

In this deliverable, we describe the following work: (i) MapReduce and multiprocessing in Python with Lithops; (ii) serverless multi-threading and shared state in Java with CRUCIAL; (iii) transparent execution of native C/C++, MPI and OpenMP applications cross-compiled to WebAssembly using FAASM.

Changes with respect to previous version of deliverable (D5.2)

This deliverable is an iteration on D5.2, and we summarise the changes with respect to that previous version. In the introduction, we update the descriptions and the status of the three main programming abstractions presented: the Lithops, CRUCIAL, and FAASM. We update the background section to summarise the current status, and shortcomings, of serverless and the state-of-the-art. For each of the three main frameworks covered, the following are the most changes:

1. we expand the description of Lithops, positioning it with respect to the rest of the CloudButton toolkit;
2. we expand the evaluation of the `multiprocessing` API with two use-cases from industry;
3. for CRUCIAL, we update the programming model with the latest extensions and include a detailed methodology for porting existing multi-threaded Java applications to serverless;
4. for FAASM, we introduce the design, implementation, and evaluation of FaasMPI;
5. we re-visit FaasMP by adding new experimental results that explore the performance and scalability in data- and compute-intensive settings;
6. we merge the section on distributed data objects from D5.2 into the FAASM section on D5.3; and
7. we update the conclusions to include the latest contributions, findings, and results.

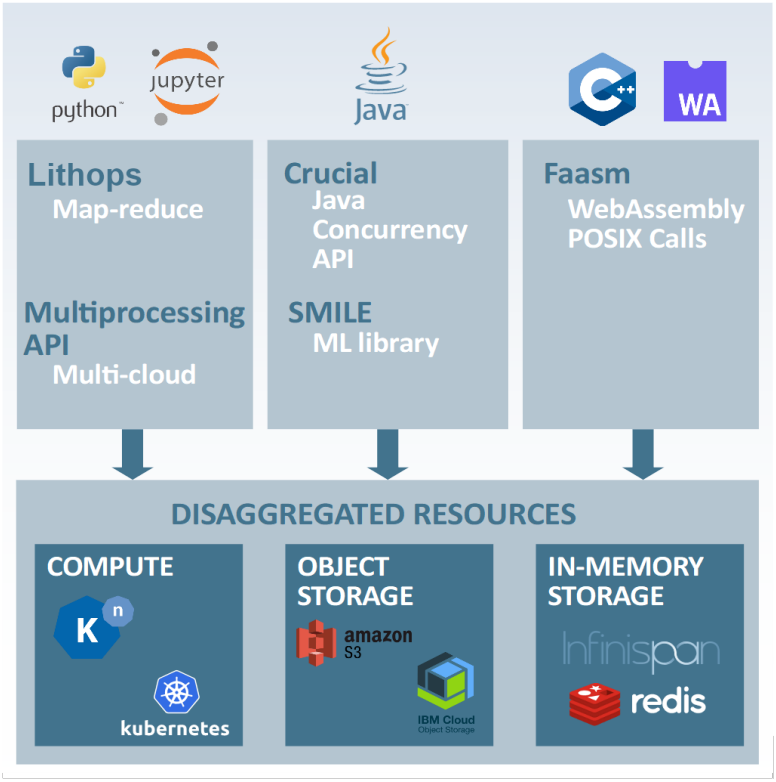


Figure 1: CloudButton toolkit architecture

1 Introduction

Previous research work on serverless computing has placed an emphasis on the underlying runtime environment, focusing on scalability and performance, but little attention has been paid to writing serverless applications. Writing applications remains a challenge, as users must learn new frameworks and concepts, or rewrite existing applications to fit underlying platforms with unfamiliar concepts. While some systems have created use-case specific programming models, none offer a generic approach, or support transparently porting legacy applications. Novel programming models are therefore necessary to achieve the access and full transparency, as described in D2.3.

For the particular case of big data applications, previous work [10] argues that a fully serverless approach may not be desirable. The authors advocate instead for a mix of serverless functions and traditional serverful applications, they call this the *ServerMix* model. As a consequence, novel serverless programming models need to also take into account that parts of the application pipeline might run in a serverful manner, making the design of these programming models even more challenging.

CloudButton address this issue by creating a programming environment that makes it easy for users to develop and scale serverless big data applications. In this deliverable, we report on our findings regarding a range of innovative programming models for serverless applications developed as part of CloudButton.

1.1 Programming models in CloudButton

CloudButton make serverless efficient for big data and easy to use, and hence must provide appropriate programming models. Figure 1 shows the CloudButton toolkit architecture, and highlights the breadth of language support and usage patterns that the project targets through its use cases. CloudButton consists of three serverless frameworks, all of which run on shared disaggregated resources. While these frameworks cater to different languages and use cases, they are united by a common principle: they **build on familiar concepts and abstractions, and thus target full transparency where possible.**

This deliverable describes how we achieve this aim in each of the three frameworks in CloudButton, namely Lithops, CRUCIAL and FAASM. Lithops uses the ubiquitous *MapReduce* paradigm [3] to provide a simple yet powerful programming model that many big data applications already adopt; it also includes a serverless integration with the standard *Python multiprocessing* library to transparently port existing applications; CRUCIAL provides a familiar *multi-threaded programming* model by mapping OS threads to underlying serverless functions using standard Java constructs; FAASM supports the two most popular C/C++ HPC frameworks, *OpenMP and MPI*, by mapping its lightweight thread-based isolation onto serverless functions; it also provides high-level *object-oriented* abstractions in several languages, giving access to simple distributed data structures.

1.2 Serverless programming in context

Existing work on serverless computing has focused on runtime design [11, 12, 13], storage performance [14, 15], state management [16, 17] and application-specific platforms [6, 8]. A comparatively small amount of attention has been paid to creating generic serverless programming models, and almost none has been paid to porting legacy applications. Jangda et al. [18] propose a formal semantics for serverless computing but do not provide a high-level programming model; PLASMA [9] introduces an elastic programming model for serverless, but focuses on actor-based code; Numpywren [7] includes a Python-based programming model, but is limited to linear algebra operations.

The official release of AWS Lambda in early 2015 introduced the idea of using stateless functions as the sole fundamental compute primitive. PyWren [19] demonstrated that the serverless model was general enough to provide the building blocks for elastic and scalable big data systems but that the current platforms suffered from critical performance barriers that needed to be lifted. Related work tuned the fundamental ideas of PyWren to their own applications [20, 21, 22, 23] or tried to provide a new storage layer [24, 25] to circumvent the limitations of the platforms. This lead to the development of stateful efficient serverless solutions [26, 27, 28], which present efficient yet scalable data processing applications.

1.3 Overview of programming abstractions in CloudButton

We describe the three main programming abstractions and implementations for serverless computing that we have developed as part of the CloudButton project in response to our use case requirements:

(1) Lithops. One core principle behind CloudButton is programming simplicity. Our focus is to make serverless computing as usable as possible, irrespective of whether programmers are cloud experts or not. We have devoted efforts to integrate Lithops with other tools (namely, Python notebooks such as Jupyter), which are popular environments from the scientific community. To break vendor lock-in and reach out to a wider community, Lithops is also **multi-cloud**. That is, Lithops applications can transparently be deployed on the most popular cloud platforms, such as IBM Cloud, Google Cloud, Amazon Web Services, Alibaba Cloud, etc., without changing a single line of code. This is thanks to its modular design, which makes it straightforward to integrate the two main types of cloud services leveraged by Lithops:

1. `Compute backends` to launch computing jobs, and
2. `Storage backends` to store all data, including intermediate results.

A compute backend is typically a FaaS platform (e.g., AWS Lambda) while a storage backend is a BaaS[1] storage service (e.g., Amazon S3), so that its two main pillars can scale independently from each other.

To simplify the serverless transition of existing multithreaded codebases, we present two different Python APIs:

- one based on Map-Reduce calls and

[1] BaaS (*Backend-as-a-Service*) is a term that has evolved in the last few years to describe any application-specific serverless cloud service, such as serverless databases.

- another based on standard Python APIs such as `concurrent.futures` and `multiprocessing`.

The `multiprocessing` API includes the implementation of the well-known `Process` and `Pool` computing abstractions, as well as the common Interprocess Communication (IPC) facilities such as the `Queue`, `Pipe`, `Lock`, `Semaphore`, `Event`, and `Barrier`, among others. By default, Lithops implements the IPC components with Redis, a popular in-memory store that offers microsecond-order latency. Although Lithops can provide a solution with only serverless components, we opted for Redis because shared serverless storage (e.g., AWS S3) is two orders of magnitude slower [15].

(2) CRUCIAL. CRUCIAL [15] enables the development of stateful distributed applications in the cloud by simply extending Java's concurrency model. It provides computation abstractions that rely on AWS Lambda to run Java's `Runnable` and `Callable` interfaces based on a basic component: the *cloud thread*. To manage state and task coordination at fine granularity, the system builds a distributed shared object (DSO) layer, with strong consistency guarantees. The application runs on the client's machine but uses disaggregated resources in the cloud to distribute computation and shared state.

The aim is to keep the simplicity of serverless in the programming model. Hence writing code in CRUCIAL is similar to ordinary concurrent Java and distribution is handled transparently. It only requires the user to use simple annotations and constructs to (i) instantiate cloud threads, (ii) annotate shared data, and (iii) use custom synchronisation objects. This simple model also facilitates the porting of existing multi-threaded applications.

The basic cloud thread abstraction lets users run simple `Runnable` tasks seamlessly in the cloud. Internally, CRUCIAL performs the appropriate transformations and connections to run the code in the disaggregated FaaS platform. In addition, the system provides a custom implementation of the `ExecutorService` interface, the `ServerlessExecutorService`, to enable powerful parallel computations directly from traditional Java concurrency code.

Since cloud functions cannot communicate directly, they must communicate through remote shared objects. CRUCIAL builds a distributed shared object (DSO) store to make OOP objects available across hosts. The DSO store uses consistent hashing to efficiently address the shared data and allows to access and update the objects at the level of object methods. This facilitates the development of applications requiring fine-grained state sharing and also enables to implement fine-grained coordination. Data durability is ensured with state machine replication to keep strong consistency.

(3) FAASM. FAASM is a high-performance stateful serverless runtime, which supports C/C++ and the two most popular HPC programming frameworks, OpenMP and MPI. OpenMP and MPI underpin a huge array of existing scientific, big data and machine learning codebases [29, 30, 31, 32]. Through FaasMP and FaasMPI, FAASM supports transparent execution of unmodified OpenMP and MPI code, making it straightforward to port this huge array of existing applications to CloudButton.

FAASM is designed around a new lightweight isolation abstraction called a *Faaslet* [26], which provides security and resource isolation using WebAssembly [33] coupled with existing OS tools such as cgroups and network namespaces. Faasm provides access to distributed state through a two-tier state architecture, which gives co-located functions zero-copy, concurrent access to in-memory state, and synchronises this state across hosts.

For OpenMP programs, FAASM gives *Faaslets* access to a shared global address space, which the runtime transparently synchronises between hosts using FDiffs. Each *FDiff* specifies a modification to the linear shared address space, defined by an offset and an array of modified bytes. *Faaslets* synchronise writes to the address space by building lists of *FDiffs*. Each *Faaslet* maintains a record of writes to shared memory pages, performs byte-wise comparisons of these pages with its parent snapshot, and propagates changes back to a main *Faaslet* via *FDiffs*. To support multiple updates to shared variables across hosts, each *FDiff* specifies a *merge* operation, which can be an arithmetic operation, e.g. summation on a shared variables.

For MPI programs, FAASM provides transparent point-to-point asynchronous message passing for *Faaslets*. *Faaslets* are grouped into *FGroups*, and are assigned a long-lived virtual address. To reduce network overheads, *Faaslets* can be migrated mid-execution to improve locality, and FaasMPI

provides locality-aware collective communication algorithms which it uses to implement most of MPI's API.

FAASM is designed to be a pluggable runtime that integrates with existing serverless platforms. In CloudButton, we use FAASM's Knative [34] integration to execute on the shared disaggregated resource layer shown in Figure 1.

2 Background

FaaS or serverless provides a highly elastic and scalable compute layer for cloud-based applications. Developers write functions in the language of their choice, the provider then provisions and bills the resources on demand, lifting the operational burden from the programmer ("serverless"). It is an advantageous model for cloud providers because they can maximise the utilisation of their resources by co-locating more tenants per machine [35]; the users—provided the development cost is not too high—benefit from fine-grain billing and the absence of operational management costs. Fig. 2 positions serverless with respect to other cloud resources in terms of shared and isolated resources.

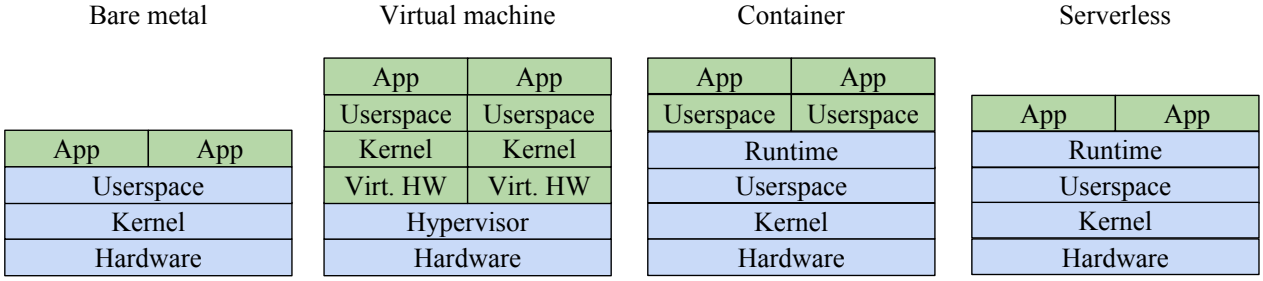


Figure 2: Different levels of isolation for different technologies in cloud computing. (Blue is shared, green is isolated.)

2.1 Serverless today

The term "serverless" was coined by AWS with the release of AWS Lambda [36]. In AWS Lambda, users break applications down into one or more stateless functions, each of which is executed on demand, based on a set of triggers. The triggers for a given function can come from over 200 different AWS services [36], including requests to HTTP endpoints, uploads to shared storage, or messages being enqueued. The platform guarantees to execute one instance of each function in response to its associated triggers, and will attempt to run functions concurrently in response to concurrent triggers. However, there are no guarantees on the level of parallelism provided, nor the delay in executing your function in response to a given trigger [37].

AWS Lambda defined the canonical container-based architecture and programming model that have come to dominate serverless computing today. Lambda continues to be one of the most popular serverless platforms [37], and has driven the development of the Firecracker MicroVM [38]. In the year after Lambda's release, all major cloud providers began offering serverless computing [39, 40, 41], and academic papers on the topic were first published, including OpenLambda [42] and PyWren [6].

The serverless systems from each major cloud provider continue to offer similar services and interfaces as when Lambda was first released, but have also added serverless orchestration frameworks like AWS Lambda Step Functions [43], and Azure Durable Functions [44]. Other work has added a more diverse range of serverless platforms, introducing statefulness [16], application-specific implementations [7], language-specific implementations [45], serverless-specific storage [17], threading [15], and fault-tolerance [46].

Before we propose new programming models for serverless, this section explores the issues with current commercial platforms (§2.2) and their storage layers (§2.3), which limit the ease with which applications can be built on top of them (§2.4).

2.2 Challenges in current serverless platforms

Commercial serverless platforms at the moment can scale shared-nothing task-parallel computation, provided that it is mostly compute-bound [47]. This still fits a variety of self-contained parallel algorithms applied on incoming streams of data, e.g., for event-driven applications in the cloud, data

transformation, Internet-of-Things (IoT) and edge computing. The mainstream technology to run such lightweight tasks are containers because of their ease of provisioning, and the large set of supported applications. Any programming model for serverless must be compatible with existing limitations of serverless platforms:

(1) Data shipping architecture. Existing platforms such as Google Cloud Functions [39], IBM Cloud Functions [41], Azure Functions [40] and AWS Lambda [36] isolate functions such that they cannot share data directly, only via external storage such as object stores like S3 [48].

Having every function share data via external storage introduces data access and serialisation overheads, forcing each function to duplicate shared data, perform repeated serialisation and deserialisation, create and maintain a connection to the same data store, and perform regular network transfers. This model has been labelled the serverless "data-shipping architecture" [49], i.e., moving data to the computation and not vice versa. These overheads are compounded as the number of functions increases, reducing the benefit of unlimited parallelism, which is what makes serverless computing attractive in the first place.

A more efficient way to share data between functions is directly via shared memory, however, sharing memory is fundamentally at odds with the goal of isolation. This makes providing shared access to in-memory state in a multi-tenant serverless environment a challenge. Cloud providers have more recently introduced function orchestration platforms that allow a form of state sharing via persistent artifacts and filesystems between functions [50, 51, 52]. However, these platforms still do not support shared memory between functions.

In §2.3, we provide a detailed analysis of different storage layers for serverless, both in the industry and academia.

(2) Cold start latency. One of the most widely studied problems in serverless today is the "cold start" problem. A cold start occurs when the isolation environment needed to execute a function is created in response to an incoming request. The request experiences a delay as it blocks waiting for the isolation mechanism to boot, and can cause tail latencies in the order of seconds, but most commonly hundreds of milliseconds [37]. An added latency of the order of hundreds of milliseconds can be an order of magnitude higher than the duration of many requests in serverless systems [53].

Container initialisation times have been reduced to mitigate the cold-start problem, which can contribute several seconds of latency with standard containers [54, 49, 37]. SOCK [12] improves the container boot process to achieve cold starts in the low hundreds of milliseconds; Catalyzer [55] and SEUSS [11] demonstrate snapshot and restore in VMs and unikernels to achieve millisecond serverless cold starts. Although such reductions are promising, the resource overhead and restrictions on sharing memory in the underlying mechanisms still remain.

The message to be taken from the work on the cold start problem is that minimising start-up latency and resource overheads in serverless isolation is critical for performance and usability. Existing work shows that VMs and containers are fundamentally ill-suited to the task, and CloudButton introduces alternative lightweight isolation mechanisms.

(3) Limited resources (memory, disk, CPU). Typical consumer cloud platforms offer only resource-limited containers compared to IaaS servers, which often represent lower compute costs for users, even though they actually come with additional operational costs.

(4) Limited programming model. Today's serverless applications achieve parallelism by executing short-lived stateless functions [36, 40, 56]. Stateless functions are useful to providers, as they have no communication or data dependencies, making it straightforward to control the application's parallelism and distribution. This level of control is critical to providers achieving sufficiently high utilisation on shared infrastructure.

Stateless ephemeral functions are effective at executing certain workloads, such as video processing and MapReduce [57, 6], but existing work has also highlighted their shortcomings [58, 59], such as the data shipping architecture and lack of inter-function communication. The majority of existing parallel applications cannot be expressed or executed using stateless ephemeral functions, because:

(i) they achieve parallelism using OS threads and processes, whose semantics are not supported by stateless ephemeral functions; (ii) they use shared memory to share data between threads, but shared memory is not available due to isolated function memory; and (iii) they use message passing to share data, which is not possible as stateless ephemeral functions can neither discover, nor directly communicate with each other.

2.3 Serverless storage layers

FaaS focuses on stateless compute, and as such the question of adequate storage layer has been treated independently. The current FaaS paradigm is not compatible with common cloud-native storage systems. For example, a serverless map/reduce job to sort 100 TB of data can end up costing $23k and take 8 days to complete[2] just the shuffle phase; compared to 50 minutes & $144 for Spark to complete entirely [23]. Indeed, the shuffle phase of the cloud sort requires storing 100 TB of data in an automatically scalable cloud storage service. On AWS Lambda, the only storage option scalable enough for such a data amount is AWS S3, with a guaranteed throughput of at least $3,500$ PUT IOPS [60, 61]. This is largely insufficient to complete the task in a timeline fashion[3]. In this case, Locus [23] suggests the issue can be remediated with a fine-tuned combination of fast and slow storage to efficiently handle the shuffle and reduce parts using an extra merge step after the reduce.

Therefore, current serverless platforms lead to a novel combination of requirements for the storage layer:

1. **Scalability**: automatic, fine-grain, and pay-per-use.
2. **Performance**: high throughput and low latency.
3. **Storage**: any object size, low cost, and ephemeral.

The CAP theorem [62] shows the difficulty of creating a reliable scalable distributed storage system with such performance guarantees, especially with fine-grained scalability. The majority of existing storage services transfer incoming data onto a medium of choice because they are designed for long-term storage. Deletions are not free operations on those platforms, thus the ephemeral storage characteristics does not drive the costs down in the same way that it can for the compute layer, which can efficiently drop or share excess resources. The savings can therefore only come from an aggressive garbage removal policy, either done by the application or the storage layer, which allows the storage layer to reclaim some of its most desired resources such as memory.

Multi-tier storage Many serverless-specific storage services leverage a combination of existing storage technologies under a single API. They aim to provide both a low latency store and a high bandwidth blob store [19]. Examples of such solutions include:

- *Pocket* [24] is an autoscaling storage system that utilises multiple storage technologies with an API that allows serverless applications to rightsize their resource use through hints. It is economical by leveraging advanced flash storage techniques for speeding up remote memory accesses [63] and is capable of DRAM-like throughput but using NVM-e drives for storage which drives down costs by 60%. Relying on pre-fetching and hints however can be problematic without a suitable programming model.
- *Cloudburst* [28] is a serverless platform built on Anna, a distributed KVS, leveraging multiple storage technologies. It monitors frequently used data and uses a consistent caching strategy to replicate it and bring it closer to the computer layer. Inversely, cold data is demoted to slower but cheaper storage in an independently-scalable media fashion.

