HORIZON 2020 FRAMEWORK PROGRAMME

CloudButton

(grant agreement No 825184)

Serverless Data Analytics Platform

D5.2 CloudButton Prototype of Abstractions, Fault-tolerance and Porting Tools

Due date of deliverable: 30-06-2020
Actual submission date: 31-07-2020

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	58
WP/Task related to this document	WP5 / T5.2
WP/Task responsible	Imperial
Leader	Peter Pietzuch (Imperial)
Technical Manager	Pedro García (URV)
Quality Manager	Pierre Sutra (IMT)
Author(s)	Simon Shillaker (Imperial), Mayeul Fournial (Imperial), Peter Pietzuch (Imperial), Daniel Barcelona (URV), Josep Sampé (URV), Pol Roca (URV)
Partner(s) Contributing	Imperial, URV
Document ID	CloudButton_D5.2_Public.pdf
Abstract	High-level programming models are essential to realising the potential of any distributed computing platform, and are commonplace in existing big data and machine learning systems. However, serverless computing lacks such abstractions, hence it is difficult for users to build new applications or port existing ones. To address this problem, we present several novel programming models in CloudButton, which adapt familiar big data, OS and HPC concepts to the serverless environment. Specifically these are: (1) MapReduce and multi-processing with the CloudButton Toolkit; (2) multi-threaded serverless programming with CRUCIAL; (3) FaasMP, automatic FaaS-ification of existing OpenMP code; 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

Table of Contents

1 Introduction **2**
1.1 Programming in CloudButton . . . 2
1.2 Serverless programming in context . . . 3
1.3 Overview of developed programming abstractions in CloudButton . . . 3

2 Background **5**
2.1 Challenges in current serverless platforms . . . 5
2.2 Serverless storage layers . . . 6
2.3 Serverless data analytics . . . 7

3 CloudButton Toolkit: Python APIs **8**
3.1 Map-Reduce API . . . 8
3.2 Python multiprocessing API . . . 9
3.2.1 An example: Deep learning video inference . . . 10

4 CRUCIAL: Serverless multi-threaded applications **13**
4.1 CRUCIAL programming model . . . 13
4.1.1 Execution abstractions . . . 13
4.1.2 State abstractions . . . 14
4.2 Sample applications . . . 14

5 FAASM: High-Performance Thread-Based Serverless **17**
5.1 FAASM and Serverless Big Data . . . 17
5.2 Faaslets . . . 17
5.3 Host interface . . . 18
5.4 Building FAASM functions . . . 20
5.5 State . . . 20
5.6 Scheduling . . . 21

6 FaasMP: Transparent use of OpenMP APIs with FAASM **22**
6.1 Background: Open Multi-Processing (OpenMP) . . . 22
6.1.1 OpenMP API . . . 22
6.1.2 Compiler code transformation . . . 23
6.1.3 Runtime library . . . 24
6.2 Related work on distributed OpenMP . . . 25
6.2.1 OpenMP to MPI translation . . . 25
6.2.2 OpenMP on software distributed shared memory (DSM) . . . 25
6.2.3 Offloading to the cloud . . . 26
6.3 FaasMP Design . . . 26
6.3.1 Platform requirements for shared memory multi-processing . . . 27
6.3.2 Challenges when distributing OpenMP . . . 28
6.3.3 Strawman design: compiling libomp.so to WebAssembly . . . 30
6.3.4 Design . . . 30
6.4 FaasMP architecture . . . 31
6.4.1 WebAssembly OpenMP runtime . . . 32
6.4.2 OpenMP toolchain . . . 33
6.5 Local library runtime implementation . . . 34
6.5.1 Forking with Wasm threads . . . 34
6.5.2 Loop support . . . 36
6.5.3 Threading and synchronisation support . . . 37

6.5.4 WebAssembly thread pool . . . 38
6.6 Experimental evaluation . . . 39
6.6.1 Linear algebra applications . . . 39
6.6.2 Local performance characteristics . . . 40
6.6.3 Distribution experiments . . . 41
6.6.4 Usability and potential . . . 43
6.6.5 Other performance considerations . . . 43

7 FaasMPI: Bridging the gap between HPC and the cloud 46
7.1 Motivating serverless MPI . . . 46
7.2 FaasMPI and FAASM . . . 46
7.3 FaasMPI architecture . . . 47
7.3.1 MPI one-sided memory access . . . 48

8 Distributed Data Objects: Object-oriented programming in FAASM 49
8.1 High-level state abstraction . . . 49
8.2 Two-tier state architecture . . . 50
8.3 Experimental evaluation . . . 50
8.3.1 Experimental set-up . . . 50
8.3.2 Experimental results . . . 51

9 Conclusion 52

Executive summary

Building distributed stateful applications is hard; users must manage coordination between workers and efficient distribution of data, while simultaneously scaling underlying system resources. Serverless computing provides an easy way to provision and scale resources, but writing applications for this environment is still challenging. This is down to the lack of a *high-level programming model*. 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 *multi-processing*, the ubiquitous *MapReduce* paradigm, and the two most popular HPC frameworks, *OpenMP* and *MPI*. In this deliverable, we describe the following work: (i) MapReduce with the CloudButton Toolkit; (ii) multi-processing with the CloudButton Toolkit to transparently port existing Python applications; (iii) serverless multi-threading in Java with CRUCIAL; (iv) WebAssembly programming in FAASM with Distributed Data Objects; (v) transparent parallel applications using OpenMP and FaasMP; and (vi) transparent execution of distributed applications using MPI and FaasMPI.

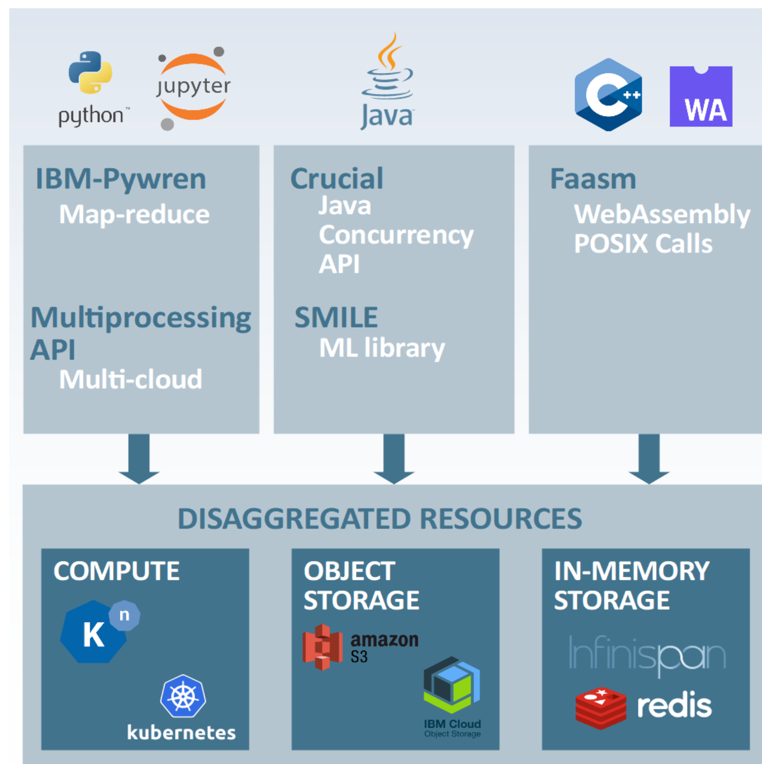


Figure 1: CloudButton architecture

1 Introduction

Much existing serverless work has covered the underlying runtime environment, focusing on its scaling and performance, but little attention has been paid to actually writing serverless applications. Writing these 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 provide a generic approach, and do not support transparently porting legacy applications. CloudButton aims to change this by creating a programming environment that makes it easy for users to build and scale serverless big data applications. In this deliverable, we report on our findings regarding a range of innovative programming models for serverless applications.

1.1 Programming in CloudButton

CloudButton aims to make serverless efficient for big data and easy to use, and hence providing appropriate programming models is key to the project's success. Figure 1 shows the CloudButton 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 all 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 the CloudButton Toolkit, CRUCIAL and FAASM. The CloudButton Toolkit 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 [10, 11, 12], storage performance [13, 14], state management [15, 16] 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. [17] 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 [18] demonstrated 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 [19, 20, 21, 22] or tried to provide a new storage layer [23, 24] to circumvent the limitations of the platforms. This lead to the development of stateful efficient serverless solutions [25, 26, 27] which present efficient yet scalable data processing applications.

1.3 Overview of developed programming abstractions in CloudButton

Next 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:

CloudButton Toolkit. One core principle behind CloudButton project is programming simplicity. Our focus is to make serverless computing as usable as possible, irrespective of whether the programmer is a cloud expert or not. We have devoted efforts to integrate Python CloudButton toolkit with other tools (e.g. Python notebooks such as Jupyter), which are very popular environments for the scientific community.

To simplify the serverless transition of existing multithreaded codebases, we present two different Python APIs: one based on Map-Reduce calls and another based on standard Python APIs such as `multiprocessing` and `concurrent.futures`.

CRUCIAL. CRUCIAL [14] 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.

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. Additionally, 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.

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 [28, 29, 30, 31]. 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* [25], which provides security and resource isolation using WebAssembly [32] 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.

FAASM is designed to be a pluggable runtime that integrates with existing serverless platforms. In CloudButton we use FAASM's Knative [33] 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 [34]; the users—provided the development cost is not too high—benefit from fine-grain billing and the absence of operational management costs.

Popular commercial platforms can be split into two categories: scalable container orchestrators such as AWS Lambda [35] and software fault isolation engines running small stateless functions, suitable for edge computing on a network [36, 37]. They are both elastic compute layers which when utilised properly can fulfil the same compute requirements at a lower cost than traditional IaaS methods thanks to the fine-grain scaling and billing which helps avoid over-provisioning. Such careful tuning implies that all platforms require custom code, not only in the form but also in the algorithms which have to deal with the platforms somewhat arbitrary limitations (e.g. 15 minutes compute limit for AWS lambdas).

Before we propose new programming models for FaaS, this section explores the issues with current commercial FaaS platforms (§2.1) and their storage layers (§2.2), which limit the ease with which applications can be built on top of them (§2.3).

2.1 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 [38]. 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. The main performance reduction comes from the function isolation and their physical dissociation from storage [39, 40, 38]. Functions must pull data from cloud storage services over the network, and the associated performance degradation is worsened by the following properties, which prevent common optimisations:

1. Low network bandwidth inside functions.
2. Lack of function-to-function communication.
3. Lack of fast and scalable shared state.
4. Functions are short-lived and not reusable.