[2] Assuming 2 GB of memory per lambda (1 GB left for the map/reduce language runtime) & $0.0095 per 1000 PUT+GET on AWS S3 London: 100 TB/2 GB = 50k partitions $\Rightarrow 50,000^2$ files $\times \$0.0000095 = \$23,750$.

[3] 50000^2 files/3,500 PUT per sec $\approx$ 8 days

- *Shredder* [64] is a multi-tenant, yet dependent on tenant cooperation, in-memory store. It uses a kernel bypass mechanism to speed up remote network accesses to avoid requiring specific RDMA-like technologies. It preserves the serverless benefits of logically decoupling compute and storage, however, it co-locates the two when possible. Shredder does not offer as much storage elasticity as Anna and cannot provide fairness guarantees between functions.

To decide on what storage technology to use, cost models allow serverless map/reduce jobs to find an optimal cost/performance balance to allocate expensive but fast storage (e.g. Redis [65]) and rate-limited but cheap storage (e.g. S3 [61]) [23] for their map-reduce operations.

Some SQL-compatible storage platforms refer to themselves as *serverless*, either because they are themselves running on serverless platforms [66], or because they offer scaling to zero and fine-grained billing associated with the underlying DBMS [67]. These are, however, not suitable for use as the high-performance ephemeral storage serverless applications because they are alternative query engines to an underlying storage service.

Infinispan provides a multi-purpose distributed in-memory store that can be used to implement distributed state for serverless functions, as demonstrated in CRUCIAL.

2.4 Serverless data analytics

The issues of current serverless platforms were identified by frameworks that seek to utilise the serverless promise of a virtually infinitely scalable compute layer. Big data applications such as PyWren use stateless functions to compute distributed operations, including map/reduce [19, 20]. After initially struggling with network efficiency, more recent work such as *numpywren* [21], which focuses on linear algebra, manages to partially overcome these issues by pipelining data. This allows numpywren to simultaneously pre-fetch and save data while executing computation. The orchestration of the functions is, however, challenging because a pipelining mechanism requires to reuse functions but their lifetime is usually limited by the platform. Therefore such solutions are not applicable to many applications.

It is common for serverless applications to require additional stateful components to handle some specialised coordination mechanisms [68, 23, 22]. This approach may be manageable on a small scale but is ultimately an issue that limits scalability and usability of systems, and often represents a critical point of failure. Some approaches focus on the custom provisioning of resources for specific applications such as statistical machine learning [69], while others provide more general frameworks to coordinate serverless machine learning (ML) [68, 70]. Often not backward compatible, they may require to rewrite all the existing ML stack to make use of their features, and they do not provide a storage layer as elastic as the compute layer. More recent approaches focus on the more fine-grained requirements of reinforcement learning (RL) [71]. They can even scale RL algorithms in a fault-tolerant manner thanks to the use of a global state disassociated from stateless functions.

Finally, the HPC community has also implemented applications to run on serverless platforms [72, 73]. These approaches are on the fringes of typical HPC applications by not using the tools commonly used by the community (e.g. RDMA, OpenMP, and MPI) and are limited to trivially scalable tasks. Supporting them is one of the main drivers in the CloudButton project.

Listing 2: Lithops example using the Map-Reduce API

```
from Cloudbutton.engine import function_executor

def my_map_function(x, y):
    return x + y

if __name__ == "__main__":
    args = [ # Init list of parameters
           (1, 2), # Args for function1
           (3, 4), # Args for function2
           (5, 6), # Args for function3
          ]  # End list of parameters

    exc = function_executor()
    exc.map(my_map_function, args)
    print(exc.get_result())
```

3 Lithops: Python APIs

Lithops exposes different APIs that can be used based on user requirements. In D5.2, we presented a first API definition based on map-reduce. Now, the flexibility of Lithops is substantially increased by mimicking the Python's `multiprocessing` API and components.

3.1 Map-Reduce API

The Map-Reduce API is the basic API used by Lithops, and it integrates the basic, low-level methods to spawn functions in the cloud. The primary object in the Map-Reduce API of Lithops is *executor*. This object allows to perform calls to the Lithops API to run parallel tasks. The standard way to get everything set up is to import the module `lithops`, and call the class `FunctionExecutor()` to get an instance of the executor:

Listing 1: Lithops example instantiating the `FunctionExecutor`

```
import lithops
lth = lithops.FunctionExecutor()
```

When an instance of the executor is created, a unique ID is assigned to the instance. This unique ID is used later to keep track of function invocations and the results stored in the storage backend. The executor loads the configuration (e.g., account details) required to grant Lithops access to the compute and storage backends necessary to launch Lithops. Once you get an instance of the executor, you can spawn functions with the next API methods:

▷ `call_async()`: The first proposed method is used to run asynchronously just one function in the Cloud. This method is non-blocking, i.e., the sequential execution of the local code continues without waiting for the results. The parameters of this method are the `function_code` and the input `data` that the function executor receives.

▷ `map()`: The second proposed method is called `map()`. This method is used to run multiple function executors. This method is also non-blocking and takes as main input the `map_function_code` and the `data` that the map function executors receive. Unlike the prior method, this one receives as input `data` a list the number of parallel functions to spawn, alongside with the input parameters that should be sent to the functions.

▷ `map_reduce()`: The third proposed method is used to execute MapReduce flows, i.e., multiple map function executors (map phase), and one or multiple reduce function executors (reduce phase). This method is also be non-blocking. It takes as input the `map_function_code`, the input `data` as a list of

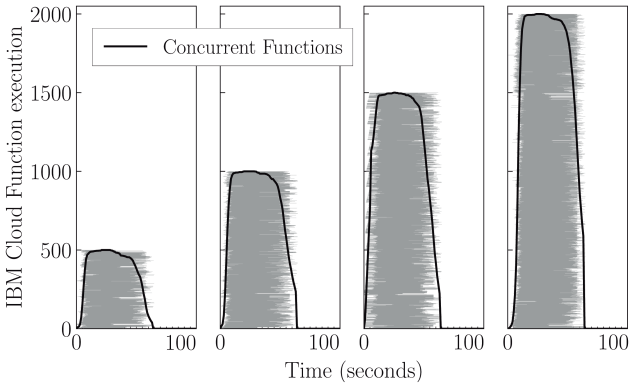


Figure 3: Elasticity and Concurrency. Black lines show total concurrent functions. Each horizontal gray line represents a function execution.

values, and the `reduce_function_code`. As in the prior method, it can spawn the desired number of mappers and reducers.

▷ `wait()`: On the client side, the `FunctionExecutor` offers a method to monitor the executions. This method is called `wait()`. It is synchronous, i.e., the local user code is blocked until the call to `wait()` ends. It provides a configurable parameter to decide when to release the call and continue the execution. Moreover, a user can decide to unlock the method in three different circumstances: (1) '`Always`': it checks whether or not some result is available on the invocation of `wait()`. If so, it returns them. Otherwise, it resumes the local execution; (2) '`Any completed`': it resumes the local execution upon termination of any function invocation; and (3) '`All completed`': it waits until all the functions have finished they execution and the results are available. In these three cases, the `wait()` method returns a 2-tuple of lists: the first list with the futures that completed and the second with the uncompleted ones.

▷ `get_result()`: This method is used to collect the results from the functions when a parallel task has finished (e.g., `map()`, `map_reduce()`, etc.). It adds some functionality such as timeout support, keyboard interruption to cancel the retrieval of results, and a progress bar to inform users about the % of task completion. Last but not least, this method is *composition-aware*: it transparently waits for an on-going function composition to complete, just returning the final result to users.

▷ `plot()`. This method is useful to display the execution trace of a parallel workflow. It outputs two plots onto the destination folder `dst` of the user's local filesystem. The first plot is a timeline diagram, which, among other things, records the invocation, activation and completion times of each function executor. The second plot is a horizontal histogram which is very convenient to quantify the degree of concurrency of the different function executors. An example of this plot is shown in Figure 3 for IBM Cloud Functions.

▷ `clean()`. This method allows to clean all the temporary data produced by Lithops once a job has ended. Usually, it is automatically called after an invocation to `get_result()`.

3.2 Storage API

To chain MapReduce jobs without forcing the client machine to download big intermediate state from the cloud, Lithops provides the Storage API. This API makes it straightforward to operate the storage backend with calls similar to those of the Python *boto3* library. One can use the Storage API to upload a file from our computer to a cloud storage service, and then read this file from a function spawned with the `call_async()` method:

Listing 3: Example using the Lithops storage API

```
from lithops import FunctionExecutor, Storage
```

```
BUCKET, KEY = 'my-bucket', 'test.txt'

def get_file(key, storage):
    return storage.get_object(bucket=BUCKET, key=KEY))

storage = Storage()
storage.put_object(bucket=BUCKET, key=KEY, body='Hi!')

with FunctionExecutor() as fexec:
    fut = fexec.call_async(get_file, KEY)
    print(fut.result())
```

3.3 Python multiprocessing API

We have extended Lithops with a `multiprocessing` module which fully implements the original Python `multiprocessing` interface. Abstractions for parallel computing (like `Process` and `Pool`) use Lithops *FunctionExecutor* API, while inter-process shared memory communication and synchronization abstractions, like `Lock`, `Pipe`, `Queue` or `Manager`, leverage Redis key-value in-memory database as an intermediary. Table 1 presents a list of the available abstractions, along with their category and description.

Table 1: Python multiprocessing API components

CATEGORY	ABSTRACTION	DESCRIPTION
Computation	`Process, Pool`	*Abstractions for launching parallel processes.*
Shared state	`Manager, BaseManager, Manager.dict, Manager.list`	*High-level shared state abstractions, basic types (list or dict) or shared complex objects (Manager, BaseManager).*
Connection	`Pipe`	*Single-producer, single-consumer bi-directional communication between two processes.*
Queues	`Queue, SimpleQueue, JoinableQueue`	*Ordered multiple-producer, multiple-consumer queues.*
Shared memory	`Value, Array`	*Shared array for basic C-types (int, float...).*
Synchronization	`Lock, RLock, Semaphore, BoundedSemaphore, Condition, Event, Barrier`	*Objects for distributed worker-to-worker coordination and synchronization.*

Lithops multiprocessing enables to transparently port local-parallel applications written in Python multiprocessing to a distributed Cloud and serverless environment without modifying the applica-

Listing 4: Example using the Lithops multiprocessing API

```
# import multiprocessing as mp
import lithops.multiprocessing as mp
import random

def pi_montecarlo(n):
    count = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if x*x + y*y < 1:
            count += 1
    return count

num_processes = 1000 # parallelism with 1000 processes is only possible with serverless!
num_points = 10000000
part_count = [int(num_points/num_processes)] * num_processes
pool = mp.Pool(processes=num_processes)
count = pool.map(pi_montecarlo, part_count)
pi = sum(count) / num_points * 4
print(f"Esitmated Pi: {pi}")
```

tion code or architecture. Providing transparency is a big deal for application programmers since a user who is not familiar with distributed computing and Cloud computing can benefit from flexible resources and scale and adapt to heavier workloads using familiar Python parallel computing abstractions. Using serverless functions to scale local-parallel applications brings several benefits. The main is productivity, since the user does not have to worry about provisioning and managing resources since the Cloud provider is managing them on their behalf. On the other hand, we can massively and instantly scale applications that would otherwise require costly cluster synchronization and management operations using virtual machines. However, making use of serverless functions entails significant overheads caused in part by the need for indirect communication used for stateful operations. In exchange, we can scale an application well beyond the physical limits of a virtual machine.

Listing 4 represents a simple example of the basic usage of the Lithops multiprocessing library. We can see that the code is valid for both local multiprocessing module and Lithops multiprocessing module. We can therefore replace the `import` statement of the local multiprocessing library with `lithops.multiprocessing` to simply use remote processes on serverless functions instead of local processes.

Serverless functions cannot use direct communication since one of *FaaS* fundamental basis is function isolation, i.e., a function has no knowledge of other functions that are being executed in its environment. For this matter, Lithops multiprocessing shared components across processes are transparently supported by using Redis database as an intermediary. We have chosen Redis for its simplicity of deployment, in-memory storage and high performance. Redis differs from other traditional key-value databases because values have a type, such as `LIST`, `STRING` or `HASHSET`. A distributed deployment of a Redis cluster can scale horizontally and fault tolerance is guaranteed thanks to replication.

We have also incorporated to Lithops a replica of Python's built-in `open` function and the `os.path` module which allows to transparently read and write files and directories stored on object storage service (like S3) as if it were a local file system. This is especially useful for *FaaS* since the volume that is mounted in the function container is volatile and the data stored there is lost when the execution finishes. In this sense, we offer serverless processes a transparent way to save or recover their state.

Python uses processes to achieve parallelism and to overcome the Global Interpreter Lock (GIL),

Listing 5: Importing of equivalent API's

```
# Modules used for local execution
# import os
# import multiprocessing as mp

# Modules used for remote execution
from lithops.Cloud_proxy import os, open
import lithops.multiprocessing as mp

# Other modules
import requests
```

Listing 6: Loading of model to the storage

```
WEIGHTS_URL = 'http://moments.csail.mit.edu/moments_models'
WEIGHTS_FILE = 'moments_RGB_resnet50_imagenetpretrained.pth.tar'

response = requests.get(WEIGHTS_URL + '/' + WEIGHTS_FILE)

with open(WEIGHTS_FILE, 'wb') as weights_file: # Transparent access to storage
    weights_file.write(response.content)
```

which prevents threads from executing in parallel on multi-core machines. With Lithops, launching processes has considerable overhead. If the granularity of the tasks is very small, the overhead of invoking many functions can be prohibitively expensive. To overcome this problem, we have implemented the job queue pattern for the Lithops multiprocessing `Pool`. In Lithops multiprocessing, workers are long-lived functions that are invoked when the `Pool` object is created. Operations on the pool (`map()`, `apply_async()...`) enqueue Lithops tasks to a Redis list that are then picked up and executed by multiple workers as they are generated. Once all jobs are finished and the worker pool is closed, a message is sent to the workers to terminate their execution. The main advantage of this implementation is that the overhead of submitting a set of tasks to a Redis list is much lower than invoking a function for every task. Also, reusing functions to execute multiple tasks avoids stragglers caused by cold invocations.

3.3.1 Application example: Deep learning video inference

This example uses Lithops multiprocessing to process videos from the Moments-in-Time video dataset [74]. It predicts actions that appear in videos using a pretrained ResNet50 deep neural network model. This code has been implemented first using local multiprocessing for development and debugging purposes. In order to scale and processes a large amount of videos in parallel, it has been ported to Lithops multiprocessing with minor modifications, so that processes are executed by serverless functions that load data from Cloud storage. The storage is used to load input video files and the serialized model for inference.

In Listing 5, we can see how the user can decide whether to use local or remote resources. By replacing the local module import statements and importing Lithops module instead, enables to access to either local files or remote files stored in the Cloud using the same sort of methods. After this change, calls to the `os` module or `open` function will either happen in the local filesystem or in the Cloud storage filesystem transparently to the user. In the same way, using Lithops `mp.Pool` will invoke remote processes instead of local processes, while keeping the same API.

First, the model is downloaded from the respective repository and then transparently stored in Cloud storage using the built-in `open` function. This step is necessary since the model has to be placed in the storage for every function to be able to access it.

Listing 7: Video prediction function

```
# Remote process function
def predict_videos(queue, video_locations):
    with open(weights_location, 'rb') as f:
        model = load_model(f)

    model.eval()
    results = []

    for video_loc in video_locations:
        with open(video_loc, 'rb') as video_file:
            frames = extract_frames(video_file, NUM_SEGMENTS)
            input_v = torch.stack([transform(frame) for frame in frames])

        with torch.no_grad():
            logits = model(input_v)
            h_x = F.softmax(logits, 1).mean(dim=0)
            probs, idx = h_x.sort(0, True)

        result = {
            'video_id': video_loc,
            'prediction': (idx[0], round(float(probs[0]), 5))
        }
        results.append(result)

    queue.put(results)
```

The video prediction function in Listing 7 shows a simple procedure that first loads the model and then loads the input videos to make predictions one by one. After the inference of all videos is completed, results are put in a queue that the main process reduces on the go. The reduce operation (Listing 8) operation processes results from the queue at the time they are completed and sent to the queue, and it creates a record with the total amount of predictions made for each category.

Finally, Listing 9 contains the main function code. First, the list of paths or keys of the input videos is obtained. The list is split among `N` parts matching the desired concurrency, and thus, each function may end up processing multiple videos. Then, `Pool.map_async` function is called to spawn multiple remote processes. It spawns as many as there are elements in the list of input data `iterdata`. After that, and since the last call was asynchronous, the main process starts performing the reduce operation with the queue allowing it to process results immediately without having to wait for all of them to complete.

As we can see, model inference is a good example of a process that can be embarrassingly parallelized thanks to Lithops multiprocessing, because there are no dependencies or communication between functions and the use of serverless functions allows for massive scaling. We have been able to see how using the same operations for local parallel programming we can scale the application to process a large amount of videos stored in Cloud storage using thousands of processes without changing the application logic nor code.

3.4 Experimental results: Map-Reduce API

In this section, we evaluate the performance of the Map-Reduce API through a real use case based on `www.airbnb.com` (Airbnb).

3.4.1 Evaluation setting

For the experiments with the Map-Reduce API, we have used the IBM Cloud services in the `us-east` region — i.e., Washington DC. As a baseline for our experiments, we have used a laptop with the

Listing 8: Reduce function

```
# Local process function
def reduce_predictions(results_queue, n_results):
    category_count = {}
    for categ in categories:
        category_count[categ] = 0

    for i in range(n_results):
        results = queue.get()
        for res in results:
            idx, prob = res['prediction']
            category = categories[idx]
            category_count[category] = category_count[category] + 1

    return category_count
```

Listing 9: Main function

```
CONCURRENCY = 1000

# Main function
def main():
    queue = mp.Queue()
    pool = mp.Pool()

    video_locations = [os.path.join(INPUT_DATA_DIR, name) for name in
        os.listdir(INPUT_DATA_DIR)]
    N = min(CONCURRENCY, len(video_locations))
    iterdata = [(queue, video_locations[n::CONCURRENCY]) for n in range(N)]

    pool.map_async(func=predict_videos, iterable=iterdata)
    result = reduce_predictions(queue, N)
    print(result)

if __name__ == '__main__':
    main()
```

following specs: Intel Core i5 (4 cores) with 16GB RAM and Ubuntu 20.04.

It is worth noting that the focus of Lithops is to simplify the parallel execution of everyday tasks in the cloud, and not to compete with complex computing stacks running on warm clusters. In this sense, it is more interesting to assess the performance benefits for non-cloud users, who typically run programs at "laptop scale". Or to put it baldly, the benefits from shifting from the laptop to the cloud.

3.4.2 Sentiment Analysis

As a real example of the Map-Reduce API, we have crafted a use case example to demonstrate how Lithops can help to process datasets stored in IBM COS. For this example, we have used Airbnb data from various cities around the world, in conjunction with a tone analyzer to uncover emotional and language tones in written text. To make it more appealing, we have plotted the results visually on a city map.

Datasets. The data was retrieved from IBM Watson Studio Community [75] and then copied to an IBM COS bucket. In particular, there is a dataset per city, which contains all the apartment reviews written by the users. As some cities are more "touristy" than others, the dataset size varies from city to city. The full dataset is made of 33 cities, with a total size of 1.9GB and 3,695,107 comments.

Experiment. We first examine how much time the experiment takes without the concurrency of Lithops. We built a Jupyter notebook in IBM Watson Studio to process sequentially all the cities. For the hardware configuration of the VM, we borrowed the same specs of our laptop: 4vCPU with 16GB of RAM. With this setup, it took 1 hour and 26 minutes to serially process all the 3,695,107 comments and render the 33 city maps, which is a significant time burden, especially for impatient users.

Next, we redid the same experiment but with the aid of Lithops. Essentially, we performed some cosmetic changes to the code to execute the experiment via the `map_reduce()` call. On the one hand, we created another Jupyter notebook in IBM Watson Studio with the same hardware configuration as in the prior test. On the other hand, we set up the Lithops IBM Cloud Functions runtime to use 1 GB of RAM.

We remember that it is possible to call `map_reduce()` with a specific chunk size. The chunk size determines the final concurrency, so we played out with different chunk sizes to understand how it affects the total execution time. Further, we set `reducer_one_per_object=True` to have a dedicated reducer per city dataset. That is, each reducer collected the partial results from its corresponding city and rendered the final map. An example of a map is depicted in Figure 4. In this case, it represents the tone analysis of the comments of the City of New York. Each point in the map represents the location of the apartment, and the color of the point signals the tone of the comments.

Table 2: Performance of tone analysis of Airbnb reviews for different chunk sizes. The results are better than in our preliminary work [76] as they have been expressly re-run for this deliverable.

Chunk size (MB)	**Number of executors**	**Execution time** (sec)	**Speedup**
——-	——-	5160	Baseline
64	47	313.87	16.43x
32	72	196.24	26.29x
16	129	95.48	54.04x
8	242	59.77	86.33x
4	471	35.01	147.38x
2	923	25.58	201.72x

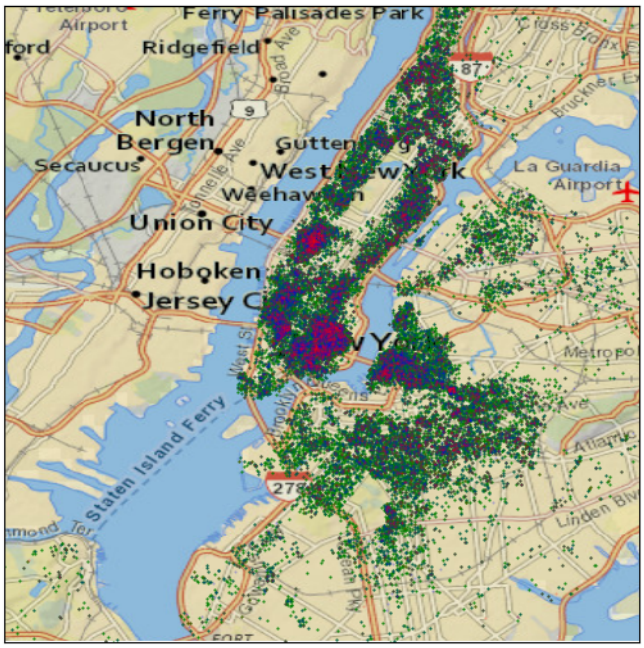


Figure 4: Tone analysis of the Airbnb reviews of New York City. Green, blue and red points stand for good, neutral, and bad comments, resp.