(2) Cold start latency Many serverless platforms are known for exhibiting high latencies (in the order of seconds) [41] before reaching the first line of user code. This is far from the expected programming model of a "function". There are two main causes for latency:

1. POSIX virtualisation/initialisation of containers; and
2. dynamic language runtime and library loading at function start.

(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.

2.2 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[1] just the shuffle phase; compared to 50 minutes & $144 for Spark to complete entirely [22]. 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 [35, 42]. This is largely insufficient to complete the task in a timeline fashion[2]. In this case, Locus [22] 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 [43] 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 [18]. Examples of such solutions include:

- *Pocket* [23] 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 [44] 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* [27] 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.

- *Shredder* [45] 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.

[1] 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 $\times \$0.0000095 = \$23,750$.

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

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 [46]) and rate-limited but cheap storage (e.g. S3 [42]) [22] for their map-reduce operations.

Some SQL-compatible storage platforms refer to themselves as *serverless*, either because they are themselves running on serverless platforms [47], or because they offer scaling to zero and fine-grained billing associated with the underlying DBMS [48]. 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 citeInfinispan provides a multi-purpose distributed in-memory store that can be used to implement distributed state for serverelss functions, as demonstrated in CRUCIAL.

2.3 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 [18, 19]. After initially struggling with network efficiency, more recent work such as numpywren [20], 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 [49, 22, 21]. 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 [50], while others provide more general frameworks to coordinate serverless machine learning (ML) [49, 51]. 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) [52]. 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 [53, 54]. 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.

3 CloudButton Toolkit: Python APIs

The Python CloudButton toolkit exposes different APIs that can be used based on user requirements. In D5.1, we presented a first API definition based on map-reduce. Now, the flexibility of the Cloudbutton toolkit 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 the Python Cloudbutton Toolkit, and it integrates the basic, low-level methods to spawn functions in the cloud. The primary object in the Map-Reduce API of the Cloudbutton toolkit is the **FunctionExecutor**. 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 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.

Listing 1: Cloudbutton toolkit 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())
```

3.2 Python multiprocessing API

The Cloudbutton toolkit supports most of the Python multiprocessing abstractions, such as the Process, Pool, Queue, Pipe, Lock, Semaphore, Event, Barrier, and also remote memory in Manager objects. These shared components across processes are transparently supported by using a Redis deployment. Thus, a common user that knows how to program with these components can already program the Cloud. On the other hand, the Cloudbutton toolkit also enables transparent access to the storage and memory in the cloud. This means any application can be executed both locally or in any supported cloud without modification. Consequently, any current multiprocessing-based application can be moved and scaled to the cloud by only changing the import statement of the script. See the next example that calculates Pi both locally using multiprocessing or transparently in the Cloud:

Listing 2: Cloudbutton toolkit example using the Python multiprocessing API

```
# from multiprocessing import Pool
from cloudbutton.multiprocessing import Pool
import random

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

np, n = 4, 10000000
part_count = [int(n/np)] * np
pool = Pool(processes=np)
count = pool.map(is_in, part_count)
pi = sum(count)/n*4
print("Esitmated Pi: {}".format(pi))
```

The objects supported by the CloudButton toolkit in the multiprocessing API are described in Table 1. The primary object in the multiprocessing API of the CloudButton toolkit is the **Pool**. Once you get an instance of the Pool, you can spawn functions to the Cloud with the next API methods:

▷ `apply()`: Call func with arguments args and keyword arguments kwds. It blocks until the result is ready. Given this blocks, apply_async() is better suited for performing work in parallel. Additionally, func is only executed in one of the workers of the pool.

▷ `apply_async()`: A variant of the apply() method which returns a result object instead of the result itself.

▷ `map()`. A parallel equivalent of the map() built-in function (it supports only one iterable argument though). It blocks until the result is ready. This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting chunksize to a positive integer.

▷ `map_async()`: A variant of the map() method which returns a result object instead of the result itself.

Table 1: Python multiprocessing API components

CATEGORY	API	DESCRIPTION
Object	Pool, Manager	*Pool and Manager object*
Connection	Pipe	*A pipe which by default is duplex (two-way)*
Queues	Queue, SimpleQueue, JoinableQueue	*Remote queues*
Data structure	Value, Array	*Generic objects*
Synchronization	Lock, RLock, Semaphore, BoundedSemaphore, Condition, Event, Barrier	*Objects for distributed worker-to-worker coordination*

▷ `imap()`: A lazier version of map().

▷ `imap_unordered()`: The same as imap() except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be "correct".).

▷ `starmap()`: Like map() except that the elements of the iterable are expected to be iterables that are unpacked as arguments. Hence an iterable of [(1,2), (3, 4)] results in [func(1,2), func(3,4)].

▷ `starmap_async()`: A combination of starmap() and map_async() that iterates over iterable of iterables and calls func with the iterables unpacked. Returns a result object.

3.2.1 An example: Deep learning video inference

This example uses the Cloudbutton Toolkit to process videos from the Moments in Time video dataset [55]. It predicts actions that appear in videos using a pretrained ResNet50 deep neural network model. By means of the toolkit's proxy API, this code is structured in a way that it can be executed in a remote environment where tasks are executed by serverless functions that load data from Cloud storage, as well as in a local environment where processes execute those tasks and load data from shared memory as a storage. The storage is used to load input video files and the serialized model for inference.

Listing 3: Importing of equivalent API's