Results. In Table 2, we report the results of this experiment. A first key observation to be made is that Lithops achieved excellent speedups, greater than 100x. This proves the huge benefit of Lithops to non-cloud users, who can readily leverage the large number of CPU cycles available in the cloud, with no need to struggle with hardware management and specialized stacks such as Spark and MPI. In practice, although Lithops exhibits some overhead, users would not care as much about parallel efficiency, but more about the savings in compute times with close to zero devops cost.

Notice that the number of function executors does not duplicate when halving the chunk size. This occurs because partitioning takes place within each dataset file. Either way, the achieved parallel execution time is proportional to the number of function executors, growing between 16.43x and 201.72x for chunks of 64MB and 2MB, respectively.

3.5 Transparent migration of Python multiprocessing applications

In this section, we evaluate the behavior of Lithops in two real use cases in order to test full access transparency and to measure performance. The scenarios used are: the POET modifications in Evolution Strategies made by the Uber research team, and the implementation of the OpenAI's Proximal Policy Optimization (PPO) algorithm in its Baselines repository. To adapt these applications to serverless, we only had to replace the `multiprocessing` import with `Lithops.multiprocessing`. Since Lithops fully implements the multiprocessing interface, the rest of the code did not need any further modification.

3.5.1 Evaluation settings

This section describes the configuration with which the experiments described below have been carried out. Experiments have been run with the following settings: Lithops orchestrator runs on a m5.2xlarge EC2 host with Ubuntu 20.04, Lambdas use a containerized Python 3.8 runtime with 1769 MB of RAM [4] as serverless function and Redis 6.2 instance runs on the host machine with Docker. The host machine and the AWS Lambdas are in the same VPC private subnet, region and availability zone (us-east-1 A), so traffic does not go through a NAT gateway nor the public internet. All Lambda functions have been executed using warm containers. All local monolithic executions have

[4] According to AWS documentation [77], a runtime of 1769 MB of memory is assigned a whole vCPU, being a vCPU a thread of a CPU with Hyper-Threading [78].

been carried out using on-demand AWS EC2 instances with different number of vCPUs.

3.5.2 Evolution strategies

In this experiment, we have used the Paired Open-Ended Trailblazer (POET) [79] implementation, which is a large Python application with about 4000 LOC (lines of code) using different *multiprocessing* abstractions like `Pool` or a shared dictionary from a Manager (`Manager.dict()`). This algorithm is part of the Evolution Strategies category, in which, evolutions of an initial population are carried out iteratively, and those evolutions are executed in parallel. The objective of this test is to analyze and compare the performance and scalability of Lithops in an iterative algorithm that maintains and uses a shared state between processes.

POET uses a shared noise table which is used to generate randomness in the evolution process. This noise table is originally implemented using shared memory. However, it is initialized when the module is loaded, so it is not using Lithops multiprocessing implementation for shared memory. Instead, since this table is read-only, each function can initialize its noise table independently of the shared memory. The algorithm also uses a shared table of parameters that are modified in each iteration. This shared data structure is implemented as a shared multiprocessing `Manager.dict()` dictionary. Therefore, there is a certain transmission of data that could imply a significant overhead.

The `multiprocessing` abstractions used in POET are: one `Context` set to *spawn* mode, one `Pool` for tasks executions, one `Manager` with two `Dict` that contain the shared stated used by all the worker processes from the Pool.

Each iteration of the algorithm performs a `Pool.map()` operation. As the task granularity is small (about 3 seconds), to try to mitigate overheads, we used the optimization of the `Pool` with job queue explained before. To carry out the measurements we have executed 5 iterations with 512 batches per chunk and a batch size of 5. All local executions have been run on a `c5.24xlarge` EC2 instance.

The results in Figure 5 show that, despite the data transmission and invocation overheads, Lithops maintains constant scalability similar to the scalability of the VM. The maximum speedup of the VM is about 40x, while Lithops is capable of reaching a speedup of around 53x, improving the best result of the VM.

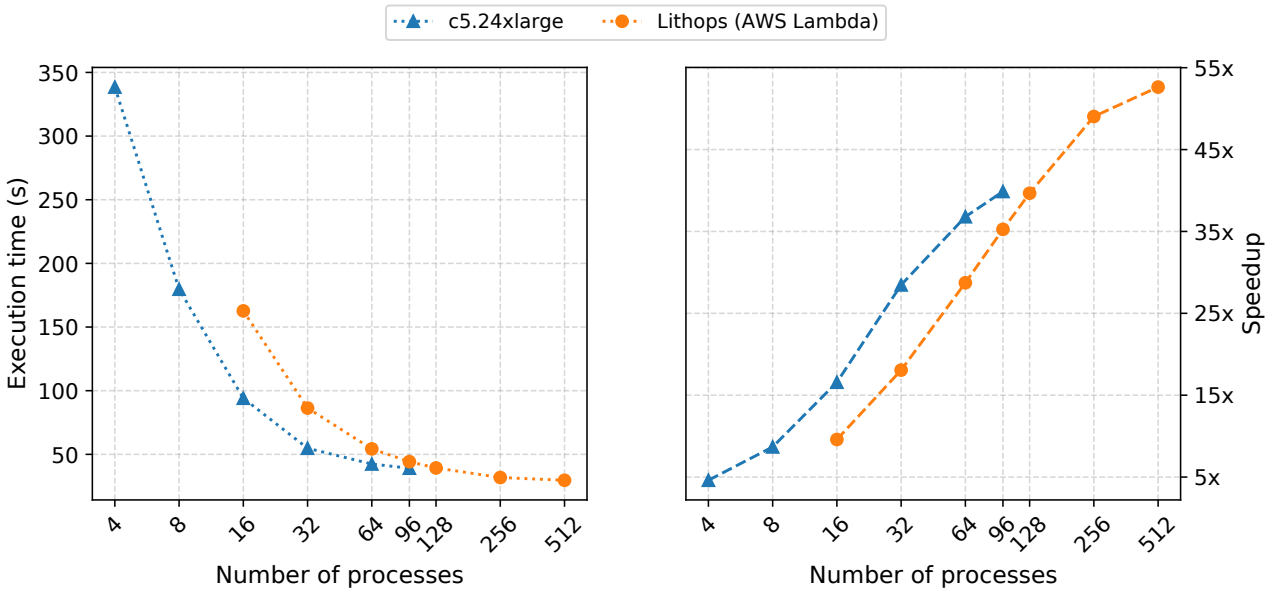


Figure 5: Evolution Strategies execution results.

Proximal Policy Optimization

OpenAI Baselines [80] is a set of high-quality implementations of reinforcement learning algorithms. It has been open-sourced to be used as a base, around which, new ideas can be added and as a tool for comparing new approaches in the reinforcement learning field. At the time of performing the experiments, the Baselines code repository contains more than 16,700 lines of Python code, so it can be considered as a complex Python module. In this experiment, we want to verify that thanks to the access transparency provided by Lithops *multiprocessing* we can simulate the vertical scaling of a virtual machine using FaaS as processes.

We have used the multiprocessing implementation of the Proximal Policy Optimization (PPO) algorithm from OpenAI baselines. The *multiprocessing* PPO version is the second implementation released by OpenAI, and it inherits some of its structure from the first version, which was based on MPI. For that reason, the *multiprocessing* PPO uses a master/worker paradigm relying on `Pipes` for the master to worker communications and vice versa.

The master process is in charge of training the model (a neural network) which, for a given scenario, decides the optimum action to do in order to maximize an objective function. The worker processes are used to simulate the environment in which actions are performed and a reaction is obtained. It is important to notice that each worker process simulates an environment. The training of the model is an iterative procedure where the workers send to the master the actual state of the environment, and it responds with an action to perform in each environment. From the `multiprocessing` API, PPO uses 1 `Context` with the *pawn* mode by default. Associated to that `Context`, it creates 1 `Process` and 1 `Pipe` for each environment emulated. The communication between the master and workers (where states and actions are transmitted) is performed using the `Pipe` associated to each worker `Process`.

In this experiment, we are training a neural network to play the Atari game Breakout, which is available in the OpenAI GYM [81]. Notice that due to the TensorFlow 1 dependency, we have used Python 3.7 in this experiment.

As this algorithm requires the use of a GPU in the master process for the neural network training, the settings for this experiment have been modified. We have used an AWS p3.2xlarge VM as monolithic system, and we tried to scale it vertically using AWS Lambdas. Since the GPU is just used in the master process that runs in the Lithops orchestrator and not in the workers that just do environment simulation, the configuration of the AWS lambdas has not been modified.

The results, available in Figure 6, show that despite the constant communication between processes and the great overhead that this entails, the combination of VM and Lithops achieves a better performance than just the VM. In more detail, the best result for the VM is achieved using 16 processes with a total execution time of 68.92s and the best result of the VM + Lithops is achieved using 64 processes with a total execution time of 61.10s, therefore it reduces an 11% the execution time. This validates that we can emulate a vertical scaling of the VM, and that it is possible to add vCPUs to a VM instantly and without prior provisioning thanks to the use of FaaS.

3.5.3 Insights and lessons learned

After studying the results of the evaluation, we have learned that we are able to transparently move local-parallel applications to distributed settings using serverless. Nevertheless, we have chosen only two representative Python applications that use the multiprocessing library. By this, we do not want to make the claim that all applications can be transparently scaled using serverless without significant degradation. In this section, we would like to discuss the insights obtained from the evaluation that have implications on the feasibility of transparency using serverless services.

Shared state interfaces

Clean shared memory abstractions to communicate processes are very important. Structured and consistent access to shared state requires suitable programming abstractions. In this case, Python multiprocessing design is a clear facilitator for achieving transparency. The ability to perform parallel execution in Python using threads is limited by the GIL, which prevents multiple threads from

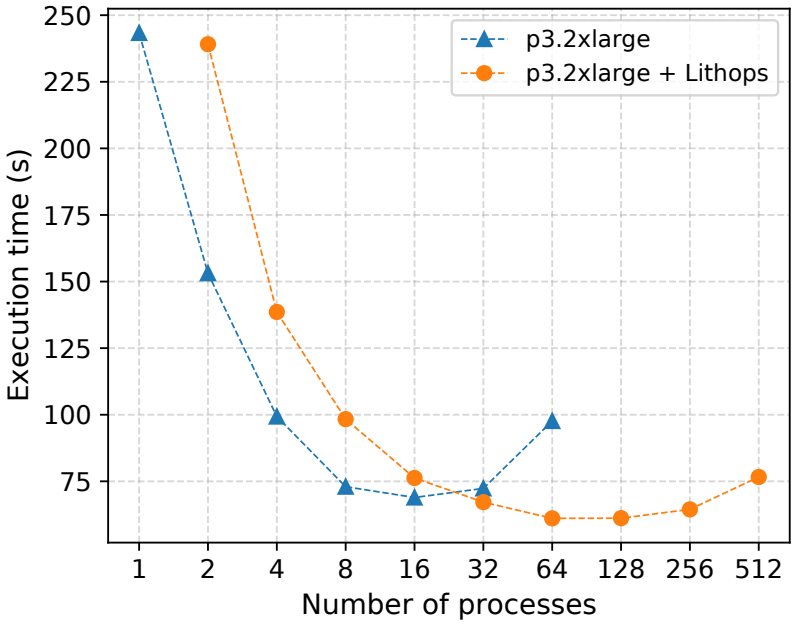


Figure 6: PPO execution results.

running simultaneously on multiprocessor architectures. For this reason, in Python, it is necessary to use processes to have true parallelism. Many of the principles of multiprocessing abstractions, such as Manager, are based on message passing and accessing shared objects (queues, dictionaries, lists...) instead of traditional memory sharing. For example, in a multi-thread application written in Java, two parallel threads can access a shared object by a reference pointer. In contrast, in Python multiprocessing, two processes access to shared state by using messages through a third process (the Manager) that has the shared state. The fact that two Python processes can't share the same address space[5] has facilitated the port of this library to its distributed implementation using disaggregated resources. If the code is not using adequate programming abstractions, full transparency may be impossible.

Latencies and overheads

Overheads are still relevant for many applications. Current Cloud settings still show relevant latency in communications, like hundreds of milliseconds to launch a serverless function, or hundred of microseconds to access in-memory storage services. We have seen that, with equal resources, the overheads generated by creating processes and by the latency of access to shared state are very noticeable. In this line, the granularity of computing tasks is clearly limited by overheads. Very fine-grained computing tasks do not make sense in the current Serverless model, since the overheads can be greater than the task run time.

Performance

Some parallel applications have certain advantages in Cloud Serverless settings that may help to mitigate some of the overheads.

First, hyper-threading may cause performance degradation in virtual machines for compute-intensive tasks using all vCPUs. Hyper-threading makes two threads share some CPU resources like the Arithmetical Logic Unit (ALU). For computationally intensive tasks, two threads are constantly fighting for the shared resources, so the CPU cannot keep up and the execution time is degraded. In HPC, disabling hyper-threading is a common practice to avoid these problems, although the capacity of effective parallelism is reduced by half. For Serverless Functions, AWS Lambda assigns a

[5] except for the multiprocessing.Array abstraction, which can only store basic C types

vCPU (an hyper-threaded CPU thread) per function with a memory configuration of 1769 MB. However, our observations show that the inefficiencies caused by hyper-threading in VMs do not occur in Lambda function executions. This provides an opportunity to further improve the parallelism of high-performance applications that require a full physical CPU for better performance.

Second, accessing large volumes of data in Cloud Object Storage from Serverless functions helps to aggregate bandwidth and accelerate data transfers. A single VM cannot compete with parallel data flows from multiple functions.

In addition, as we have seen in the validation of Section 3.5.2, disaggregated resources can serve as "accelerators" for a VM. That is, when a VM reaches the maximum occupancy of local resources, it could allocate and move computation to disaggregated resources, e.g., to serverless functions. In this way, we could benefit from both fast-access local memory for shared state-dependent processes running on the VM and high flexibility and scalability for stateless processes.

Fault tolerance and serverless services

The fault tolerance of our solution is based on the assumption that the underlying disaggregated resources are fault-tolerant. When programming a monolithic local system, fault tolerance is not taken into account because local resources do not fail. When we move to a distributed environment, if the disaggregated resources (compute, memory and storage) mask the possible failures that may occur, then the application programmer can also assume that they will not fail, and we can continue with the same programming model that does not contemplate error handling and rely on the same local programming model.

Precisely, both AWS Lambda and AWS S3 are fault-tolerant. AWS Lambda can detect and retry failed invocations, while AWS S3 objects are replicated. However, in-memory storage is still not offered as a managed service with scalability and fault tolerance. We are relying on a dedicated Redis service, which must be properly managed now. If the data flows exceed the capacity of this intermediate node, the experiment would fail.

Regarding storage, we are now intercepting file access that is routed to Object Storage. But Object Storage has certain limitations regarding small files or read/write operations. Intensive use of such operations by applications would also preclude transparency. Serverless disaggregated memory and fine-grained storage services are needed in the Cloud.

3.5.4 Conclusions

We have demonstrated that Python's multiprocessing message-passing shared state design enables to seamlessly port local-parallel applications over disaggregated serverless resources in the Cloud. Despite the considerable added overheads, applications that do not require heavy access to shared memory do not exhibit performance degradation compared to a resource-equivalent VM. On the other hand, serverless cloud services, such as FaaS or object storage, allow to massively exploit the parallelism of applications to further reduce the execution time and increase the speedup, all without the need to modify the application code or architecture thanks to access transparency. These results lead us to expect that, when network latencies are reduced, access transparency will become more viable, which has important implications, such as transparently porting legacy applications to the Cloud or the ability to program the Cloud as a parallel supercomputer, thus hiding the complexities of distributed systems.

Abstraction	Description
`CloudThread`	Cloud functions are invoked like threads.
`ServerlessExecutorService`	A simple executor service for task groups and distributed parallel *for*s.
Shared objects	Linearizable (wait-free) distributed objects (e.g., `AtomicInt`, `AtomicLong`, `AtomicBoolean`, `AtomicByteArray`, `List`, `Map`).
Synchronization objects	Shared objects providing primitives for thread synchronization (e.g., `Future`, `Semaphore`, `CyclicBarrier`).
`@Shared`	User-defined shared objects. Object methods run on the DSO servers, allowing fine-grained updates and aggregates (e.g., `.add()`, `.update()`, `.merge()`).
Data persistence	Long-lived shared objects are replicated. Persistence may be activated with `@Shared(persistence=true)`.

Table 3: CRUCIAL programming abstractions

4 CRUCIAL: Serverless multi-threaded applications

CRUCIAL is a system for the development of stateful distributed applications on serverless environments. To simplify the writing of an application, CRUCIAL provides a thread abstraction that maps a thread to the invocation of a serverless function: the *cloud thread*. This abstraction can be extended to build task management systems with serverless thread pools. 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 cloud 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 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.

CRUCIAL also focuses in not increasing 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 synchronise functions at the application level, so that (imperative) implementations of popular algorithms such as *k*-means can be effortlessly ported to serverless platforms.

A complete description of the design, implementation and evaluation of CRUCIAL is detailed in D4.2. Here we provide a description of its API and programming abstractions.

4.1 CRUCIAL programming model

CRUCIAL presents an object-based programming model that can be integrated with any concurrent object-oriented language. Our prototype library supports the Java programming language. Programs in CRUCIAL resemble regular multi-threaded, object-oriented Java ones. The library is based on annotations and simple constructs that the user uses or substitutes in their code, allowing to easily move applications to the cloud. The abstractions comprise execution constructs and shared objects and are summarised in Table 3.

4.1.1 Execution abstractions

Cloud threads. Users code their applications as programs that run multiple threads concurrently. When using CRUCIAL, a conventional parallel computing `Thread` is replaced with a `CloudThread`,

which is the smallest unit of computation in the library. Tasks that run on threads are still defined as a `Runnable` and passed to a `CloudThread` that executes it. The distinction resides in that this class hides execution details that allow the tasks to run on a cloud function in the FaaS platform.

Serverless executor service. As a higher-level execution abstraction, CRUCIAL offers the `ServerlessExecutorService`. This class allows the execution of `Runnable` and `Callable` objects by implementing the Java `ExecutorService` interface. It facilitates the submission of individual tasks and fork-join parallel constructs (`invokeAll`) to the cloud, retaining the full expressivity of the original interface. Additionally, this executor also includes a distributed parallel *for* to run n iterations of a loop across m workers. To use this feature, the user specifies the in-loop code (through a functional interface), the boundaries for the iteration index, and the number of workers m.

4.1.2 State abstractions

State handling. The library already includes a set of base shared objects to support mutable shared data across serverless functions. This group consists of common objects such as integers, counters, maps, lists and arrays. These objects are *wait-free* and *linearizable*. 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. The `@Shared` annotation also gives programmers the ability to craft their own custom shared objects. The library 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)`. Distributed references are supported, permitting a reference to cross the boundaries of a cloud thread. This feature helps preserve the simplicity of multi-threaded programming in CRUCIAL.

Data Persistence. Shared objects in CRUCIAL can be either *ephemeral* or *persistent*. By default, shared objects are ephemeral and only exist during the application lifetime. Once the application finishes, they are discarded. Nonetheless, it is also possible to make them persistent with the annotation `@Shared(persistent=true)`. In such a case, the annotated object outlives the application lifetime and is only removed from storage by an explicit call.

Synchronisation Vanilla serverless functions support only uncoordinated embarrassingly parallel operations, or bulk synchronous parallelism (BSP). To provide fine-grained coordination of cloud threads, the library offers a number of primitives such as cyclic barriers and semaphores. 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.

4.2 Sample applications

Listing 10 presents an application implemented with CRUCIAL. This simple program is a multi-threaded Monte Carlo simulation that approximates the value of π. The application uses the cloud thread abstraction to coordinate a fork-join thread structure that runs several instances of a regular `Runnable` class. The tasks carry the estimation of π and use the library's shared object `counter` to store their global state. The previous fork-join pattern can also be implemented using the `ServerlessExecutorService`. In this case, instead of directly creating the threads, we simply use the content of Listing 11.

An application that outputs an image approximating the Mandelbrot set with a gradient of colours is shown in Listing 12. In this case, the shared state is a user-defined class that is annotated with `@Shared`. The basic structure of the algorithm is a simple loop that can be parallelised. The rows of the image are processed in parallel, using the `invokeIterativeTask` method of the `ServerlessExecutorService` class. This method takes as input a functional interface (`IterativeTask`) and three integers. The interface defines the function to apply on the index of the *for* loop. The integers define respectively the number of tasks among which to distribute the iterations, and the boundaries of these iterations (`fromInclusive`, `toExclusive`).