```
LOCAL_EXEC = False
INPUT_DATA_DIR = 'momentsintime/input_data'

if LOCAL_EXEC:
    import os
    pool_initargs = {
        'compute_backend': 'localhost',
        'storage_backend': 'localhost'
        }
    weights_location = '/dev/shm/model_weights'
    INPUT_DATA_DIR = os.path.abspath(INPUT_DATA_DIR)

else:
    from cloudbutton.cloud_proxy import os, open
```

```
    pool_initargs = {
        'compute_backend': 'ibm_cf',
        'storage_backend': 'ibm_cos',
        'runtime': 'dhak/pywren-runtime-pytorch:3.6',
        'runtime_memory': 2048
        }
    weights_location = 'momentsintime/models/model_weights'
```

After the type of the execution has been decided (local or remote) in Listing 3, a change in the import of the standard Python library to the proxy version from Cloudbutton Toolkit enables accessing 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.

Listing 4: Loading of model to the storage

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

os.system('wget ' + '/'.join([ROOT_URL, WEIGHTS_FILE]))

with builtins.open(WEIGHTS_FILE, 'rb') as f_in:
    weights = f_in.read()

with open(weights_location, 'wb') as f_out:
    f_out.write(weights)
```

As the first step, the model is pulled from the respective repository and then loaded to the storage. This step is necessary since the model has to be placed in the storage for every function to be able to access it. There is a little subtlety in this code which is the two accesses to the files. In the first call, the native `builtins.open` method is used to force the opening of a local file which is the serialized model that just has been downloaded. In the second call, however, it is intended that the call to `open` may store the model to shared memory or to Cloud storage.

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

```
    queue.put(results)
```

The video prediction function in Listing 5 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. This reduce 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:

Listing 6: Reduce function

```
def reduce_predictions(queue, n):
    pred_x_categ = {}
    for categ in categories:
        pred_x_categ[categ] = 0

    for i in range(n):
        results = queue.get()
        res_count += len(results)
        for res in results:
            idx, prob = res['prediction']
            pred_x_categ[categories[idx]] += 1

    return pred_x_categ
```

Listing 7: Main function

```
CONCURRENCY = 1000

def main():
    queue = Queue()
    pool = Pool(initargs=pool_initargs)

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

    pool.map_async(func=predict_videos, iterable=iterable)
    pred_x_categ = reduce_predictions(queue, N)
    print(pred_x_categ)
```

Finally, Listing 7 contains the main function code. First, the list of paths or keys of the input videos is obtained. Then, that list is split among `N` parts matching the desired concurrency, and thus, each function may end up processing multiple videos. It is then when the `call_async` function is called to map the lists of keys with the prediction function. 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 parallelised thanks to the Cloudbutton Toolkit, because there are no dependencies or communication between functions and the use of serverless backends allows for massive scaling.

ABSTRACTION	DESCRIPTION
`CloudThread`	Serverless functions are invoked like threads.
`ServerlessExecutorService`	Groups of tasks are managed with a simple executor service.
Shared objects	Linearizable (wait-free) distributed objects (e.g., `AtomicInt`, `List`, `Map`, . . .).
Synchronization objects	Shared objects for thread synchronization primitives (e.g., `CyclicBarrier`, `Semaphore`, `Future`).
`@Shared`	User-defined shared object. Methods are run on the DSO servers, allowing fine-grained updates and aggregates (`.add()`, `.update()`, `.merge()`, . . .).
Data persistence	Long-lived shared objects are replicated. Use `@Shared(persistence=true)` to activate it.

Table 2: 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 2.

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 8 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 9.

An application that outputs an image approximating the Mandelbrot set with a gradient of colours is shown in Listing 10. 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 8: 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 9: 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 11 shows a more complex application that uses synchronization primitives like a barrier.

Listing 10: 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();
    }
}
```

Listing 11: 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());
    }
}
```

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 [41, 56]. 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 [57].

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 [42] 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 [38]. 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 [32]. 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* [58] and network access is limited using *network namespaces* [58] 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 2. 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 [33, 59]. 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

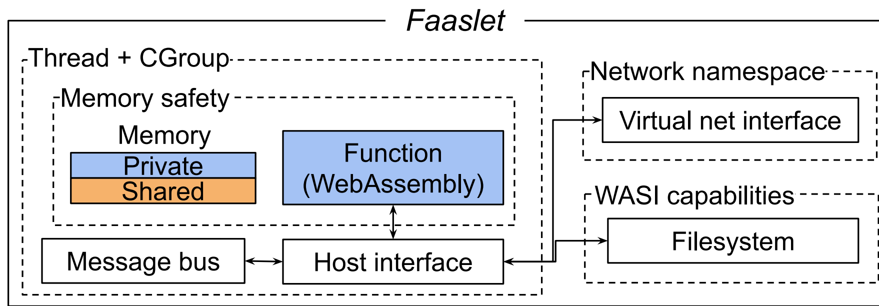


Figure 2: Faaslet isolation in FAASM

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 [58].

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 (§2.1), 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 3. 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 (§5.5) 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

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 3: FAASM host interface (The final column indicates whether functions are defined as part of POSIX or WASI [60].)

platforms such as Fastly [37] or CloudFlare [36], which also use WebAssembly, can only share state through external distributed key-value stores for their workers [61, 62].

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) [63] 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.

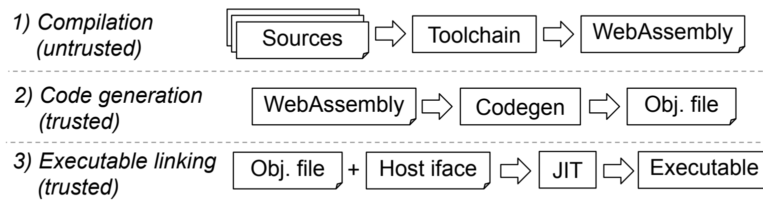


Figure 3: Creation of a Faaslet executable

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 3 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 [32]. FAASM relies on a trustworthy open-source WebAssembly *embedder*, WAVM [64] and LLVM-JIT libraries [65], 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 [65] (i.e., Clang, compiler-rt, libcpp, etc.), which was built to cross-compile to WebAssembly. The shipped libc is musl [66], a small and fast libc implementation compared to the traditional glibc [67]. 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 [68, 69] and import them in user programs.

5.5 State

The FAASM state is maintained by a key-value (KV) store of byte arrays, which can hold arbitrary data without synchronisation overheads. It is accessed by the user through a two-tier approach. Each level can be locked independently depending on the consistency model required by the user's application.

Global state: Any distributed KV store, which in our prototypes is Redis [46]. Faaslets can `push` local changes to write to the state or `pull` to update the local value. An implementation of a simple distributed lock with timeout can be used by Faaslets to synchronise.

Local state: The local replica of the state value is mapped to shared-memory regions of the co-located Faaslets. A local read-write lock can be used to synchronise access to the local state.

FAASM is agnostic as to the specific KVS used for its global state tier. In CloudButton we use Redis provided by the shared disaggregated resource layer as shown in Figure 1.

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 [70]. The metadata of the Faaslet runtime is serialised in a protocol buffer message [71] 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 [72], linear algebra [73] and big data [29]. 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 [74]. It is supported by every major C/C++ compiler [75, 76] and, due to its popularity in the HPC community, it is also implemented in multiple scientific compilers [77]. 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 [74]. 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 [78]. Existing work has attempted to remove this dependence on MPI by both adding distribution to OpenMP itself (§6.2), converting OpenMP to MPI [79], and offloading OpenMP to the cloud [30].

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 4, 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 4: Overview of main OpenMP pragmas

```
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 12: Racy thread-count.c OpenMP example

3. The runtime behaviour gives **predictable performance** [80]. 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 12 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 13 what the compiler's internal representation of the program looks like if we could convert it back into C++ after the compiler applied the OpenMP

```
static void parallel_section(int *number_of_threads)
{
    *number_of_threads++;
}