This second example illustrates the expressiveness and convenience of our library. In particular,

Listing 10: Monte Carlo simulation to approximate π.

```
public class PiEstimator implements Runnable{
    private final static long ITERATIONS = 100_000_000;
    private Random rand = new Random();
    @Shared(key="counter")
    crucial.AtomicLong counter = new crucial.AtomicLong(0);

    public void run(){
        long count = 0;
        double x, y;
        for (long i = 0L; i < ITERATIONS; i++) {
            x = rand.nextDouble();
            y = rand.nextDouble();
            if (x * x + y * y <= 1.0) count++;
        }
        counter.addAndGet(count);
    }
}

List<Thread> threads = new ArrayList<>(N_THREADS);
for (int i = 0; i < N_THREADS; i++) {
    threads.add(new CloudThread(new PiEstimator()));
}
threads.forEach(Thread::start);
threads.forEach(Thread::join);
double output = 4.0 * counter.get() / (N_THREADS * ITERATIONS);
```

Listing 11: Using the `ServerlessExecutorService` to perform a Monte Carlo simulation.

```
ServerlessExecutorService se = new ServerlessExecutorService();
List<Callable> tasks = IntStream.range(0, N_THREADS).mapToObj(i -> Executors.callable(new
    PiEstimator())).collect(Collectors.toList());
se.invokeAll(tasks);
```

as in multi-threaded programming, CRUCIAL allows to express concurrent tasks with lambdas and pass them shared variables defined in the encompassing class.

The *k*-means implementation in Listing 13 shows a more complex application that uses synchronisation primitives like a barrier.

4.3 Porting to serverless

Benefits and target applications CRUCIAL can be used not only to program serverless-native applications, but also to port existing single-machine applications to serverless. Successfully porting an application comes with several incentives; namely the ability to (i) access on-demand computing resources; (ii) scale these resources dynamically; and (iii) benefit from a fine-grained pricing for their usage. To match the programming model of CRUCIAL, Java applications that can benefit from a portage should be multi-threaded. Moreover, as with other parallel programming frameworks, they should be inherently parallel.

Methodology Low effort is required to port existing Java applications with CRUCIAL. The following steps should be taken: **(1)** Replace the `ExecutorService` or `Thread` instances with their CRUCIAL counterparts (Table 3). **(2)** Make `Serializable` each immutable object passed between cloud threads. **(3)** Substitute the concurrent mutable objects shared by threads with the equivalent ones provided by the DSO layer. For example, an instance of `java.util.atomic.AtomicBoolean` is replaced with

Listing 12: Mandelbrot set computation in a distributed parallel *for*.

```
public class Mandelbrot implements Serializable {
    @Shared(key = "mandelbrotImage")
    private MandelbrotImage image = new MandelbrotImage();

    private static int[] computeMandelbrot(int row, int width, int height, int maxIters)
        {...}

    private void doMandelbrot() {
        image.init(COLUMNS, ROWS);
        ServerlessExecutorService se = new ServerlessExecutorService();
        se.invokeIterativeTask((row) -> image.setRowColor(row, computeMandelbrot(row,
            COLUMNS, ROWS, MAX_INTERNAL_ITERATIONS)), N_TASKS, 0, ROWS);
        se.shutdown();
    }
}
```

`org.crucial.dso.AtomicBoolean`. **(4)** Regarding synchronization primitives, transform them into distributed objects. For example, a cyclic barrier can be replaced with `org.crucial.dso.CyclicBarrier`. **(5)** If the application uses the `synchronized` keyword, some rewriting is necessary. Recall that this keyword is specific to the Java language and allows to use any (non-primitive) object as a monitor [82].

CRUCIAL does not support the `synchronized` keyword out of the box since it would require modifying the JVM. Two solutions are offered: (i) create a monitor object in DSO and use it where appropriate; or (ii) create a method for the object used as a monitor that contains all the code in the `synchronized{..}` block. Then, this object is annotated as `@Shared` in the application, and the method called where appropriate. The first solution is simple, but it might not be the most efficient since it requires to move data back and forth the cloud threads that use the monitor. The second solution needs rewriting part of the original application but is more in line with the object-oriented approach in CRUCIAL and it may perform better.

Limitations and solutions The above methodology works for most applications, yet it has limitations. First, some threading features are not available in the framework —e.g., signaling a cloud thread. Second, CRUCIAL does not natively support arrays (e.g., `T[] tab`). Recall that Java offers native methods to manipulate such data types. For instance, calling `tab[i]=x` assigns the value (or reference) x to `tab[i]`. Transforming a native call is not possible with just annotations. The solution to these two problems is to rewrite the application appropriately, as in the case of `synchronized`.

Another issue is related to data locality. Typically, a multi-threaded application initialises shared data in the main thread and then makes it accessible to other threads for computation. Porting such a programming pattern to FaaS implies heavy data serialization, which is inefficient.

Instead, we can pass a distributed reference that is lazily de-referenced by the thread. To illustrate this point, consider Listing 14 which counts the number of occurrences of the word ‘‘`serverless`’’ in a document. The application first constructs a reference to the document (line 2). Then, the document is split into chunks. For each chunk, the number of occurrences of the word is counted by a cloud thread (line 8). The results are then aggregated in the shared counter ‘‘`wordcount`’’. Reading the document in full at line 2 and serializing it to construct the chunks is inefficient. Instead, the application should send a distributed reference to the cloud threads at line 8. Then, upon calling `split`, the chunks are created on each thread by fetching the content from remote storage.

Listing 13: *k*-means implementation with CRUCIAL.

```
public class KMeans implements Runnable{
    private CyclicBarrier barrier = new crucial.CyclicBarrier();
    @Shared(key = "delta")
    private GlobalDelta globalDelta = new GlobalDelta();
    @Shared(key = "iterations")
    private AtomicInteger globalIterCount = new AtomicInteger();
    // Wraps a list of @Shared centroids
    private GlobalCentroids centroids = new GlobalCentroids();

    public void run(){
        loadDatasetFragment();
        int iterCount = globalIterCount.intValue();
        do {
            correctCentroids = globalCentroids.getCorrectCoordinates();
            resetLocalStructures();
            localDelta = computeClusters();
            globalDelta.update(localDelta);
            centroids.update(localCentroids, localSizes);
            barrier.await();
            globalIterCount.compareAndSet(iterCount, iterCount++);
        } while (iterCount < maxIterations && !endCondition());
    }
}
```

Listing 14: Parallel word count.

```
public class WordCount {
   private Document document = new Document(LOCATION);
   private String word = "serverless";
   private void compute() {
      AtomicLong counter = new AtomicLong("wordcount");
      ServerlessExecutorService se = new ServerlessExecutorService();
      se.invokeIterativeTask(i -> counter.addAndGet(countWords(word, document.split(i))),
          N_TASKS, 0, N_TASKS);
   }
}
```

5 FAASM: High-performance thread-based serverless

FAASM is a high-performance stateful serverless runtime, which isolates functions using a lightweight mechanism called a *Faaslet*. Faaslets are based on threads, which operate in a shared address space on each host. This means that, while Faaslets provide isolation and fair access to resources, they also support concurrent, zero-copy access to shared state held in memory. This is in contrast to existing serverless platforms which isolate functions in their own container of VM, and do not support parallel processing on shared data. This thread-based approach makes FAASM uniquely placed to support thread-based programming models, such as OpenMP and MPI, as well as more simple applications based on pthreads.

5.1 FAASM and serverless big data

In addition to locally shared state, FAASM synchronises state across hosts using a two-tier state architecture. This two-tier state, coupled with lightweight Faaslet isolation, is how FAASM address two key problems facing highly parallel serverless big data, namely the *container resource footprint* and *data access overhead*.

The container resource footprint is the high cost associated with container-based isolation, when compared to the short-lived, high-volume functions that make up serverless big data. Containers have start-up latencies in the hundreds of milliseconds to several seconds, leading to the *cold-start* problem in today's serverless platforms [37, 49]. The large memory footprint of containers limits scalability—while technically capped at the process limit of a machine, the maximum number of containers is usually limited by the amount of available memory, with only a few thousand containers supported on a machine with 16 GB of RAM [83].

Data access overheads are caused by the stateless nature of existing container-based platforms, which force state to be maintained externally, e.g. in object stores such as Amazon S3 [61] or passed between function invocations. Both options incur costs due to duplicating data in each function, repeated serialisation, and regular network transfers. This results in current applications adopting an inefficient "data-shipping architecture", i.e. moving data to the computation and not vice versa—such architectures have been abandoned by the data management community many decades ago [47]. These overheads are compounded as the number of functions increases, reducing the benefit of unlimited parallelism, which is what makes serverless computing attractive in the first place.

Faaslets provide multi-tenant isolation with orders of magnitude lower overheads than containers or VMs. This is done in part, using *software fault isolation* (SFI) with WebAssembly [33]. Each function associated with a Faaslet, together with its library and language runtime dependencies, is compiled to WebAssembly before being uploaded to the system. The FAASM runtime then executes multiple Faaslets, each with a dedicated thread, within a single address space. For resource isolation, the CPU cycles of each thread are constrained using Linux *cgroups* [84] and network access is limited using *network namespaces* [84] and *traffic shaping*. Many Faaslets can be executed efficiently and safely on a single machine.

Since Faaslets share the same address space, they can access shared memory regions with local state efficiently. This allows the co-location of data and functions and avoids serialisation overheads. Faaslets use a two-tier state architecture, a *local* tier provides in-memory sharing, and a *global* tier supports distributed access to state across hosts. The FAASM runtime provides a state management API to Faaslets that gives fine-grained control over state in both tiers. Faaslets also support stateful applications with different consistency requirements between the two tiers.

5.2 Faaslets

Faaslets are the isolation mechanism used in FAASM and are shown in Figure 7. They are built around an instance of a WebAssembly module to provide the necessary isolation guarantees for multi-tenancy in serverless clouds; in contrast, traditional serverless systems typically rely on containers [34, 36]. Faaslets provide a more lightweight execution environment than containers by only virtualising the necessary environment for serverless functions. Therefore, Faaslets have a low memory footprint and can be spawned in the hundreds of microseconds against hundreds of milliseconds

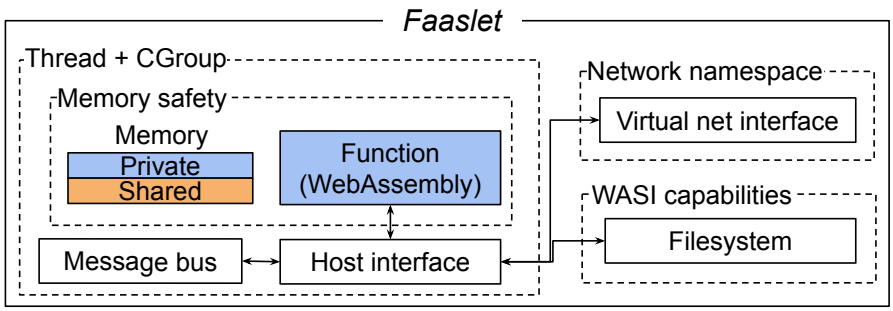


Figure 7: Faaslet isolation in FAASM

for container. This brings Faaslets much closer to the user's idea of the programming model of a *function*, which FaaS is meant to provide and is the unit of granularity that FAASM manages. We provide below details of the resource control FAASM has in place.

Memory safety using WebAssembly. The untrusted user code is compiled to WebAssembly, which can be translated to a safe intermediate representation (IR). Once instantiated, the WebAssembly code running inside the Faaslet is guaranteed to only access its linear memory through efficient bounds checking. Segments of this linear memory can, however, be mapped to multiple Faaslets simultaneously, enabling shared in-memory regions to be efficiently shared when Faaslets are co-located on a host. This can be enforced by an appropriate scheduling policy from the runtime.

CPU access using cgroups. On each host, Faaslets run as part of a shared thread pool within a process control group (cgroup)—a technology shared with containers—to establish fair local CPU access guaranteed by the Linux kernel [84].

Network access using namespaces, virtual interfaces and traffic shaping. Each Faaslet has a separate virtual network interface which resides in its own network namespace to provide isolated access to networking . Traffic shaping is applied to this virtual interface to limit the rate of traffic at the function level. This is to ensure they cannot saturate the host and is a useful monitoring point for network usage. This can be used to mitigate low bandwidth issues, especially important given that it is possible to pack many more Faaslets per machine than containers.

To mitigate the serverless cold starts (Section 2.2), users can define initialisation code separately from their main function code, during which the language runtime and packages will be loaded. The resulting WebAssembly memory can be safely serialised at this point and saved to the state which once pulled on each host set to run this Faaslet will speed up start-up times by 490× compared to the equivalent start-up process for a container.

5.3 Host interface

Faaslets interact with the platform using *import functions* that are provided to users modules by the FAASM runtime to control the execution of serverless functions or perform traditional OS and libc operations. The FAASM host interface is outlined in Table 4. The following summarises the main features of the host interface, which our programming abstraction can exploit:

Serverless-specific APIs. There are two main types of serverless operations to support. First, Faaslets can invoke other Faaslets and wait for their completion with custom mechanisms to set and get input data. Second, Faaslets can interact with the shared memory state described below (§**??**) by being provided direct pointer access to it. This latter feature makes Faaslets more suitable for big data processing than other serverless isolation such as shared-nothing containers that have to rely on HTTP protocols to operate on any shared data. Even serverless edge computing platforms such as Fastly [86] or CloudFlare [87], which also use WebAssembly, can only share state through external distributed key-value stores for their workers [88, 89].

Class	Function	Action	Standard
Calls	`byte* read_call_input()`	Read input data to function as byte array	*none*
	`void write_call_output(out_data)`	Write output data for function	
	`int chain_call(name, args)`	Call function and return the `call_id`	
	`int await_call(call_id)`	Await the completion of `call_id`	
	`byte* get_call_output(call_id)`	Load the output data of `call_id`	
State	`byte* get_state(key, flags)`	Get pointer to state value for `key`	
	`byte* get_state_offset(key, off, flags)`	Get pointer to state value for `key` at offset	
	`void set_state(key, val)`	Set state value for `key`	
	`void set_state_offset(key, val, len, off)`	Set `len` bytes of state value at offset for `key`	
	`void push/pull_state(key)`	Push/pull global state value for `key`	
	`void push/pull_state_offset(key, off)`	Push/pull global state value for `key` at offset	
	`void append_state(key, val)`	Append data to state value for `key`	
	`void lock_state_read/write(key)`	Lock local copy of state value for `key`	
	`void lock_state_global_read/write(key)`	Lock state value for `key` globally	
Dynlink	`void* dlopen/dlsym(...)`	Dynamic linking of libraries	POSIX
	`int dlclose(...)`	*As above*	
Memory	`void* mmap(...)`, `int munmap(...)`	Memory grow/shrink only	
	`int brk(...)`, `void* sbrk(...)`	Memory grow/shrink	
Network	`int socket/connect/bind(...)`	Client-side networking only	WASI
	`size_t send/recv(...)`	Send/recv via virtual interface	
File I/O	`int open/close/dup/stat(...)`	Per-user virtual filesystem access	
	`size_t read/write(...)`	*As above*	
Misc	`int gettime(...)`	Per-user monotonic clock only	
	`size_t getrandom(...)`	Uses underlying host `/dev/urandom`	

Table 4: FAASM host interface (The final column indicates whether functions are defined as part of POSIX or WASI [85].)

WASI & POSIX compatibility. This part of the host interface deals with application control of memory, files, network, clock, and random numbers within the limits of WebAssembly safety guarantees. The WebAssembly System Interface (WASI) [90] aims to standardise server-side WebAssembly. This means that user applications, which previously had to be compiled with an `unknown` operating system target, can now be compiled for the more portable `wasi` platform. The popularity of WASI is growing, and with it the number of programs that can run in Faaslets without modifications.

Interface extensibility. Although WASI-core contains a fairly small number of essential operations, it is not designed with serverless compatibility in mind. As such the FAASM host interface has the issue of striving to be a sensible serverless interface, whilst having to support both POSIX and WASI concepts. It is challenging to map existing POSIX/WASI concepts to serverless because those two system interfaces can conflict with each other. For example, POSIX and WASI both have a concept of threads, but it can be non-trivial to figure out what their accompanying synchronisation mechanisms should translate to.

Byte arrays. Function inputs, results and state are represented as simple byte arrays, as is all function memory. This avoids the need to serialise and copy data as it passes through the API, and makes it trivial to share arbitrarily complex in-memory data structures.

5.4 Building FAASM functions

Each function in FAASM is first compiled to WebAssembly and uploaded to the system. To convert this into an executable, it needs to be combined with the host interface and other FAASM helper libraries. This process is outlined in Figure 8 and is made up of three steps, with the user only aware of the first. This first step generates a WebAssembly module that can safely handled onwards thanks to the WebAssembly guarantees [33]. FAASM relies on a trustworthy open-source WebAssembly *em-*

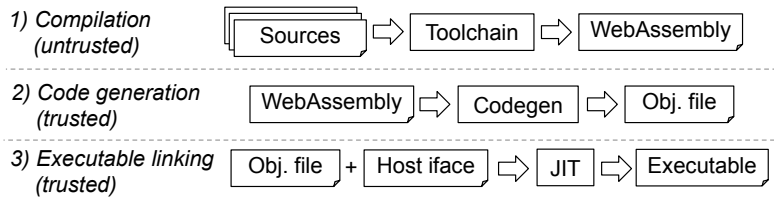


Figure 8: Creation of a Faaslet executable

bedder, WAVM [91] and LLVM-JIT libraries [92], to validate and manipulate WebAssembly modules and object code, which make up the second and third steps. A more detailed description for C/C++ functions is as follows:

1. The user project and the FAASM public library function declarations are compiled to a WebAssembly module using the FAASM toolchain. The library functions are unresolved symbols, which means that they are declared as import functions in the module. The user can then **upload the Wasm module** to a FAASM host.

2. The Wasm module is verified by the embedder and compiled to an **object file** that can be linked with the FAASM libraries later. This allows to update the host interface without recompiling the user's project.

3. The library public function definitions are provided as export functions and intrinsics along with the rest of the host interface. Those are linked with the object files and the linker can now resolve the previous *import functions* and output a **trusted executable**.

In step 1, the user compiles their code using the FAASM toolchain that includes the popular LLVM compiler infrastructure [92] (i.e., Clang, compiler-rt, libcpp, etc.), which was built to cross-compile to WebAssembly. The shipped libc is musl [93], a small and fast libc implementation compared to the traditional glibc [94]. FAASM is also compatible with dynamic languages too. It supports the ubiquitous CPython language runtime for which the host interface supports dynamic library loading. FAASM relies on projects such as Pyodide to compile the main scientific Python packages to Wasm [95, 96] and import them in user programs.

5.5 State

Faaslets expose state through their low-level state API, or through *distributed data objects (DDO)*. DDOs are language-specific classes that expose a convenient high-level state interface, and are implemented on top of FAASM's low-level key/value state API. FAASM employs a *two-tier* state architecture that combines local sharing with global distribution of state: a *local tier* provides shared in-memory access to state on the same host; and a *global tier* allows FAASM to synchronise state across hosts.

5.5.1 High-level state abstraction

DDOs hide the two-tier state architecture, providing transparent access to distributed data. Functions, however, can still access the state API directly, either to exercise more fine-grained control over consistency and synchronisation, or to implement custom data structures. Each DDO represents a single state value, referenced throughout the system using a string holding its respective state key.

FAASM writes changes from the local to the global tier by performing a *push*, and read from the global to the local tier by performing a *pull*. DDOs may employ push and pull operations to produce variable consistency, such as delaying updates in an eventually-consistent list or set, and may lazily pull values only when they are accessed, such as in a distributed dictionary. Certain DDOs are immutable, and hence avoid repeated synchronisation.

Listing 15 shows both implicit and explicit use of two-tier state through DDOs to implement stochastic gradient descent (SGD) in Python. We use Python for the following examples, as it makes it

Listing 15: Distributed SGD application with Faasm

```
1  t_a = SparseMatrixReadOnly("training_a")
2  t_b = MatrixReadOnly("training_b")
3  weights = VectorAsync("weights")
4
5  @faasm_func
6  def weight_update(idx_a, idx_b):
7    for col_idx, col_a in t_a.columns[idx_a:idx_b]:
8      col_b = t_b.columns[col_idx]
9      adj = calc_adjustment(col_a, col_b)
10     for val_idx, val in col_a.non_nulls():
11       weights[val_idx] += val * adj
12       if iter_count % threshold == 0:
13         weights.push()
14
15 @faasm_func
16 def sgd_main(n_workers, n_epochs):
17   for e in n_epochs:
18     args = divide_problem(n_workers)
19     c = chain(update, n_workers, args)
20     await_all(c)
21     ...
```

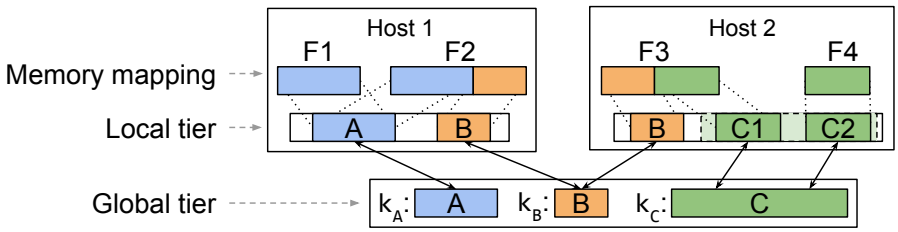


Figure 9: FAASM two-tier state architecture

easiest to convey the business logic in the examples succinctly. The `weight_update` function accesses two large input matrices through the `SparseMatrixReadOnly` and `MatrixReadOnly` DDOs (lines 1 and 2), and a single shared weights vector using `VectorAsync` (line 3). `VectorAsync` exposes a `push()` function which is used to periodically push updates from the local tier to the global tier (line 13). The calls to `weight_update` are chained in a loop in `sgd_main` (line 19).

Function `weight_update` accesses a randomly assigned subset of columns from the training matrices using the `columns` property (lines 7 and 8). The DDO implicitly performs a pull operation to ensure that data is present, and only replicates the necessary subsets of the state values in the local tier—the entire matrix is not transferred unnecessarily.