int main()
{
    int number_of_threads = 0;
    __kmpc_runtime_fork(parallel_section, &number_of_threads);
    return number_of_threads;
}
```

Listing 13: Output of the compiler transformation (simplified)

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 [75], is responsible for running the `parallel_section` function on each available OpenMP threads.

Compiling the same `count-unsafe.c` program (Listing 12) 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 [81] 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 12), no synchronisation mechanism was used for `number_of_threads`, which means that our implementation was racy.[3] Several low-level concurrency primitives are available for threads to synchronise (§6.1.1). Listing 14 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` [76] or Clang/LLVM and `libomp` [75]. 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.

[3] 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 14: 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; (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 [82, 79, 83]. 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 (DSM)

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 [84], 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) [84] 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 [85]. 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 [85], 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 distributed state operates at a byte-level granularity rather than a page-level granularity, hence does not face this problem. Each state value is represented as an arbitrary array of bytes which is transferred over the network when needed, and placed in a fixed region of a host's local memory.

`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 will incorporate this in our design by using the fine-grain control of the FAASM local and global state consistency to only enforce appropriate levels of synchronisation[25].

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** [85].

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 [86] for Map-Reduce jobs [30]. This can be used in applications like edge-compute for mobile devices [87]. 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.

6.3 FaasMP Design

Next we describe FaasMP, an implementation of OpenMP built into FAASM, which takes unmodified OpenMP code and automatically executes it as serverless functions. This is done by intercepting calls to the OpenMP runtime library, and converting tasks to serverless functions on-the-fly.

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.

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 5: WebAssembly and serverless compatibility for `libomp` system calls

6.3.1 Platform 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 [88]. 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 [89]—with a filter to only show activity from `libomp`. Table 5 shows the observed system calls and groups them into categories. The system calls are ticked if there is a reasonable equivalent implementation 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 [25, 26] 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 [58]. Although with Crucial [26] (see §4), we also explore the use of AWS Lambda [35] as a "cloud thread" executor, the lack of shared memory in such cases is an issue when providing efficient threads.

Conversely, Faaslets [25] 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 [58]. The `futex` interface has strong performance guarantees, and the `robust_list` has strong safety guarantees. The CAP theorem [43] 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.

6.3.2 Challenges when distributing OpenMP

Serverless and FaaS requires a fundamentally distributed programming model. Therefore, after studying the local requirements above, we consider previous attempts at distributing OpenMP to share work across many nodes, namely paged-based software DSM and MPI (§6.2). Table 6 summarises the desirable features (or lack thereof) that the related work found useful (or missing) from their mechanism of choice. We contrast those within the FAASM serverless environment which presents a unique combination of features compared to previous approaches.

Note that DSM and Faaslets do not have to add complex source-to-source translation and dataflow analysis to the OpenMP compilation process to apply their distribution mechanisms [90, 84] whereas the MPI approach needs to first transform OpenMP to MPI [82]. MPI also emphasises data immutability, which is in direct contrast with how existing OpenMP applications are written, i.e, they use shared mutable memory and critical sections to synchronise manipulation of shared data.

Network efficiency. MPI is the most network efficient mechanism when carefully programmed for. By emphasising immutability, it requires only a single network connection for updating synchronised remote objects. DSM must modify the page with the lock on it and the page with the data, and Faaslets in the general case have to perform multiple network connections to update the global state: locking the global state, retrieving the data, transferring the data after update, and unlocking the global state.

Faaslets are, however, like MPI, capable of only putting on the wire small objects and not whole pages of page diffs like in DSM. FAASM cannot aggregate communications like MPI [79]; `libMPNode` [90] can implement OpenMP concepts hierarchically (e.g. operate a local reduction first then open a single network communication per node to aggregate the results).

Category	Property	DSM	MPI	FAASM
Paradigm similarities	Does not require additional compilation steps other than OpenMP transformations to be in distributed	✓	✗	✓
	Pointer-level access to mutable shared memory	✓	✗	✓
Network efficiency	Single overhead for synchronisation and data transfer	✗	✓	✗
	Network-efficient aggregation of communications	✓	✓	✗
	Smallest possible bytes on the wire for shared data	✗	✓	✓
Consistency	Local and global control of state consistency	✗	✓	✓
Remote memory access	Guaranteed progress (no thrashing) when accessing contentious remote objects	✗	✓	✗
	Fine-grain control on state placement on the nodes	✗	✓	✓
	Transparent remote data pre-fetching	✓	✗	✗
Fault tolerance	Data replication	✓	✗	✗
	OpenMP program carries on after node failure	✗	✗	**NA**
Multi-tenancy	Scale in and efficient resource utilisation	✗	✗	✓
	Tenant isolation on shared boxes	✗	✗	✓

Table 6: Distributed OpenMP feature map

Consistency. Control over global and local consistency in MPI and FAASM avoids needless memory waits caused by enforcing sequential consistency for all DSM operations when OpenMP allows for relaxed memory ordering on many operations. Indeed even if a node possesses the latest version of a page, the DSM must perform a remote page invalidation before allowing the threads to write to the page. OpenMP programs only need to perform such remote protocols around synchronisation points and as such do not need the entirety of their state to be consistent.

Remote memory access. DSM systems are prone to trashing when two nodes try to access the same remote page, which can be especially problematic in OpenMP when two popular locks are put on the same page. This forced previous approaches such as `libMPNode` [90] to allocate distinct data on different pages.

DSM tends to be mostly transparent to the user compared to MPI, and they thus do no share the same optimisation possibilities. For example, MPI users can pull only the required data on each node, while DSM users can have a runtime system to monitor and prefetch pages based on recurring access patterns [82].

Fault tolerance. DSM allows for page replication but requires the program to be tolerant to node failures for which OpenMP is not a suitable programming model. Previous work would completely abort programs upon node failure. Although MPI programs can exhibit some fault tolerance, no OpenMP to MPI approach can make any recovery guarantee on node failure or network partition.

Multi-tenancy and isolation. We reach the same conclusion as for the local platform requirements (§6.3.1): the integration of an OpenMP runtime in a dynamic shared environment is challenging, and this must not hinder the isolation guarantees, low resource-footprint and scalability of the FAASM layer. These are primary requirements placed on our implementation and design that must be prioritised for our contributions to FAASM to be accepted.

6.3.3 Strawman design: compiling libomp.so to WebAssembly

Given the above requirements and challenges, we can exclude a design that is commonly used in related work and that was originally suggested for this project. It consists of compiling Clang `libomp` for the FAASM platform and provide the following modifications: (i) use Faaslets instead of threads as the parallel mechanism; and (ii) secure the library to be used in a possibly adversarial environment and ensure fair resource access.

Previous approaches to distributed OpenMP worked by extending an existing compiler and its associated runtime (§6.2). For example, Cluster OMP [84] is part of the Intel compiler package, whereas `libMPNode` [90] chooses to adapt GNU's `libgomp` [76] library after describing LLVM's `libomp` [88] library as much more complex because of its multi-architecture and multi-platform support. However, our choice is dictated by other factors than just ease of implementation. The GNU C/C++ compiler does not support WebAssembly, but LLVM does and is already in the FAASM toolchain. It is thus in principle possible to compile the OpenMP runtime library of Clang to WebAssembly and adapt the platform to support the necessary system calls and libc mechanisms mentioned previously in Table 5 to run in a distributed fashion. This holistic approach is also taken by FAASM to extend the host interface to support specific system calls needed by language runtimes. We outline the required work for this design:

1. implement `clone` in FAASM to spawn Faaslets;
2. compile `libomp` to WebAssembly and make multi-tenancy changes in the runtime; and
3. implement `futex` to provide lightweight synchronisation for Faaslets.

This approach may be more comprehensive than our design (§6.3.4), but the steps listed above are non-trivial, dependent on each other's success, and require change to three complex codebases (FAASM, Clang++ and `libomp` [91, 65, 88]). There are two other blockers for this approach: (1) as explained in §6.3.1, the necessary systems calls do not fit a WASI interface, and even less in a serverless host interface; (2) operating at the system call level does not give enough information to include the necessary performance optimisations for distributing OpenMP. Previous approaches [82, 79] use techniques such as inserting tracing code at compile time to detect patterns in remote data accesses, and employ careful thread placement and message aggregation to optimise the application runtime. The current FAASM scheduler and state do not have mechanisms to use this information to perform optimisations (e.g. physical co-location of data and Faaslets or pre-fetching). This design would therefore suffer from the issues other runtimes have eliminated unless extra work is put into changing FAASM distribution mechanisms.

6.3.4 Design

One of the goals of the CloudButton project is to **allow existing OpenMP programs to run in a serverless environment**. Let us assume it is possible to apply the OpenMP compiler transformations to the original application and compile the result to WebAssembly. Since the behaviour of the OpenMP transformations is dictated by the compiler's runtime library, we can replace the compiler's library with our own, `libfaasmp`. Our library then leverages WebAssembly and Faaslets to provide the required isolation guarantees and to distribute the OpenMP threads.

Re-implementing the library allows us to satisfy the serverless isolation and multi-tenancy requirements. We implement two execution contexts: (i) a local one that seeks to match the native local library performance of the compiler's library; (ii) a distributed context that uses Faaslets to scale the application. This design prescribes two required components to fulfil our objective of seamlessly supporting existing code:

1. A cross-target build system to compiling existing code to both native binaries and Wasm modules after applying OpenMP transformations. This should be integrated with the first step of the FAASM build process (§5.4) and be highly usable.

Positives	Negatives
Incremental support of the OpenMP API	Only supports OpenMP