Updates to the shared weights vector in the local tier are made in a loop in the `weight_update` function (line 11). It invokes the `push` method on this vector (line 13) sporadically to update the global tier. This improves performance and reduces network overhead, but introduces inconsistency between the tiers. SGD tolerates such inconsistencies and it does not affect the overall result.

5.5.2 Two-tier state architecture

Faaslets represent state with a key/value abstraction, using unique *state keys* to reference *state values*. The authoritative state value for each key is held in the global tier, which is backed by a distributed key-value store (KVS) and accessible to all Faaslets in the cluster. Faaslets on a given host share a local tier, containing replicas of each state value currently mapped to Faaslets on that host. The local

tier is held exclusively in Faaslet shared memory regions, and Faaslets do not have a separate local storage service, as in SAND [13] or Cloudburst [16].

Figure 9 shows the two-tier state architecture across two hosts. Faaslets on host 1 share state value A; Faaslets on both hosts share state value B. Accordingly, there is a replica of state value A in the local tier of host 1, and replicas of state value B in the local tier of both hosts.

The `columns` method of the `SparseMatrixReadOnly` and `MatrixReadOnly` DDOs in Listing 15 uses *state chunks* to access a subset of a larger state value. As shown in Figure 9, state value C has state chunks, which are treated as smaller independent state values. Faaslets create replicas of only the required chunks in their local tier.

Ensuring local consistency. State value replicas in the local tier are created using Faaslet shared memory. To ensure consistency between Faaslets accessing a replica, Faaslets acquire a *local read lock* when reading, and a *local write lock* when writing. This locking happens implicitly as part of all state API functions, but not when functions write directly to the local replica via a pointer. The state API exposes the `lock_state_read` and `lock_state_write` functions that can be used to acquire local locks explicitly, e.g. to implement a list that performs multiple writes to its state value when atomically adding an element. A Faaslet creates a new local replica after a call to `pull_state` or `get_state` if it does not already exist, and ensures consistency through a write lock.

Ensuring global consistency. DDOs support varying levels of consistency between the tiers as shown by `VectorAsync` in Listing 15. To enforce strong consistency, DDOs must use *global read/write locks*, which can be acquired and released for state keys using the functions `lock_state_global_read` and `lock_state_global_write`, respectively. To perform a consistent write to the global tier, an object acquires a global write lock, calls `pull_state` to update the local tier, applies its write to the local tier, calls `push_state` to update the global tier, and releases the lock.

5.5.3 Experimental evaluation

To demonstrate the use of DDOs in an experiment, we implement the same distributed *stochastic gradient descent* (SGD) algorithm as in Listing 15 in C/C++ to run text classification on the Reuters RCV1 dataset [97]. This updates a central weights vector in parallel with batches of functions across multiple epochs.

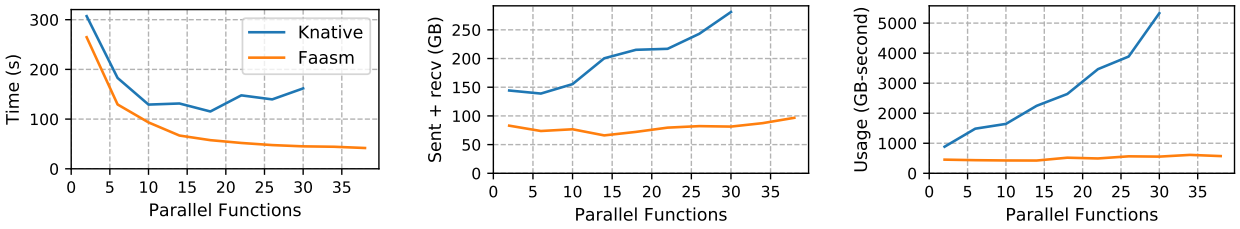
5.5.4 Experimental set-up

Serverless baseline. To benchmark FAASM against a state-of-the-art serverless platform, we use Knative [98], a container-based system built on Kubernetes [99]. All experiments are implemented using the same code for both FAASM and Knative, with a Knative-specific implementation of the Faaslet host interface for container-based code. This interface uses the same undelrying state management code as FAASM, but cannot share the local tier between co-located functions. Knative function chaining is performed through the standard Knative API. Redis is used for the distributed KVS and deployed to the same cluster.

FAASM integration. We integrate FAASM with Knative by running FAASM runtime instances as Knative functions that are replicated using the default autoscaler. The system is otherwise unmodified, using the default endpoints and scheduler.

Testbed. Both FAASM and Knative applications are executed on the same Kubernetes cluster, running on 20 hosts, all Intel Xeon E3-1220 3.1 GHz machines with 16 GB of RAM, connected with a 1 Gbps connection.

Metrics. In addition to the usual evaluation metrics, such as execution time, throughput and latency, we also consider *billable memory*, which quantifies memory consumption over time. It is the product of the peak function memory multiplied by the number and runtime of functions, in units of GB-seconds. It is used to attribute memory usage in many serverless platforms [36, 39, 41]. Note that all memory measurements include the containers/Faaslets and their state.



(a) Training time (b) Network transfers (c) Memory usage

Figure 10: Machine learning training with SGD with FAASM and containers (Knative)

5.5.5 Experimental results

To test the scalability of the prototype we ran both Knative and FAASM with increasing numbers of parallel functions.

Figure 10a shows the training time. FAASM exhibits a small improvement in runtime of 10% compared to Knative at low parallelism and a 60% improvement with 15 parallel functions. With more than 20 parallel Knative functions, the underlying hosts experience increased memory pressure and they exhaust memory with over 30 functions. Training time continues to improve for FAASM up to 38 parallel functions, at which point there is a more than an 80% improvement over 2 functions.

Figure 10b shows that, with increasing parallelism, the volume of network transfers increases in both FAASM and Knative. Knative transfers more data to start with and the volume increase more rapidly, with 145 GB transferred with 2 parallel functions and 280 GB transferred with 30 functions. FAASM transfers 75 GB with 2 parallel functions and 100 GB with 38 parallel functions.

Figure 10c shows that billable memory in Knative increases with more parallelism: from approx. 1,000 GB-secs for 2 functions to over 5,000 GB-secs for 30 functions. The billable memory for FAASM increases slowly from 350 GB-secs for 2 functions to 500 GB-secs with 38 functions.

The increased network transfer, memory usage and duration in Knative is caused primarily by data shipping, e.g. loading data into containers. FAASM benefits from sharing data through its local tier, hence amortises overheads and reduces latency. Further improvements in duration and network overhead come from differences in the updates to the shared weights vector: in FAASM, the updates from multiple functions are batched per host; whereas in Knative, each function must write directly to external storage. Billable memory in Knative and FAASM increases with more parallelism, however, the increased memory footprint and duration in Knative make this increase more pronounced.

5.6 Scheduling

The scheduling policy is crucial for having efficient state-sharing. Faaslets provide the sharing mechanisms to utilise the FAASM state efficiently but it is up to the scheduler to provide an efficient policy to co-locate Faaslets running the same function. FAASM defines *warm* nodes as being hosts on which the state on which an instance of the Faaslet Wasm module is already loaded. Each node is aware via the global state of the list of *warm* nodes for a given user function and can offload work if necessary when it reaches saturation. The FAASM scheduler is similar in this regard to a *distributed shared state* scheduler such as Omega [100]. The metadata of the Faaslet runtime is serialised in a protocol buffer message [101] sent between workers to instruct the welcoming host on which Faaslet it needs to run.

This scheduling policy is sufficiently simple and self-contained that FAASM can be executed using a number of disaggregated compute platforms. In CloudButton we use Knative as shown in Figure 1. We integrate CloudButton with Knative by running FAASM runtime instances as Knative functions that are replicated using the default autoscaler. The system is otherwise unmodified, using the default endpoints and scheduler. The default Knative scheduler passes functions to FAASM runtime instances in a round-robin fashion, after which they will share work amongst themselves as described above.

6 FaasMP: Transparent use of OpenMP APIs with FAASM

OpenMP is a popular parallel programming API based on multi-threading and shared memory. It is used in several domains including machine learning [102], linear algebra [103] and big data [30]. OpenMP encourages programmers to distribute code across threads, explicitly specifying which data is shared and which data is unique to a given thread. Existing OpenMP implementations target a single host, but the underlying concept of small, concurrent tasks lends itself well to a serverless programming model.

6.1 Background: Open Multi-Processing (OpenMP)

OpenMP is an API in the form of compiler directives or *pragmas* and a runtime library to write cross-platform multi-threaded programs for Fortran and C/C++ on shared-memory devices [104]. It is supported by every major C/C++ compiler [105, 106] and, due to its popularity in the HPC community, it is also implemented in multiple scientific compilers [107]. Its programming model follows the fork/join model in which all threads share a common address space. Only the thread stack is private to them along with explicitly marked private variables.

The initial root directive, namely `#pragma omp parallel`, is placed on top of a code block, to indicate that this section should be run in parallel. Other directives can be used inside of this parallel section to perform common parallel programming operations such as creating a critical section, waiting for other threads, or distributing slices of an array to threads. At compile time, parallel sections are extracted into functions, shared variables are made into stack variables, and the directives are transformed by the compiler into calls to the compiler-specific runtime library that is responsible for running the extracted functions in parallel. At runtime, the compiler-specific runtime library invokes threading APIs to parallelise the code.

OpenMP is still under active development after more than two decades since its original release. Recent versions focus on supporting GPUs [104]. OpenMP's most notable peer in the HPC field is MPI, which is the de-facto standard for distributed applications. The two APIs are complementary and often used together, with OpenMP providing local parallelism, and MPI distributing tasks across hosts [108]. Existing work has attempted to remove this dependence on MPI by both adding distribution to OpenMP itself (Section 6.2), converting OpenMP to MPI [109], and offloading OpenMP to the cloud [31].

6.1.1 OpenMP API

OpenMP is intuitive for a majority of programmers thanks to its shared-memory model that allows for incremental addition of parallelism. Its paradigm is based on the fork/join model where a *master* thread (with an id of 0) forks into *slave* threads ($id > 0$) that can run a code block marked with the `parallel` pragma. Threads have access to common synchronisation mechanisms such as locks or barriers. We describe those as *core* in Table 5, and we will use their descriptive names to refer to them (e.g. `critical` or `barrier`). The rest of the API in the table can be divided into two categories: (1) the *task* API which we will not implement because it is not widely used in practice even though it could fit well into a serverless model by replacing glue code and ad-hoc await mechanisms; and (2) the *GPU* API specifically for GPU-based processing.

The pragmas are accompanied by public library functions declared in `omp.h` to interact with the OpenMP runtime for operations such as getting the thread number or setting the next number of threads. Code examples can be found in the next section, which highlight the reasons why OpenMP is so popular:

1. Most idiomatic OpenMP code still works when compiled with non-OpenMP compilers, which means that programmers can often ignore the OpenMP semantics to understand an application. This makes OpenMP **simple**.

2. The burden of using low-level platform-specific parallel APIs such as `pthread` and identifying shared variables is put on the compiler. This makes the code **portable**.

TYPE	PRAGMA	DESCRIPTION
Core	`atomic`	Atomic access to memory location.
	`barrier`	Synchronisation of all threads in this region.
	`critical`	Next codeblock is a critical section.
	`flush`	Synchronise the view of the objects in memory.
	`for [simd]`	Distribute the for loop to the threads [with SIMD instructions].
	`master`	Next codeblock should only be executed by master thread.
	`parallel [reduce]`	Start of a parallel section [with a variable to accumulate at the end].
	`single`	Next codeblock should only be executed by one thread.
Tasks	`task`	Define a task.
	`taskgroup`	Specifies which tasks to wait on.
	`taskloop [simd]`	Distribute the loop as tasks.
	`taskwait`	Wait for child tasks.
	`taskyield`	Suspend current task.
GPU	`target & distributed`	OpenMP 4.2 and above clauses for GPU control.

Table 5: Overview of main OpenMP pragmas

3. The runtime behaviour gives **predictable performance** [110]. Users who avoid expensive operations such as long critical sections, frequent forking and joining can expect linear performance with respect to the number of cores.

6.1.2 Compiler code transformation

The compiler performs code transformations on the abstract syntax tree (AST) to convert the parsed OpenMP code into regular C/C++ code that can be code generated. Listing 16 is a basic OpenMP program that counts the number of threads that execute a parallel section marked with the directive `# pragma omp parallel` and returns it.

We instruct the OpenMP compiler to interpret the OpenMP pragma by using the OpenMP flag: `clang -fopenmp count-unsafe.c`. Intuitively, we can understand that the parallel section has been given to a thread per core. The variable `number_of_threads` was shared automatically, such that each thread could increment it. We show in Listing 17 what the compiler's internal representation of

```
1 int main(void) {
2     int number_of_threads = 0;
3     # pragma omp parallel
4     {
5         number_of_threads += 1;
6     }
7     return number_of_threads;
8 }
```

Listing 16: Racy thread-count.c OpenMP example

```
1  static void parallel_section(int *number_of_threads)
2  {
3      *number_of_threads++;
4  }
5
6  int main()
7  {
8      int number_of_threads = 0;
9      __kmpc_runtime_fork(parallel_section, &number_of_threads);
10     return number_of_threads;
11 }
```

Listing 17: Output of the compiler transformation (simplified)

the program looks like if we could convert it back into C++ after the compiler applied the OpenMP transformations.

On lines 1 to 4 of this source code, the compiler extracts the parallel section into a new function, called `parallel_section`. On line 9 in the `main` function, instead of calling the `parallel_section` function directly, the compiler generates a call to the runtime library function `__kmpc_runtime_fork`, passing the function `parallel_section` as an argument. On lines 8 and 9, the shared variable is created on the stack and given by reference to the parallel section. The runtime library forking function, called `__kmpc_fork_call` in Clang's implementation [105], is responsible for running the `parallel_section` function on each available OpenMP threads.

Compiling the same `count-unsafe.c` program (Listing 16) in a traditional manner with `clang count-unsafe.c`, works and returns 1 because the OpenMP `# pragma omp parallel` compiler directive is ignored by the compiler unless OpenMP compilation is specified. This shows that OpenMP can be a non-intrusive API.

Being based on the C/C++ languages, OpenMP programs offer little concurrency safety and cannot check at compile time for unsafe memory operations. Our claim that OpenMP code can be entirely transparent does not apply to runtime library functions calls, for example, that users must place behind pre-processor `#ifdef OpenMP` blocks. Race conditions, deadlocks, and other concurrency-related issues such as false sharing [111] may still happen when running an OpenMP program. The concurrency of execution is not abstracted away for the programmer, only the platform-specific constructs.

In the initial program (Listing 16), no synchronisation mechanism was used for `number_of_threads`, which means that our implementation was racy.[6] Several low-level concurrency primitives are available for threads to synchronise (§6.1.1). Listing 18 shows (i) how the `critical` pragma can be used to synchronise our program (line 9); (ii) how to control the visibility of variables explicitly with `default(none)` and `shared` (line 7); and (iii) how the public library function `omp_get_max_thread` can be used to obtain in advance the number of threads the next parallel section will run with (line 5).

6.1.3 Runtime library

Popular C/C++ compilers supporting OpenMP implement their own runtime libraries such as Intel's compiler `libiomp`, the GNU C/C++ Compiler (GCC) with `libgomp` [106] or Clang/LLVM and `libomp` [105]. Compilers have varying levels of support for OpenMP API, which has grown over time. Unfortunately, LLVM's library, already the most complex library because of its multi-platform support, has been implementing undocumented ABI compatibility with `libgomp`, which we will need to circumvent for our WebAssembly implementation.

[6] Technically, this is dependent on the platform that the program runs on (i.e. atomic increments). OpenMP shared-variables are however not guaranteed to be synchronised. Threads have a local view of the shared-data that needs to be flushed and/or synchronised to be valid.

```c
#include <assert.h>
#include <omp.h>

int main(void) {
    int expected_num_threads = omp_get_max_threads();
    int number_of_threads = 0;
    # pragma omp parallel default(none) shared(number_of_threads)
    {
        # pragma omp critical
        {
            number_of_threads++;
        }
    }
    assert(number_of_threads == expected_num_threads);
}
```

Listing 18: Complete thread-count.c

6.2 Related work on distributed OpenMP

Whilst it is possible to use MPI and OpenMP together for big data applications, it is not straightforward, with no support from the frameworks themselves. More crucially for this project, MPI applications fundamentally embrace a serverful design (§2), trading off multi-tenancy and isolation against performance. The main challenge for distributing OpenMP-only programs is that **the API and user applications are designed assuming the shared-memory access latency is similar to the rest of the memory**. We can distinguish three different approaches for distributed OpenMP that tackle this challenge in different way: (i) by translating it to MPI; (ii) by using distributed shared memory; and (iii) by offloading to big data platforms in the Cloud. All these approaches require deployment in exclusive clusters and cannot scale based on the application's demand.

6.2.1 OpenMP to MPI translation

This strategy first operates source-to-source translation from OpenMP to Single Program Multiple Data (SPMD), the bases of MPI and generates collective communication code based on the semantic of the translated OpenMP constructs [112, 109, 113]. This approach also requires a runtime system to monitor, schedule and optimise communication between the threads and perform dynamic dataflow analysis. For example, the runtime needs to manage a control flow graph for every array involved in the global communication, or pre-fetch data when parallel sections are called in a loop and the runtime can identify recurring patterns. The performance of these approaches thus relies mainly on their runtime and does not scale past 100 threads.

6.2.2 OpenMP on software distributed shared memory

This approach is the most popular because page-based distributed shared memory (DSM) can support existing code with little modification and so was released in a commercial compiler, Intel Cluster OpenMP [114], that provided support for compiling OpenMP applications to run on small clusters. OpenMP allows most of the program's execution to be consistent only around the synchronisation points thereby allowing distributed processes to operate on their local DSM pages efficiently. Moreover, we can expect users to optimise for page locality, which is an optimisation pattern for page-based DSM. We detail below the details of the two main systems and our takeaways from the literature for this project.

Intel Cluster OpenMP. Cluster OpenMP (ClOMP) [114] was released in 2006 as part of the Intel C/C++ and Fortran compiler. The only porting step required by the Intel compiler was data privatisation and marking certain variables explicitly sharable for the DSM. However, ClOMP showed bad

performance on fine-grained data distribution, especially through ethernet, and was outperformed or equalled by MPI on existing applications [115]. The DSM layer was more suited for procedural Fortran than for C/C++ pointers logic and as such showed poor performance for common programming patterns like C++ STL algorithms. The minimal overheads measured in micro-benchmarks for a distributed reduce was three orders of magnitude slower than local memory [115], but those figures could be amortised for large parallel sections.

libMPNode. `libMPNode` is a modification of the Linux target only GNU `libgomp` that run on Popcorn Linux which provides thread migration and multiple reader/single writer protocol for paged-granularity DSM in a cluster. Threads are assigned to nodes using a new `node` keyword thus hindering the programs portability and not offering support for unmodified applications. They further need to optimise their algorithms based on the underlying DSM page size to minimise co-location.

Thanks to the fixed thread placement, it implements some OpenMP directives in a way that requires a minimal number of open network communication (one per node, instead of one per thread), which allows for good scaling across nodes after fine-tuning the code for distribution. Other issues include a lack of multi-tenant isolation, and an excessive resource footprint, notably from the distribution layer.

Summary of DSM-based OpenMP. `libMPNode` needs to prevent DSM page thrashing that occurs when multiple nodes are trying to fetch the same remote page (but not necessarily the same data) by placing contention element on separate pages. FAASM's *FDiffs* operate at a byte-level granularity rather than a page-level granularity, hence do not face this problem.

`libMPNode` makes the remark that OpenMP allows for threads to have a local view of the data until it is explicitly flushed or reaches a synchronisation point, but that their DSM enforces sequential consistency for every page access. They suggested they could improve their performance if their DSM could differentiate between the two sorts of accesses. We incorporate this into our concurrency and synchronisation primitive's design.

With ClOMP, the failure of one process terminates the whole program, monitored through heartbeats, and the failure mode of `libMPNode` is assumed to be similar. OpenMP is not originally designed for fault-tolerance, and general distributed fault-tolerance is not achievable for existing programs but a subset of the API can be made fault-tolerant in a serverless environment.

Results from the DSM approaches showed that **OpenMP program scalability does not imply page-based DSM scalability** [115].

6.2.3 Offloading to the cloud

The recent addition of *target* devices to OpenMP has led to the idea of using the OpenMP API as a Spark client [116] for Map-Reduce jobs [31]. This can be used in applications like edge-compute for mobile devices [117]. This approach tries to democratise the utilisation of the cloud by re-using a known C/C++ API, but falls short in terms of generalisability, fine-grained scalability, or useful applications hindsight.

Our key design idea on how to achieve our goal of making OpenMP serverless is to implement a new OpenMP runtime library. Our library, FaasMP, replaces the runtime library of the compiler such as `libomp`, `libiomp` or `libgomp`, which is in charge of the threading specifics of the platform.

6.3 Requirements for shared memory multi-processing

In this section, we explain in a practical fashion what concepts an existing, efficient, implementation of an OpenMP runtime library is based on. We start by looking at the open-source code of `libomp`, the LLVM runtime library [118]. Note that `libomp` is a large and complex codebase, which supports both UNIX and Windows platforms. It will be important to study the LLVM codebase, notably for understanding its private API.

Next, we use small OpenMP code examples and `strace`—a Linux utility to trace system calls [119]—with a filter to only show activity from `libomp`. Table 6 shows the observed system calls and groups them into categories. The system calls are ticked if there is a reasonable equivalent implementation

Category	Description	POSIX System call	Wasm	Serverless
Memory	Grown/Shrink heap	`mmap/munmap`	✓	✓
		`brk/sbrk`	✓	✓
Forking	Create thread	`clone`	✓	✓*
	Shared-memory	`mprotect`	✓	✓*
Scheduling	Cede CPU	`sched_yield`	✓	∼
	Pin thread to CPU core	`sched_[get|set]affinity`	✓	✗
Signals	Set signal handler	`rt_sigaction`	✗	?**
	Set allowed signals	`rt_sigprocmask`	✗	?**
Synchronisation	Fast user-space locking	`futex`	✓	✗
	Auto-lock release	`[get|set]_robust_list`	✓	?**

*: Never done together; **: Further semantics ramifications

Table 6: WebAssembly and serverless compatibility for `libomp` system calls

in WebAssembly and if there exists or could exist a serverless implementation of the system call. WebAssembly can support most of the required operations, but many operations in serverless are rendered difficult, notably because of the distribution of the functions onto inhomogeneous domains. We now explain each category in more detail.

(a) Memory. Local memory management is already supported transparently in stateful serverless environments [26, 27] but FAASM additionally supports local shared-memory between Faaslets.

(b) Forking. The other categories of the table all depend on the chosen implementation of `clone`, which is called with the `CLONE_THREAD` flag: it does not `fork` the process but creates a thread in the same address space [84]. Although with Crucial [27] (see §4), we also explore the use of AWS Lambda [60] as a "cloud thread" executor, the lack of shared memory in such cases is an issue when providing efficient threads.

Conversely, Faaslets [26] are capable of sharing memory, not through a forking interface but through calls to `mmap` using the `MAP_SHARED` parameter. This differs from a POSIX thread that shares the entire process memory. Faaslets share only discrete memory regions in the WebAssembly memory through the FAASM state API because the general process memory needs to be isolated. We explore extending this behaviour to allow for thread forking using the WebAssembly threading mechanism proposal implemented in WAVM and use Faaslets as thread executors.

(c) Scheduling. Yielding the CPU can be done in WebAssembly and in serverless computing in general. In distributed systems with weak scheduling guarantees, however, it may not be possible to yield to a specific thread, which is a useful mechanism in cooperative multitasking and even a necessity to implement high-performance synchronisation constructs.

Similarly, the CPU `affinity` system calls to pin a thread or a process to a CPU core could be allowed in Wasm but is difficult to implement in a multi-tenant serverless environment and seem to go against the principles of elasticity of the compute layer. We can further infer that if `libomp` needs to pin threads and become cache-conscious, it relies on a predictable memory model, preferably homogenous across threads. This could also be evidence of thread pooling by `libomp`.

(d) Signals. Signals are a POSIX form of inter-process communication (IPC), which, if integrated into a serverless platform, may help solve a commonly raised FaaS issue: the lack of effective direct communication between functions. There are, however, concerns about such support in the FAASM

host interface:

1. How to support signals in a WASI-based host interface, when WebAssembly does not have processes and thus does not support signals?
2. How to implement efficient and/or fault-tolerant distributed signalling?

Signals are often used for performing asynchronous I/O, but FAASM proposes other, more idiomatic, ways of doing asynchronous I/O, by spawning a Faaslet to handle an I/O request for example. Both FAASM and OpenMP encourage the use of shared memory as the preferred communication mechanism, and therefore supporting POSIX signals would not provide any benefits for supporting existing OpenMP programs.

(e) Synchronisation. Synchronisation and consistency mechanisms are perhaps the most difficult mechanisms to integrate into a distributed serverless environment. For example, a `futex` is a mechanism for fast userspace locking, and threads can have a `robust_list` of futexes that they hold and that should be automatically released if the thread terminates without explicitly releasing them [84]. The `futex` interface has strong performance guarantees, and the `robust_list` has strong safety guarantees. The CAP theorem [62] can immediately instruct us of the difficulties arising when designing a distributed system trying to deliver these guarantees. Moreover, depending on the implementation of those mechanisms, user applications may be exposed to new classes of issues (e.g. distributed deadlocks, starvation, and thrashing) that programs were not designed for.

6.4 FaasMP Design

Summarising the previous sections, to execute multi-threaded applications correctly with shared memory, FAASM must provide consistency guarantees across hosts, alongside suitable synchronisation primitives such as locks and barriers. FAASM makes distributed shared memory consistent by sending FDiffs between hosts that communicate updates to the shared address space (Section 6.4.1).

OpenMP uses *reductions* to aggregate parallel updates to shared variables without the coordination overhead of mutexes. FAASM supports reductions using *merge operations*, which allow it to combine multiple *FDiffs* to the same memory region using arithmetic operations (Section 6.4.2). Finally, FAASM provides custom implementations of coordination primitives including mutexes, barriers and latches (Section 6.4.3).

6.4.1 Synchronising changes to shared memory

By default, OpenMP assumes only weak consistency guarantees on the memory shared between threads; stronger consistency is requested explicitly through synchronisation primitives. Assuming code is free from data races, FAASM must correctly execute multi-threaded applications: it must ensures that writes to shared memory from a child thread are visible to the parent thread when it joins that child. Changes must be visible to all threads when entering a critical section, or exiting an explicit or implicit barrier.

When FAASM needs to execute a child thread, it creates a new *Faaslet* from a snapshot of the main *Faaslet*. This snapshot is maintained until all child threads have finished execution, and acts as the *master snapshot* for the shared address space. FAASM synchronises all subsequent changes across distributed *Faaslets* via this master snapshot: it receives updates to it from remote hosts, and uses it to calculate updates to send to remote hosts.

A *Faaslet* maps its linear memory from the master snapshot, if executing on the main host, or a replica of the master snapshot, if executing on a worker host. The *Faaslet* then tracks the changes made to the shared address space by application code. FAASM write-protects all memory pages of the *Faaslet*'s linear memory using `mprotect()` and handles the page faults caused by application code by marking the page dirty and resetting its read/write permissions.

To send these changes back to the main host when the *Faaslet* completes or reaches a barrier, the *Faaslet* performs a byte-wise comparison of the modified pages with its local copy of the master snapshot. This results in a list of FDiffs that specify the offset at which the changes occurred and the

modified bytes. The main host receives these *FDiffs* from worker hosts and uses them to update the master snapshot.

The FAASM runtime must update the master snapshot replicas on remote hosts, e.g. when exiting a barrier. It transmits only the *FDiffs* required to update the remote replicas, and not the whole snapshot. To enable this, the FAASM runtime on the main host keeps track of which bytes have been updated by incoming *FDiffs*, then sends a new set of *FDiffs* with these changes to the worker hosts.

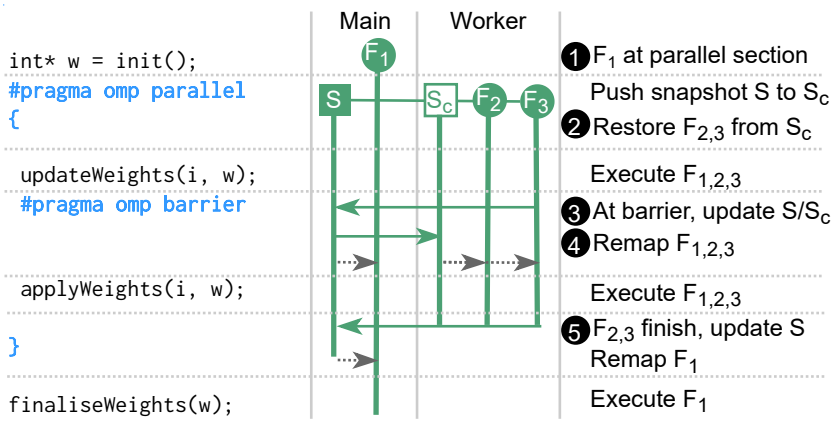


Figure 11: Synchronisation of a shared address space. Distributed *Faaslets* execute an OpenMP `parallel` section with a `barrier`.

Fig. 11 gives an example of shared memory synchronisation, which shows how FAASM executes an excerpt of OpenMP code using 3 *Faaslets* across 2 hosts. ❶ When the main *Faaslet* enters an OpenMP `parallel` section, it triggers a *control point* at which FAASM creates the master snapshot of the shared address space from the main *Faaslet*. ❷ FAASM then creates 2 more *Faaslets* from a replica of this main snapshot on the worker host, and each *Faaslet* executes the body of the `parallel` section. ❸ When all *Faaslets* have reached the `barrier`, each creates a list of *FDiffs* that the FAASM runtime on the main host uses to update the master snapshot. ❹ On exiting the barrier, the main hosts's FAASM runtime sends another list of *FDiffs* to update the snapshot replica on the worker host. All *Faaslets* then remap their own linear memory to the local copy of the snapshot and continue execution. ❺ At the end of the `parallel` section, the parent *Faaslet* joins the child *Faaslets*, and again FAASM uses the *FDiffs* from each *Faaslet* to update the main snapshot. Finally, the main *Faaslet* remaps its memory from the snapshot.

6.4.2 Supporting reductions on shared variables

OpenMP allows multiple threads to aggregate changes to shared variables using *reductions*. A reduction aggregates the changes made by threads without the coordination overhead of protecting these updates with mutexes. FAASM supports reductions through *merge operations*, which allow FAASM to combine multiple *FDiffs* on a single shared variable.

Figure 12 shows OpenMP code for a `parallel` section that performs disjoint updates to a shared vector and a `reduction` section that performs a summation on a shared variable. FAASM spawns 3 child *Faaslets* when the main *Faaslet* reaches the parallel section, creating the main snapshot on the main host and a replica on the worker host. Each *Faaslet* maps its linear memory from its local copy of the snapshot.

In the first parallel section, each *Faaslet* updates its value in the `w` vector. The resulting *FDiffs* can be written directly to the main snapshot without a merge operation. In the `reduction` section, each thread updates their local copy of the variable `x`, generating a *FDiff* on the same memory region. Since the `reduction` specifies a summation over `x`, FAASM combines these *FDiffs* in the master snapshot using a `sum`.

Tab. 7 lists the merge operations supported by FAASM. They include simple arithmetic operations

of summation, subtraction, multiplication and division, as commonly found in parallel reductions. The operations involve four values: A_0, the starting value in the main snapshot; B_0, the value held in the copy of the snapshot on the remote host; B_1, the updated value after the thread has executed on the remote host; and A_1, the value written to the main snapshot by the operation.

```
int n = get_max_num_threads()
int[] w = initWeights(n);

#pragma omp parallel {
  int t = omp_get_thread_num();
  w[t] = updateWeight(w, t, n);
}

applyWeights(w);
int x = 0;

#pragma omp for reduction(+:x) {
for(int i=0; i<n; i++) {
  x += w[i] > threshold ? 1 : 0;
}
```

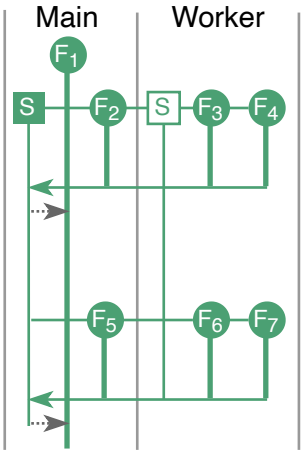


Figure 12: Reductions in OpenMP. *Faaslets* on 2 hosts make non-conflicting updates in a `parallel` section, and perform a reduction.

Merge operation	Formula	Data types
sum	$A_1 = A_0 + (B_1 - B_0)$	All numeric
subtract	$A_1 = A_0 - (B_0 - B_1)$	All numeric
multiply	$A_1 = A_0 * (B_1 / B_0)$	All numeric
divide	$A_1 = A_0 / (B_0 / B_1)$	All numeric
overwrite	$A_1 = B_1$	Arbitrary bytes

Table 7: Merge operations supported by FAASM. FAASM overwrites the original value A_0 in the master snapshot with value A_1. B_0 is the value seen in the snapshot on the remote host before the *Faaslet* executed, and B_1 is the value after execution.

6.4.3 Synchronisation primitives

In addition to providing consistent shared memory and reduction operations, FAASM must support the synchronisation primitives in multi-threaded code that control concurrent access to shared data. FAASM offers the following primitives:

Mutexes. A mutex guarantees that only one thread can access data at a given time. In FAASM, application code that acquires a mutex triggers a *control point*, and the associated *Faaslet* requests a lock on the mutex from the FAASM runtime on the main host. When locking the mutex, the FAASM runtime returns the *FDiffs* to update that *Faaslet*'s local copy of the shared memory snapshot. This way, the thread holding the mutex is guaranteed to observe the updates of other threads that have also held it; when releasing the mutex, the *Faaslet* returns its own set of *FDiffs* to the FAASM runtime.

Atomic operations. Atomic arithmetic operations do not guarantee consistency, only atomicity. To perform such operations, each *Faaslet* acquires a host-local mutex to avoid data races on the local copy of the shared memory. FAASM then uses a merge operation corresponding to the arithmetic operation to merge the resulting *FDiffs* across hosts.

Barriers. A barrier is either implicit or explicit: an implicit barrier is introduced by a `parallel` section; an explicit barrier is added manually. Barriers require that all threads block until they have

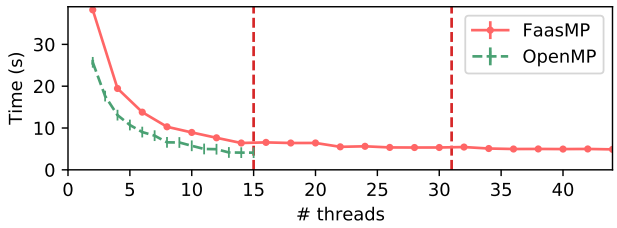


Figure 13: Dense matrix multiplication

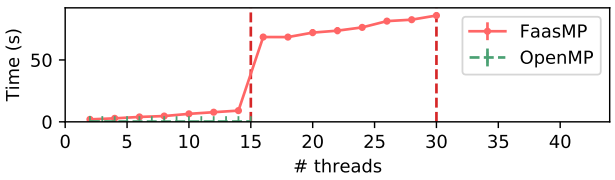


Figure 14: Lulesh simulation

Figure 15: Scalability in big data setting using multi-threading and shared memory with increasing numbers of threads (Dotted vertical lines indicate boundaries at which the application is distributed across more cloud workers.)

completed the barrier. Afterwards, all threads must observe a consistent view of the shared memory. On entering a barrier, *Faaslets* send their *FDiffs* to the main host and block. After all threads have completed, the main host sends the aggregated *FDiffs* to all *Faaslets*, which unblock.

Latches. A latch allows threads to decrement a counter and/or wait for it to reach zero. Latches are used implicitly in `nowait` OpenMP operations: the main thread blocks until all child threads have reached the latch. *Faaslets* on remote hosts can make non-blocking requests to the master host to decrement a latch, or blocking requests to wait for the latch to reach zero.

6.5 Experimental evaluation

This experiment explores the scalability of FaasMP in the context of a data-intensive ("big data") scientific workload. Our goal is to validate our CloudButton KPIs in terms of the scalability achieved. We measure the performance when using FAASM's shared memory synchronisation and the distributed coordination primitives, as implemented by FaasMP.

We run two multi-threaded data-intensive applications using OpenMP: a dense parallel matrix multiplication from the ParResKernels benchmarks [120], and Lulesh [121], a big data hydrodynamics simulation application. Note that matrix multiplication is computationally intensive but makes comparatively light use of FaasMP's shared memory synchronisation and coordination primitives.

The Lulesh [121] workload solves a Sedov blast problem to simulate a material undergoing the forces from a shock [122]. The workload uses OpenMP to divide the large-scale simulation into a series of iterations over a cube of fluid subdivided into a 3D mesh. Each iteration is parallelised over a fixed number of threads, synchronising changes to a shared model at the end of the iteration, thus stressing FAASM's shared memory synchronisation, and its distributed coordination primitives. To investigate scalability, we execute the workload using increasing numbers of threads, (i) natively on a single machine and (ii) a distributed FAASM cluster deployed on top of Knative.

Figure 15 shows the measured execution times of both applications running with an increasing number of threads in FAASM and native OpenMP. Native OpenMP applications cannot be distributed, hence we only measure each OpenMP execution up to a number of threads equal to the CPU cores available on a single host.

For matrix multiplication, FAASM adds a constant overhead of 30% on a single host, due to the overhead of performing floating-point arithmetic in WebAssembly [123]. However, the overhead for Lulesh on a single host becomes progressively worse as we add more threads, due to the cost of making system calls through the *Faaslet* interface, and the contention arising from synchronising internal *Faaslet* state.

To scale beyond the parallelism available on a single host, FAASM adds remote threads on other hosts. The first three threads added in this manner do not improve upon the single host matrix multiplication performance, as the cost of cross-host synchronisation outweighs the benefits of increased

parallelism. As FAASM adds more remote threads, we see a 5% improvement in the performance of the single host version. Once FAASM scales Lulesh across more than one host, we see an order of magnitude increase in overhead versus the best single-host performance, which increases steadily as it introduces more remote threads. This increasing overhead is due to the increasing levels of cross-host communication involved in coordination and shared memory synchronisation, and we further analyse it below.

FAASM's performance overhead for the Lulesh experiment is high. It is partly down to some characteristics of the specific workload, and partly down to implementation issues that can be improved upon. The workload is divided into short-lived parallel sections separated by barriers, hence coordination overheads outweigh the benefit of distribution. Each *Faaslet* executes for less than 1 ms, after which it must wait for the slowest straggler to complete and synchronise its *FDiffs* with the main host. The implementation offers the following opportunities for improvement: (i) each *Faaslet* tracks changes to the shared address space using write-protected pages and signal handling, this can be done more efficiently with recent Linux kernels and `userfaultfd` [124]; (ii) each *Faaslet* copies each *FDiff* before sending, which can be avoided by making the transport layer zero-copy; and (iii) the FAASM runtime exhibits high levels of contention on a single host, which can be mitigated with more efficient shared data structures and locks.

7 FaasMPI: Bridging the gap between HPC and cloud

The two worlds of High-Performance Computing (HPC) and Cloud Computing have traditionally shown little overlap, each having their own disjoint sets of popular languages and frameworks. HPC is primarily focused on performance and fine-grained control of underlying resources, while the Cloud targets ease of use and hides underlying hardware from users.

Accordingly, Fortran and C/C++ are the most popular HPC languages [29], and HPC frameworks like MPI and OpenMP expose users to hardware-specific features such as SIMD instructions and GPU offloading [104]. Newer HPC languages such as Chapel [125] and Charm++ [126] introduce high-level programming constructs, with NumPy-like arrays implemented with Chapel in Arkouda [127]. However, these are yet to see wide adoption outside the HPC community. In contrast, frameworks commonly used in Cloud environments like Spark [128] and Flink [129] offer high-level APIs [1] in dynamic languages such as Python. Serverless providers most commonly target Python and Javascript support [60, 39, 40], with little or no support for popular HPC languages and frameworks.

One of the stated goals of CloudButton is to bridge this gap by transparently executing HPC applications on serverless infrastructure, thus providing low cost, flexible scaling, without losing the expressivity and control of traditional HPC frameworks. We do this with FaasMPI, native support for MPI built into FAASM.

7.1 Motivating serverless MPI

MPI is a widely used standard for writing distributed applications, and while it is still most commonly employed in high-performance computing (HPC), it also crosses the divide into non-HPC frameworks. MPI support is found in machine learning frameworks like Horovod [32] and Microsoft's CNTK [130] and Alchemist [131] demonstrates an MPI backend for Spark.

MPI supports point-to-point and collective communication, both synchronously and asynchronously, and users express applications as a set of distributed workers sharing immutable messages. This fits well with the serverless paradigm for three reasons: (i) MPI applications are already structured around large numbers of small distributed tasks; (ii) message-passing can be efficiently implemented using existing serverless storage mechanisms; (iii) tasks address each other through numeric "ranks", so are independent of the underlying networking and communication layer. Actor-based programming is similarly well suited to serverless for the same reasons, and has been explored in PLASMA [9]. However, the breadth and volume of existing MPI codebases dwarfs that of actor-based applications, so it is a more compelling option given the aims of CloudButton. With serverless MPI we can support a wide variety of existing use-cases in big data, machine learning, fluid dynamics, genomics, astrophysics and other scientific applications [132].

7.2 FaasMPI's integration in Faasm

MPI 1.0 was released in 1994 and has been developed and augmented ever since. Although more recent developments have added advanced, useful features, it is the basic point-to-point messaging and collective communication from earlier MPI releases that underpins the majority of open-source MPI code today [29]. For this reason, FaasMPI targets only this core functionality, namely: (i) synchronous and asynchronous point-to-point messaging; (ii) broadcast and all-to-all; (iii) scatter, gather and all-gather; (iv) reduce and all-reduce. FAASM also supports custom types and custom reductions. A full list of the MPI functions supported by FaasMPI is given in Table 8.

MPI applications normally execute in a static environment on a set of hosts provisioned ahead of time. In contrast, FAASM and other serverless platforms aim to scale up and down to meet a user's need. To address this disconnect, FAASM lets users specify the level of parallelism they require for their MPI application on a *per request* basis. This means that users can execute the same application at different scales without changing any configuration or redeploying the code.

Users can compile MPI applications using the standard FAASM toolchain, which is based on LLVM tools such as Clang [133]. As with all FAASM functions, the output of this compilation is a WebAssembly file that can be uploaded and invoked on a FAASM cluster.

MPI Category	Function	Action
Environment	`MPI_Init()`	Ensure all functions initialised
	`MPI_Comm_size()`	Number of functions in communicator.
	`MPI_World_size()`	Number of functions in world.
	`MPI_Finalize()`	Finish MPI operations.
	`MPI_Abort()`	Exit MPI application.
Point-to-point	`MPI_Send()`	Send message to function (sync).
	`MPI_Isend()`	Send message to function (async).
	`MPI_Recv()`	Receive message from function (sync).
	`MPI_Irecv()`	Receive message function (async).
	`MPI_Probe()`	Get information on incoming message.
	`MPI_Wait()`	Wait for async operation.
Collective	`MPI_Bcast()`	Broadcast to all other functions.
	`MPI_Alltoall()`	Send all-to-all message.
	`MPI_Barrier()`	Wait for all functions to reach barrier.
	`MPI_Scatter()`	Divide array across all functions.
	`MPI_Gather()`	Receive from all functions into array.
	`MPI_Allgather()`	All-to-all version of `MPI_Gather`.
	`MPI_Reduce()`	Reduce data from all other functions.
	`MPI_Allreduce()`	All-to-all version of `MPI_Reduce`.

Table 8: MPI functions supported in FaasMPI

FAASM itself is built on *Faaslets*, a lightweight isolation mechanism which uses WebAssembly for memory safety [33]. Faaslets allow functions to interact with the underlying host through a specialised *Host Interface*, which supports standard POSIX-like calls for memory management, file I/O and networking, as well as serverless-specific calls for sharing state and interacting with other functions. MPI is implemented as an extension of this Host Interface, with calls incurring the same minimal overheads as the other functions in the interface.

7.3 FaasMPI design

To support MPI-like message passing in FAASM, FaasMPI provides each *Faaslet* with a virtual address for asynchronous messaging, and groups *Faaslets* into *FGroups* (Section 7.3.1). Network bandwidth is the bottleneck for most message-passing applications, and FaasMPI implements two mechanisms to mitigate these overheads. First, *Faaslets* can be migrated mid-execution to improve locality (Section 7.3.2). Second, FaasMPI provides locality-aware implementations of MPI's collective communication API to minimise the number of cross-host messages (Section 7.3.3).

7.3.1 Virtual addressing for *Faaslets* and *FGroups*

Each *Faaslet* that executes a process in a multi-process/message-passing (i.e. MPI) must be able to message any other *Faaslet*. To support this message passing, FaasMPI associates *Faaslets* with long-lived virtual address for communication. In addition, FaasMPI organises *Faaslets* into *FGroups*. The virtual address for a *Faaslet* is then the group id together with a logical index inside the group. Using this virtual address, *Faaslets* can then asynchronously send and receive messages to and from other *Faaslets* in the group by referring to that index. To accomodate to the asynchronous nature of a serverless environment, where there may exist delays between the scheduling of different *Faaslets*, or *Faaslets* may be migrated mid-execution, FaasMPI implements the transport layer in an asynchronous fashion: sending messages do not block the sender *Faaslet*, and the receiver *Faaslet* receives the message whenever it is available.

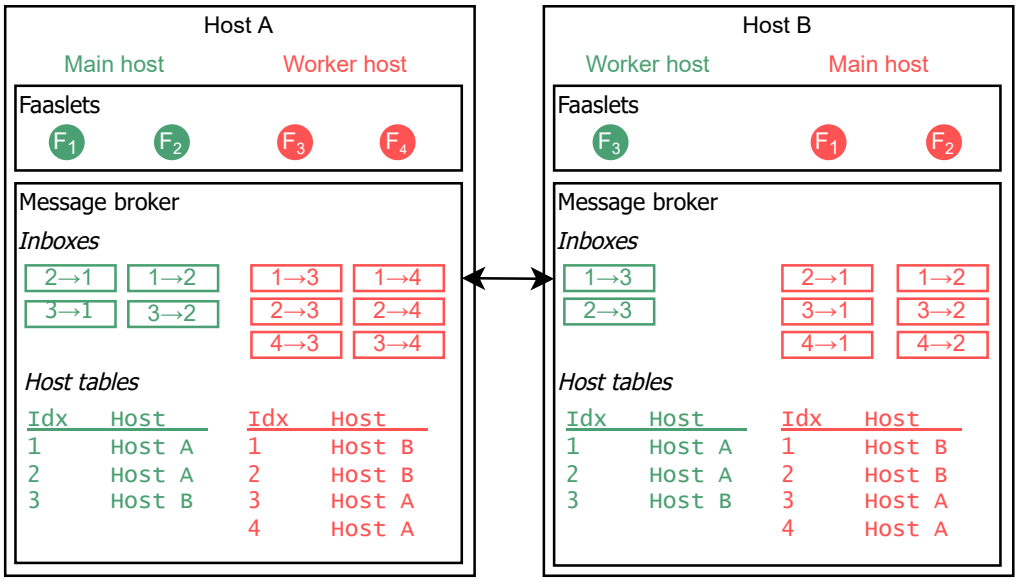


Figure 16: FaasMPI message passing architecture. FAASM runtime instances act as either the *main* host or *worker* host for each application. They add, remove and migrate *Faaslets*, and asynchronously pass messages between *Faaslets*.

By default, all *Faaslets* that execute an application are in the same *FGroup*. FaasMPI may create new *FGroups* on MPI calls like `MPI_Comm_create()`, which allows the application to control communication groups. A *FGroup* assigns each *Faaslet* an *index*, which, as introduced before, FaasMPI uses as an address for that function in its implementation of message passing APIs. Each FAASM runtime that executes a *Faaslet* holds a replica of the *FGroup* metadata with a *host table*, which maps the indexes of *Faaslets* in the *FGroup* to the host on which they have been scheduled. For each *Faaslet* of the group executed on a host, the FAASM runtime has a set of queues to buffer messages sent to that index, one queue for each other *Faaslet* in the group, these are called *inboxes*.

When a *Faaslet* reaches a function that requires sending a message, e.g. `MPI_Send()`, the FaasMPI implementation of the FAASM runtime in the host of the sending *Faaslet* looks up the recipient index in the host table for that group. If the recipient is on the same host, FAASM directly enqueues the message on the relevant in-memory queue. This results in low-latency intra-host messaging compared to using the local loopback network interface or inter-process communication (IPC). If the recipient is on another host, FAASM sends the message to the runtime on that host, where it is enqueued.

Figure 16 shows an overview of two applications distributed across two hosts, and a representation of the internal state of their respective *FGroups*. The first application (green) is running in two *Faaslets* on Host A, and one *Faaslet* on Host B, while the second application (red) is running in two *Faaslets* on each host. Each *Faaslet* in the first application has two inboxes, as there are two other *Faaslets* from which it may receive messages, and each *Faaslet* in the second application has three inboxes, as there are three *Faaslets* from which it may receive messages. The host tables for each application are replicated across both hosts, and record which *Faaslets* for each application are executing on which host.

7.3.2 Migrating *Faaslets*

To reduce the effect of network overheads when executing an MPI application using FaasMPI on FAASM, it is desirable to minimise the number of cross-host messages sent. To that extent, the FAASM runtime can decide to migrate *Faaslets* running MPI code to co-locate MPI functions. Migration decisions are determined by a *scheduling policy*, and are a consequence of the multi-tenant nature of FAASM: when the MPI application is first scheduled, *Faaslets* are scattered across available hosts; after other applications finish, FAASM may be able to group some of the scattered *Faaslets*.

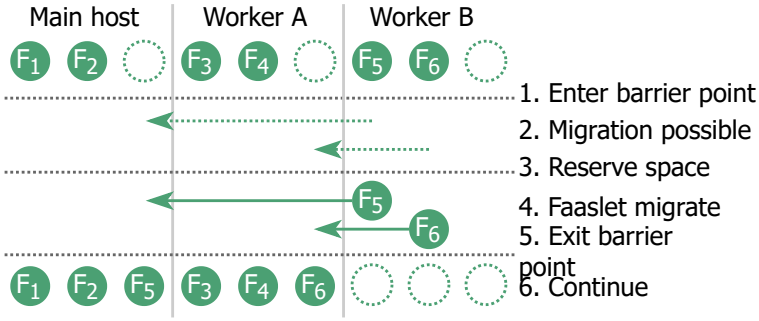


Figure 17: *Faaslet* migration at `MPI_Barrier`.

To simplify the migration process, *Faaslet* migration may only be carried out at special MPI barrier calls like `MPI_Barrier` or `MPI_Allreduce`. These are special MPI calls that block all *Faaslets* of an application. Figure 17 illustrates *Faaslet* migration. When *Faaslets* reach one of these barrier calls, they wait for a notification from the application's main host. When the main *Faaslet* reaches the barrier on the main host, it queries the *scheduler* for migration decisions. The scheduler, periodically and in the background, applies its scheduling policy and decides on function migrations if the current function execution deviates from the desired allocation. It then sends messages to all FAASM runtimes on the hosts involved in the migrations. To migrate a function, the involved FAASM runtimes reserve the necessary resources for the *Faaslets*. If the resources have become unavailable, the migration is aborted. After that, the *Faaslets* to be migrated perform the migration and notify the main host. Once the main host has received notifications from all migrated *Faaslets*, it allows the *Faaslets* to exit the barrier point.

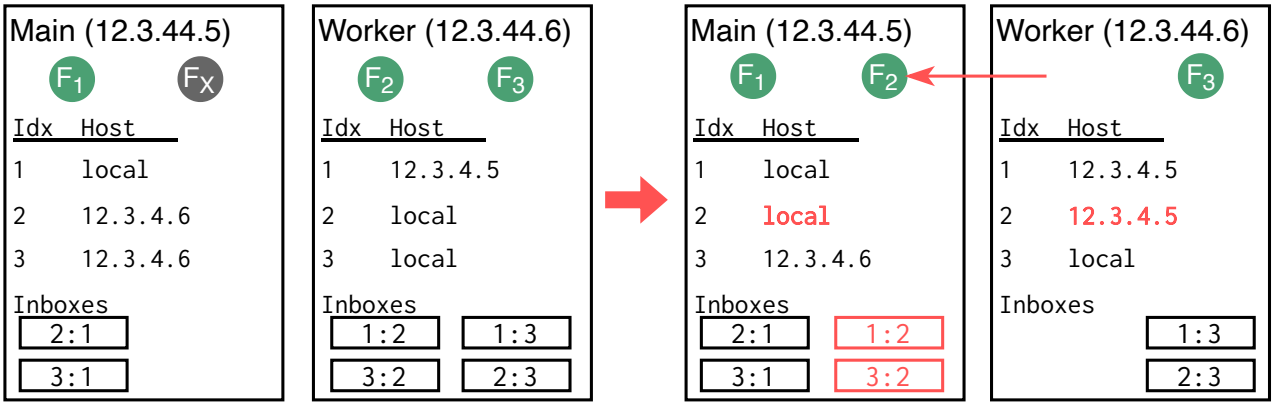


Figure 18: Preserving a *FGroup* while migrating. One function from the *FGroup* is migrated from a worker host to the main host, and FAASM updates the host-local metadata and queues.

When migrating *Faaslets* across hosts, the FAASM runtime must also update the metadata and queues associated with *FGroups* to which the migrating *Faaslets* belong. Figure 18 shows how FAASM updates a *FGroup* during migration. Initially, the main host executes one *Faaslet* from the group alongside a *Faaslet* from another application; the worker host executes two other *Faaslets* from the group. When the *Faaslet* from the other application completes, it frees up resources on the main host; when the *Faaslets* reaches a barrier call, FAASM migrates one of the functions from the worker host to the main host. Before completing the migration, each FAASM runtime updates its address table and creates or deletes queues to accommodate the new or departing *Faaslet*, respectively.

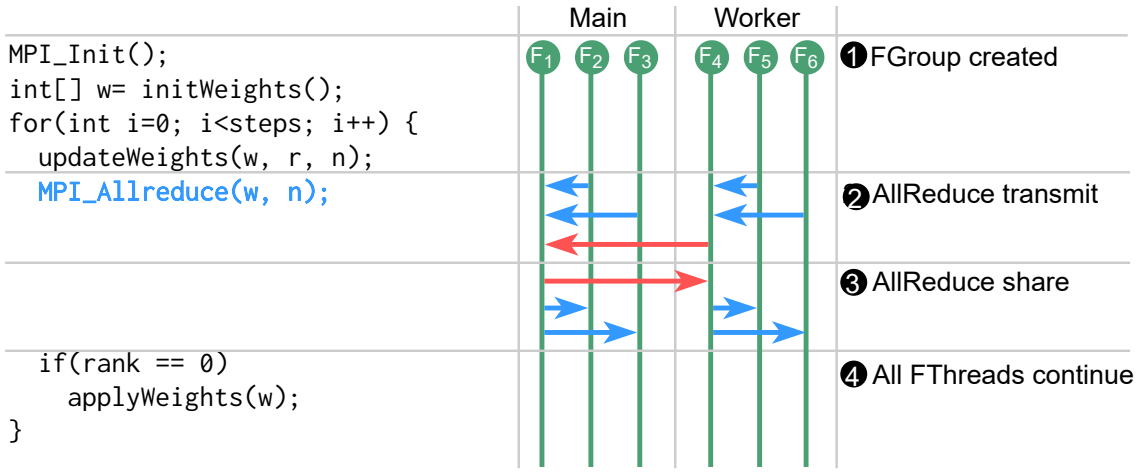


Figure 19: MPI collective communication. A host-leader on each host sends/receives intra-host messages (blue) and sends/receives cross-host message (red) to/from other host-leaders.

7.3.3 Optimising collective communications

In addition to simple point-to-point messaging, many message MPI make use of *collective communication* operations, such as all-reduce and broadcast. These operations are widely used in distributed ML training through specialised libraries [134, 135, 136]. FAASM supports several of these operations (see Table 8), and implements them in a *FGroup*-aware manner. This is, the implementation minimises latency (*i.e.* number of cross-host messages) by exploiting knowledge of *Faaslet* placement to maximise intra-host messaging.

Figure 19 shows the underlying message passing performed by FaasMPI when application code makes a call to `MPI_Allreduce()`. ❶ When FAASM creates an *FGroup*, it selects one *Faaslet* on each host to be the *host-leader* for that host. Any messages that need to be sent to *Faaslets* on other hosts are sent by all *Faaslets* to their host-leader, which batches the messages into single cross-host requests. All-reduce takes place in two steps: ❷ an initial reduce in which results from all *Faaslets* on each host are sent to the main host via their host-leader; and ❸ a broadcast of the final result to all *Faaslets*, which is delivered via their host-leader.

FAASM's implementation of collective communication operations reduces the cross-host messages to one per remote host involved in each step. It then uses fast in-memory queues for the *Faaslet* to host-leader communication. This approach reduces latency (Figure 7.4.1) and bandwidth usage, and enables pipelining: after a *Faaslet* has asynchronously messaged its local leader, it can continue execution.

7.4 Experimental evaluation

To evaluate if FaasMPI achieves CloudButton's scalability KPIs, we compare FaasMPI against scientific applications written in OpenMPI [137], one of the most popular open-source OpenMPI frameworks, and the same code cross-compiled (lift-and-shift) to run on FAASM.

We pick two data- and compute-intensive scientific applications: the ParRes Kernels [120] and LAMMPS [138, 139]. The ParResKernels are a collection of compute kernels that can be used to compare parallel programming frameworks. Each kernel reproduces a common parallel application pattern and runs it in a loop. For example, kernels reproduce behaviours like a sparse matrix multiplication, large matrix transpose, or collective communication patterns like broadcast and all-reduce. LAMMPS is a popular molecular dynamics simulator written in C++ and MPI. It can run a variety of large-scale simulations and benchmarks.

We deploy FaasMPI and OpenMPI on a Kubernetes cluster [99] on Microsoft Azure [140]. For the native deployment (OpenMPI), we package the OpenMPI code and runtime on a Docker image and

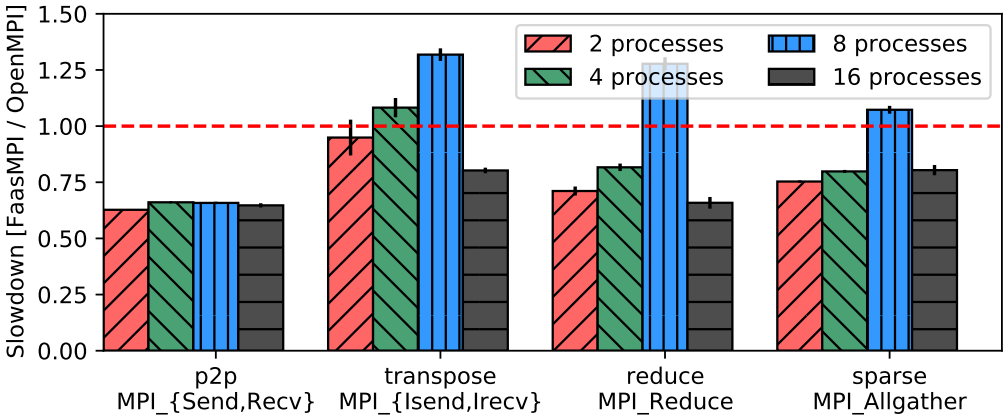


Figure 20: Scalability for MPI kernels from ParResKernels [120] (This compares FaasMPI to native OpenMPI.)

deploy it in a static cluster using a `ReplicaSet`. We deploy FaasMPI as a serverless service on top of Knative [141].

7.4.1 ParRes kernels

This experiment explores the base performance overhead introduced when executing kernels from the ParRes Kernels benchmarking suite [120]. As the suite isolates particular aspects of parallel and message passing communications, results can be interpreted as base overheads for FaasMPI's MPI's implementation.

For this experiment, we use FAASM to deploy 4 kernels with increasing levels of parallelism on 4 hosts, and measure the performance of FaasMPI compared to running natively with OpenMPI.

Figure 20 shows that the average slowdown is between 0.6–1.25× compared to native, i.e. FaasMPI runs equally (if not) faster across different kernels, degrees of parallelism, and distributions from 1 to 4 hosts. FaasMPI uses shared memory (same process) for inter-rank communication whereas OpenMPI uses IPC. As a consequence, kernels benefit from faster intra-host messaging in FaasMPI. Furthermore, the implementation of FaasMPI's collective communication primitives, e.g. `MPI_Reduce()`, reduces inter-host messages, exploiting the faster local messaging path (Section 7.3.3). However, the cross-host network layer in CloudButton must support concurrent applications from different users, this adds an overhead to distribution for some kernels (Section 7.3.1).

7.4.2 Molecular simulations using LAMMPs

In this experiment, we investigate the compute and network overheads of message passing with FaasMPI versus a native MPI implementation when executing a complex long-running scientific application.

As a workload, we use one of the 5 standard benchmarking problems in the LAMMPS benchmarking suite: the Lennard-Jones (LJ) atomic fluid simulation with 4 million atoms. To stress FaasMPI's communication layer, we also take the *controller* example [142], and manually increase the number of synchronisation steps. This way, we achieve three orders of magnitude more cross-host messaging than the LJ benchmark. We refer to the LJ benchmark as *compute-bound*, and the modified *controller* one as *network-bound*. We increase the degree of parallelism for LAMMPS both on FaasMPI and natively on OpenMPI.

Figure 21 shows that, for the compute-bound benchmark, the elapsed time (right vertical axis) and speed-up (left vertical axis) of both FaasMPI and OpenMPI are similar across parallelism degrees: from 1 parallel function to 16 parallel functions (distributed across 4 hosts). FaasMPI slightly outperforms OpenMPI due to the faster intra-host messaging and the locality-aware collective communication implementation.

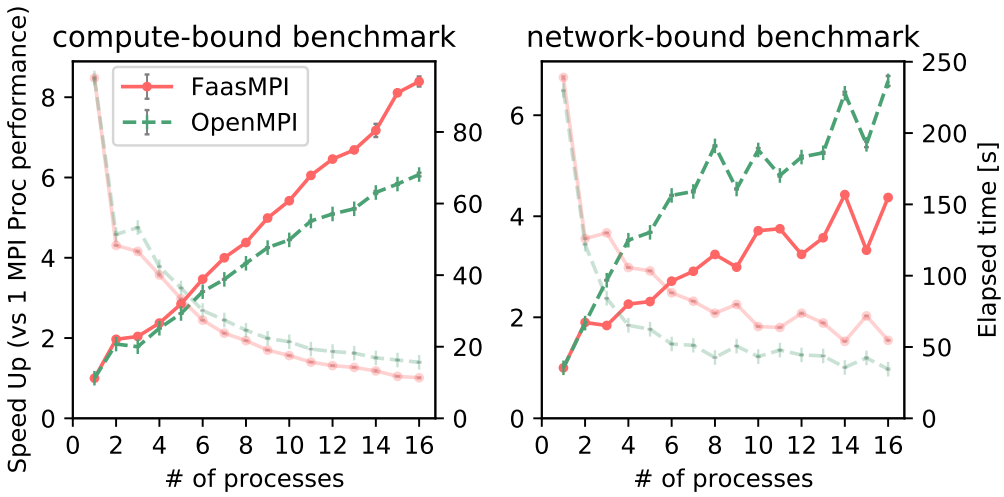


Figure 21: Scalability for message passing (LAMMPS) with different serverless resources

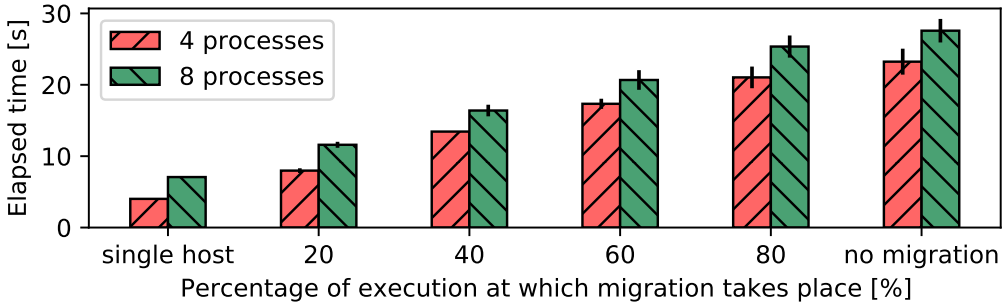


Figure 22: Execution time under rank migration with all-to-all message passing (We migrate half of the serverless resources at runtime to maximise locality.)

For the network-bound benchmark, FaasMPI introduces a consistent overhead of around 1.25× compared to native, with peaks in runtime for particular degrees of parallelism. The consistent slowdown is due to an additional level of indirection in FaasMPI's transport layer to support execution of concurrent applications. The runtime peaks appear when running with 9, 12, and 15 parallel functions, respectively. These parallelism degrees have the highest ratios of inter- to intra-host messages due to the usage in LAMMPS of cartesian MPI communicators: LAMMPS lays out MPI processes in a cartesian grid, and only sends messages between neighbours. A higher inter- to intra-host message ratio means that LAMMPS benefits less from in-memory messaging, and relies more on FaasMPI's inter-host network layer, which, as described, adds overhead due to the management of *Faaslets* groups and multi-tenancy.

7.4.3 Migration experiments

The next experiment explores the performance impact when FaasMPI migrates MPI ranks of an MPI application at runtime.

As a synthetic workload, we execute a network-bound *all-to-all* kernel that performs an all-to-all synchronisation over a vector in a loop. We provision a KNative cluster with two *Standard_D8_v5* hosts. We then enforce distribution by adopting a scheduling policy that allocates half the resources on each host. We then trigger a barrier control point at either 20%, 40%, 60% or 80% of the iterations, and measure the execution time of the kernel (including migration) with different numbers of parallel functions. For 4 parallel functions we use *Standard_D4_v5* hosts, and for 8 we use the *Standard_D8_v5*

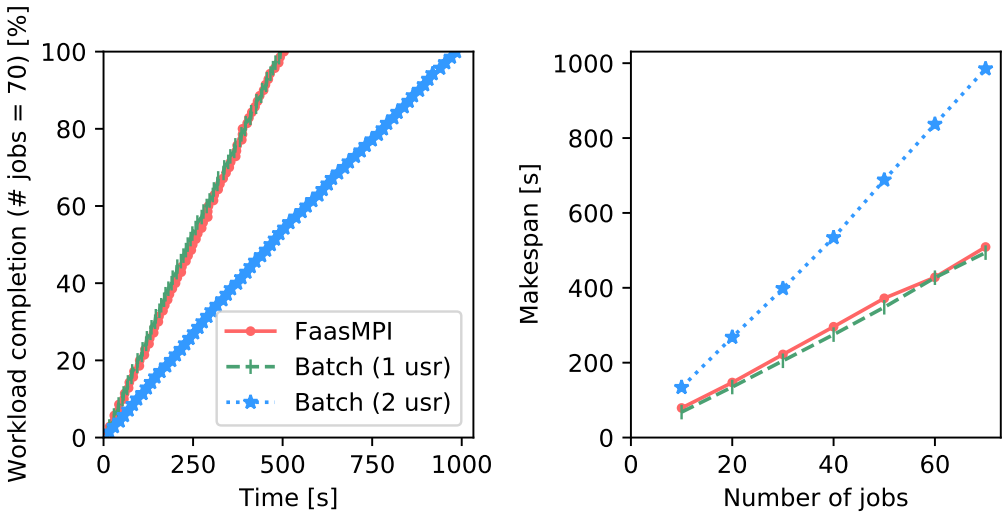


Figure 23: Progress of running a trace of scientific applications on shared cluster (left); makespan of trace (right) (This compares FaasMPI to OpenMPI with a batch scheduler.)

hosts. As baselines, we compare execution time when all functions execute on the same host (single host) and are not migrated at all (no migration).

Figure 22 shows that, with FaasMPI, it is always worth migrating ranks to maximise co-location. Migration takes approximately 0.5 seconds. This is 8% of the execution time of 8 processes-single host, and 2% of the execution time of 8 processes-no migration. As a consequence, the overhead that FaasMPI introduces when migrating (Section 7.3.2) is sufficiently low to reduce the elapsed time: no migration is 5.8× slower than single host, and migrating at 40% of execution results in a 42% runtime reduction (versus no migration). This is true even if the migration happens late into the kernel execution: when the barrier control point is at 80%, migrating results in a runtime reduction of 10%. When the number of functions doubles from 4 to 8, overall execution time increases, as the workload is proportional to the number of parallel functions, but the benefits are the similar: 40% runtime reduction migrating at 40% of execution, and 8% when migrating at 80%.

7.4.4 Multi-tenancy experiments

This last experiment explores the benefits of using FaasMPI to execute scientific applications on a shared cluster. This is, the performance benefits of supporting multi-tenant MPI clusters.

As a workload, we generate a trace of molecular simulation jobs with varying levels of parallelism. Each job runs LAMMPS [138, 139]. We pick the Lennard-Jones (LJ) atomic fluid simulation with 4 million atoms, as it is one of the five standard benchmarking problems in the LAMMPS benchmarking suite [143]. We deploy a 4-node (*Standard_D8_v5*) Kubernetes cluster, and schedule jobs sequentially, until all of them are completed.

For the Batch baseline, we use an implementation of a batch scheduler that imitates the behaviour of Azure Batch: it allocates jobs at the granularity of VMs assigned to a given user. (We could not use Azure Batch directly as it currently does not support scheduling jobs belonging to different users.) We report the progress of completion for one particular trace and the total makespan (time between first job is submitted and last one is finished).

Figure 23 shows that FaasMPI's performance is equivalent to that of a dedicated cluster (1 usr) with the additional isolation guarantees of *Faaslets*. FaasMPI's performance does not deteriorate when increasing the number of users (as it is a multi-tenant runtime), whereas the batch OpenMPI baseline decreases linearly (2 usr) with the number of users that share the cluster resources. This improvement is due to FaasMPI's ability to schedule jobs with a process granularity rather than a user/VM granularity.

8 Conclusions

In this deliverable, we have described three programming models that we have developed in CloudButton to better support the building stateful serverless applications: Lithops, CRUCIAL, and FAASM. Our goal has been to support a range of popular programming languages and paradigms, and enable a true "lift-and-shift" experience for users when moving their workloads to a serverless cloud.

In Lithops, we have presented a programming model to transparently distribute Python applications in a serverless environment. Lithops provides implementations for two popular APIs for parallel programming in Python: Map-Reduce, and *multiprocessing*. By replacing the library import for its Lithops' counterpart, users can automatically deploy single-node Python code to a variety of serverless backends.

With CRUCIAL, we have presented an object-based programming model to run multi-threaded Java code transparently in a serverless environment. CRUCIAL introduces two key differences with respect to multi-threaded Java: threads are replaced by *cloud threads*, and executor services by *serverless executor services*. CRUCIAL users can also use a set of shared objects using well-known Java syntax.

With FAASM, we have presented a stateful serverless runtime using a new lightweight isolation mechanism, *Faaslets*, based on WebAssembly. FAASM transparently supports two popular programming models for parallel scientific applications: MPI with FaasMPI, and OpenMP with FaasMP. By using FAASM scientists can deploy legacy MPI and OpenMP code in a serverless environment, without any hardware configuration nor any constraints on the number of parallel processes. In doing so, we fulfil the "lift and shift" nature of the project. We experimentally evaluated the scalability of FaasMP and FaasMPI, demonstrating that they satisfy the big data requirements of CloudButton.

In summary, the work presented in this deliverable takes us closer to a key goal of CloudButton, which is to make it easy for users to move from single machine code and traditional big data frameworks, to the cheap, flexible scalable deployments on serverless clouds. The requirement for ease-of-use is one of the key KPIs for the CloudButton project. We implement approaches based on familiar concepts such as multi-threading, multi-programming, MapReduce and object-oriented programming, as well as transparent execution of existing code using WebAssembly technology built with OpenMP and MPI.

References

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[2] Tensorflow, "TensorFlow Lite." `https://www.tensorflow.org/lite`, 2020.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM international symposium on high performance distributed computing*, pp. 810–818, 2010.

[5] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with mapreduce: a survey," *AcM sIGMoD Record*, vol. 40, no. 4, pp. 11–20, 2012.

[6] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," in *ACM Symposium on Cloud Computing (SOCC)*, 2017.

[7] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "Numpywren: Serverless Linear Algebra," *arXivpreprint arXiv:1810.09679*, 2018.

[8] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "A Case for Serverless Machine Learning," *Systems for ML*, 2018.

[9] B. Sang, P.-L. Roman, P. Eugster, H. Lu, S. Ravi, and G. Petri, "PLASMA: Programmable Elasticity for Stateful Cloud Computing Applications," in *ACM European Conference on Computer Systems (EuroSys)*, 2020.

[10] P. García-López, M. Sánchez-Artigas, S. Shillaker, P. Pietzuch, D. Breitgand, G. Vernik, P. Sutra, T. Tarrant, A. Juan-Ferrer, and G. París, *Trade-Offs and Challenges of Serverless Data Analytics*, pp. 41–61. Cham: Springer International Publishing, 2022.

[11] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "SEUSS: Skip Redundant Paths to Make Serverless Fast," in *ACM European Conference on Computer Systems (EuroSys)*, 2020.

[12] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[13] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," in *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[14] A. Klimovic, Y. Wang, S. University, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[15] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures," in *ACM/IFIP Middleware Conference*, 2019.

[16] V. Sreekanti, C. W. X. C. Lin, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful Functions-as-a-Service," *arXiv preprint arXiv:2001.04592*, 2020.

[17] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the Gap Between Serverless and its State with Storage Functions," in *ACM Symposium on Cloud Computing (SOCC)*, 2019.

[18] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, "Formal foundations of serverless computing," *ACM on Programming Languages (OOPSLA)*, 2019.

[19] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," *CoRR*, vol. abs/1702.04024, 2017.

[20] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the ibm cloud," in *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, (New York, NY, USA), p. 1–8, Association for Computing Machinery, 2018.

[21] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "numpywren: serverless linear algebra," masters thesis, EECS Department, University of California, Berkeley, Oct 2018.

[22] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "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)*, (Boston, MA), pp. 363–376, USENIX Association, Mar. 2017.

[23] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, (Boston, MA), pp. 193–206, USENIX Association, Feb. 2019.

[24] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 427–444, USENIX Association, Oct. 2018.

[25] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in anna," *Proc. VLDB Endow.*, vol. 12, p. 624–638, Feb. 2019.

[26] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," USENIX Association, 2020.

[27] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the faas track: Building stateful distributed applications with serverless architectures," in *Proceedings of the 20th International Middleware Conference*, Middleware '19, (New York, NY, USA), p. 41–54, Association for Computing Machinery, 2019.

[28] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," 2020.

[29] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A Large-Scale Study of MPI Usage in Open-Source HPC Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, Association for Computing Machinery, 2019.

[30] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf," *Procedia Computer Science*, 2015.

[31] H. Yviquel and G. Araujo, "The cloud as an openmp offloading device," 08 2017.

[32] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[33] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to Speed with WebAssembly," *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[34] "Knative platform," 2020.

[35] M. Shahrad, R. Fonseca, Íñigo Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," 2020.

[36] Amazon Web Services, "AWS Lambda." `https://aws.amazon.com/lambda/`, 2020.

[37] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[38] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, USENIX Association, 2020.

[39] Google, "Google Cloud Functions." `https://cloud.google.com/functions/`, 2020.

[40] S. Malik, "Azure Functions." `https://azure.microsoft.com/en-us/services/functions/`, 2020.

[41] IBM, "IBM Cloud Functions." `https://www.ibm.com/cloud/functions`, 2020.

[42] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "Serverless Computation with openLambda," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, USENIX Association, 2016.

[43] Amazon Web Services, "AWS Step Functions." `https://aws.amazon.com/step-functions`, 2022.

[44] Microsoft, "Azure Durable Functions." `https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview`, 2022.

[45] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the "Micro" Back in Microservice," *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[46] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A fault-tolerance shim for serverless computing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, Association for Computing Machinery, 2020.

[47] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[48] Amazon Web Services, "AWS S3." `https://aws.amazon.com/s3/`, 2020.

[49] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless Computing: One Step Forward, Two Steps Back," *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[50] Amazon, "AWS Step Functions." `https://aws.amazon.com/step-functions/`, 2020.

[51] Microsoft, "Azure Durable Functions." `https://docs.microsoft.com/en-us/azure/azure-functions/durable-functions-overview`, 2020.

[52] Apache Project, "Openwhisk Composer." `https://github.com/ibm-functions/composer`, 2020.

[53] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, 2020.

[54] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-service Computing," in *Annual International Symposium on Microarchitecture (MICRO)*, 2019.

[55] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[56] Google, "Google Cloud Functions." `https://cloud.google.com/functions`, 2021.

[57] S. Fouladi, R. S. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads," *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[58] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[59] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *ArXiv*, 2019.

[60] A. W. Services, "Lambdas," 2020.

[61] A. W. Services, "Aws s3," 2020.

[62] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, p. 51–59, June 2002.

[63] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash = local flash," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), p. 345–359, Association for Computing Machinery, 2017.

[64] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the gap between serverless and its state with storage functions," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2019.

[65] R. Labs, "Redis," 2020.

[66] M. Perron, R. C. Fernandez, D. DeWitt, and S. Madden, "Starling: A scalable query engine on cloud function services," 2019.

[67] Microsoft, "Azure serverless sql," 2020.

[68] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, (New York, NY, USA), p. 13–24, Association for Computing Machinery, 2019.

[69] A. Bhattacharjee, Y. D. Barve, S. Khare, S. Bao, A. Gokhale, and T. Damiano, "Stratum: A serverless framework for lifecycle management of machine learning based data analytics tasks," *ArXiv*, vol. abs/1904.01727, 2019.

[70] J. Carreira, "A case for serverless machine learning," 2018.

[71] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," *CoRR*, vol. abs/1712.05889, 2017.

[72] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: The prospect of serverless scientific computing and hpc," in *CARLA*, 2017.

[73] X. Niu, D. Kumanov, L.-H. Hung, W. Lloyd, and K. Y. Yeung, "Leveraging serverless computing to improve performance for sequence comparison," in *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB '19, (New York, NY, USA), p. 683–687, Association for Computing Machinery, 2019.

[74] M. Monfort, A. Andonian, B. Zhou, K. Ramakrishnan, S. A. Bargal, T. Yan, L. Brown, Q. Fan, D. Gutfruend, C. Vondrick, and A. Oliva, "Moments in time dataset: one million videos for event understanding," 2018.

[75] IBM, "Watson studio gallery." `https://dataplatform.cloud.ibm.com/gallery`.

[76] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless Data Analytics in the IBM Cloud," in *ACM/IFIP Middleware Conference*, 2018.

[77] Amazon Web Services, "Developer Guide for AWS Lambda," 2021.

[78] Amazon Web Services, "Optimize CPU options, Amazon EC2 User Guide for Linux Instances," 2021.

[79] R. Wang, J. Lehman, J. Clune, and K. Stanley, "Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions," *ArXiv*, vol. abs/1901.01753, 2019.

[80] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "OpenAI Baselines." `https://github.com/openai/baselines`, 2017.

[81] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," 2016.

[82] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. ACM*, vol. 17, p. 549–557, Oct. 1974.

[83] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is Lighter (and Safer) than your Container," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[84] M. Kerrisk, "Linux manual pages," 2020.

[85] Mozilla, "WASI: WebAssembly System Interface." `https://wasi.dev/`, 2020.

[86] Fastly, "Edge compute," 2020.

[87] C. Kenton Varda, "Fine-grained sandboxing with v8 isolates," 2020.

[88] Fastly, "Edge dictionaries," 2020.

[89] CloudFlare, "Worker kv," 2020.

[90] WebAssembly Community Group, "WebAssembly system interface."

[91] A. Scheidecker, "Wavm," 2020.

[92] L. Project, "Llvm 10 release notes," 2020.

[93] R. Felker, "musl libc," 2020.

[94] G. S. foundation, "The gnu c library (glibc)," 2020.

[95] P. S. Foundation, "Cpython," 2020.

[96] I. project, "Pyodide," 2020.

[97] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "RCV1: A New Benchmark Collection for Text Categorization Research," *Journal of Machine Learning Research*, 2004.

[98] Google, "KNative Github." `https://github.com/knative`, 2020.

[99] Kubernetes, "Kubernetes- Production-Grade Container Orchestration." `https://kubernetes.io/`, 2022.

[100] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *ACM European Conference on Computer Systems (EuroSys)*, 2013.

[101] Google, "Protocol buffers," 2020.

[102] I. Corporation, "Intel(r) math kernel library for deep neural networks (intel(r) mkl-dnn)," 2020.

[103] Intel, "Parallel kernels," 2020.

[104] O. A. R. Board, "Openmp api specification: Version 5.0," 2018.

[105] L. Project, "Llvm openmp runtime library. technical report," 2015.

[106] T. G. OpenMP and O. Implementation, "Gnu offloading and multi processing runtime lib," 2020.

[107] T. P. Group, "Pgi compiler openmp documentation," 2020.

[108] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf," *Procedia Computer Science*, vol. 53, pp. 121 – 130, 2015. INNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015.

[109] A. Basumallik, S.-J. Min, and R. Eigenmann, "Programming distributed memory sytems using openmp," pp. 1–8, 01 2007.

[110] M. Bull, "Measuring synchronisation and scheduling overheads in openmp," 02 2002.

[111] M. Ghane, A. M. Malik, B. Chapman, and A. Qawasmeh, "False sharing detection in openmp applications using ompt api," in *International Workshop on OpenMP*, pp. 102–114, Springer, 2015.

[112] O. Kwon, F. Jubair, and R. Eigenmann, "A hybrid approach of openmp for clusters," vol. 47, pp. 75–84, 09 2012.

[113] O. Kwon, F. Jubair, S.-J. Min, H. Bae, and R. Eigenmann, "Automatic scaling of openmp beyond shared memory," vol. 7146, 09 2011.

[114] J. P. Hoeflinger, "Extending openmp to clusters," *White Paper, Intel Corporation*, 2006.

[115] C. Terboven, D. a. Mey, D. Schmidl, and M. Wagner, "First experiences with intel cluster openmp," in *OpenMP in a New Era of Parallelism* (R. Eigenmann and B. R. de Supinski, eds.), (Berlin, Heidelberg), pp. 48–59, Springer Berlin Heidelberg, 2008.

[116] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "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)*, (San Jose, CA), pp. 15–28, USENIX, 2012.

[117] M. Mortatti, H. Yviquel, and G. Araujo, "Automatic ray-tracer cloud offloading in openmp," *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 428–435, 2018.

[118] L. Project, "LLVM openmp runtime library mirror repository."

[119] P. Kranenburg, "strace release notes," 2020.

[120] ParResKernels Team, "Parallel Research Kernels." `https://github.com/ParRes/Kernels`, 2021.

[121] Ian Karlin and Abhinav. Bhatele and Bradford L.. Chamberlain and Jonathan. Cohen and Zachary Devito and Maya Gokhale and Riyaz Haque, and Rich Hornung and Jeff Keasler and Dan Laney and Edward Luke and Scott Lloyd and Jim McGraw and Rob Neely and David Richards and Martin Schulz and Charle H. Still and Felix Wang and Daniel Wong, "Lulesh programming model and performance ports overview," *LLNL-TR-641973*, 2012.

[122] LLNL, "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," *LLNL-TR-490254*, 2005.

[123] A. Jangda, B. Powers, E. Berger, and A. Guha, "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code," in *USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[124] Linux Manual Page, "userfaultfd." `https://man7.org/linux/man-pages/man2/userfaultfd.2.html`, 2022.

[125] M. Weiland, "Chapel, fortress and x10: novel languages for hpc," *EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706*, 2007.

[126] L. V. Kale and S. Krishnan, "Charm++ a portable concurrent object oriented system based on c++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pp. 91–108, 1993.

[127] M. Merrill, W. Reus, and T. Neumann, "Arkouda: interactive data exploration backed by chapel," in *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, pp. 28–28, 2019.

[128] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, *et al.*, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[129] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[130] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 2135–2135, 2016.

[131] A. Gittens, K. Rothauge, S. Wang, M. W. Mahoney, J. Kottalam, L. Gerhardt, M. Ringenburg, and K. Maschhoff, "Alchemist: An apache spark mpi interface," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 16, p. e5026, 2019.

[132] I. Karlin, Y. Park, B. R. de Supinski, P. Wang, B. Still, D. Beckingsale, R. Blake, T. Chen, G. Cong, C. Costa, *et al.*, "Preparation and optimization of a diverse workload for a large-scale heterogeneous system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–17, 2019.

[133] LLVM Project, "LLVM 9 Release Notes." `https://releases.llvm.org/9.0.0/docs/ReleaseNotes.html`, 2020.

[134] S. Zhuang, Z. Li, D. Zhuo, S. Wang, E. Liang, R. Nishihara, P. Moritz, and I. Stoica, "Hoplite: Efficient and fault-tolerant collective communication for task-based distributed systems," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, Association for Computing Machinery, 2021.

[135] Facebook Incubator, "Gloo: Collective Communication Library with Various Primitives for Multi-Machine Training.." `https://github.com/facebookincubator/gloo`, 2020.

[136] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, no. 2, 2009.

[137] OpenMPI, "OpenMPI: Open Source High Performance Computing." `https://www.open-mpi.org/`, 2021.

[138] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, 1993.

[139] Sandia National Laboratories, "LAMMPS Molecular Dynamics Simulator." `https://lammps.sandia.gov/index.html`, 2020.

[140] Microsoft, "Azure: Cloud Computing Services." `https://azure.microsoft.com/en-us/`, 2021.

[141] The Knative Authors, "Knative - Enterprise-grade Serverless on your own terms.." `https://knative.dev/docs/`, 2021.

[142] Sandia National Laboratories, "Examples - LAMMPS Documentation." `https://docs.lammps.org/Examples.html`, 2022.

[143] Sandia National Laboratories, "Benchmarks - LAMMPS Documentation." `https://docs.lammps.org/Speed_bench.html`, 2022.