No runtime analysis required	Implementation tied to Clang's internal API
Individually optimisable OpenMP concepts	Feasibility assumptions
Less radical design commitments from FAASM	

Table 7: Critical analysis of the design

2. A Wasm OpenMP runtime library using faaslets for running OpenMP threads implementing the main public and private OpenMP runtime library functions. Those functions are mostly independent from each other and can be implemented one at a time.

Table 7 summarises the pros and cons of this design. This is not a holistic approach since we only aim to support OpenMP threads, and the implementation will be tied to the compiler's private library API. However, compared to its main competing design (see §6.3.3), our design allows for implementing and optimising OpenMP pragmas one at a time, instead of focusing on the implementation of low-level concepts that will likely fail to scale (e.g. `futex` have to support all three of distributed locks, barriers and reduction sections). We rely on the compiler doing the heavy work to prepare the code, and do not require to modify the compiler to insert additional runtime analysis clues for optimisation which will save implementation time for this project. This design also integrates more nicely with FAASM by not having to commit part of the host interface to the OpenMP design but is instead an optional additional feature which helps with our objective of contributing back to the project.

6.4 FaasMP architecture

In this section, we present the integration with FAASM of the design that we explained. Figure 4 describes what steps have to be taken in order to start executing a parallel section. The start of a parallel section is marked by an `#omp parallel` pragma that gets compiled by Clang to the `libomp` function `__kmpc_fork_call` with arguments: (i) `function`, a pointer to the function the OpenMP threads will be executing, (ii) `args`, a list parameters—mainly shared variables—to call the `function` with, (see §6.1.2 for the technical details). The timeline of the forking process is thus:

1. A thread calls `__kmpc_fork_call(`*`function`*`, `*`args`*`)` as described in §6.1.1.
2. The translation layer: (i) looks up *`function`* in the WebAssembly module function table (<u>`function`</u>) and (ii) looks up *`args`* in the WebAssembly address space (<u>`args`</u>).
3. Threads are created using one of two mechanisms:
   - When using Wasm threads inside a Faaslet, TLS is set for the current parallel section, and the function is executed in the user's thread pool.
   - When running in new Faaslets: (i) the execution state is snapshotted to be restored on remote hosts; (ii) the Faaslets are scheduled to run <u>`function`</u>(<u>`args`</u>) and given the metadata about the current parallel section to set the runners TLS.

The translation layer holds the compiler-specific OpenMP library functions and defines a boundary that limits the impact of compiler changes in our implementation. We use WAVM intrinsic functions to define the compiler private API function to be able to manipulate the function table and the memory. This layer is therefore tied to both the compiler and the Wasm embedder executing the code. The latter is less of an issue than for Clang's internal API, because we only use the public

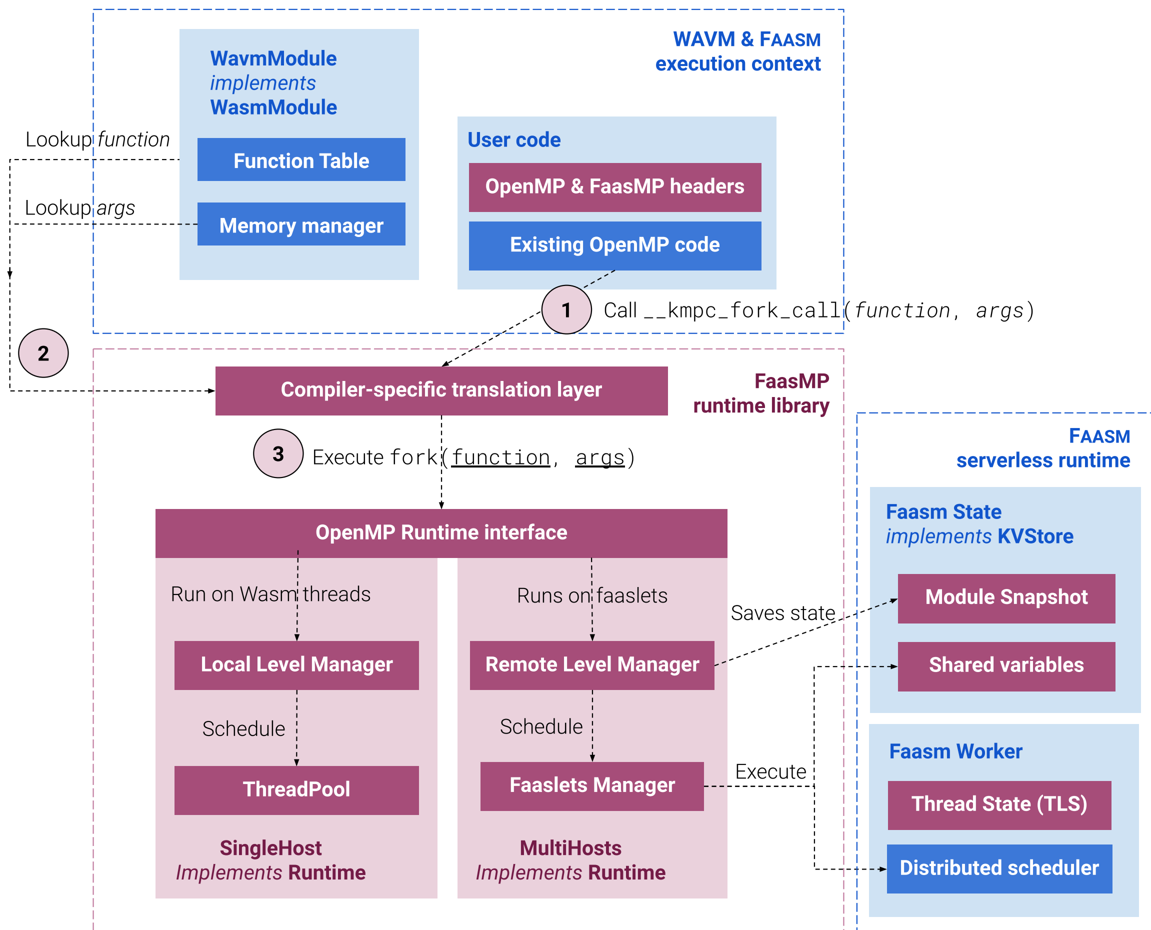


Figure 4: Architectural overview of the forking process (Purple is for FaasMP).

API of the WAVM library. The functions are exported through normal WebAssembly import/export mechanisms.

The architecture allows to switch between execution modes seamlessly and also can handle recursion. The user code does not need to be recompiled to switch between the two, and the choice of library backend is simply a configuration choice that can vary at runtime and even be different between nested sections. The caller invoking a fork can thus be any thread of execution, OpenMP or not.

6.4.1 WebAssembly OpenMP runtime

We implement a multi-tenant library inside FAASM that uses local WebAssembly threads as the executors for OpenMP threads inside a Faaslet. We ensure that our work is usable by existing applications and then explain how we built a high-performance local OpenMP library in a multi-tenant environment (§6.5).

It is unknown how to apply OpenMP transformations to the user code and compile the result to WebAssembly. Multiple issues can arise when trying to do so, for example the generated code by the compiler may depend on the binary format of the executable, but WebAssembly modules are not compatible with the traditional ELF-format; or it may be necessary to change the code generation in order to use Faaslets as threads, even though we aim to not have to modify the compiler.

Transformation and distribution. We argue the transformed OpenMP code is in a suitable state to be distributed following the compiler pass. As shown in §6.1.2, after the compiler applied the OpenMP transformations, the original OpenMP program is free from OpenMP-specific constructs and (i) the parallel code is extracted to separate functions, (ii) the shared variables are clearly separated as arguments to those functions, and (iii) the OpenMP pragmas are translated into calls to well-defined runtime library functions [75]. As such, if there existed a compiler that could apply the OpenMP transformations to a program and cross-compile the result to WebAssembly, we would have the necessary symbols, shared variable control and hooks into the code to be able to implement the distributed runtime library.

Novel use of LLVM. It is not necessary for the transformation and compilation processes to be handled by a unique compiler, the OpenMP transformations are typically applied on a compiler's internal representation, but those can sometimes be ported to other compilers IRs. Clang [65] is the only single compiler that can do both steps and conveniently is already part of the FAASM toolchain. To our understanding, we are the first to employ the `-fopenmp` and `-target=wasm32` flags together, which is a stable combination of flags only for recent LLVM releases[4].

The result of our prototype showed that with a suitable modification of the compiler in the FAASM toolchain, we can compile OpenMP programs such as the ones in Listing 14. Using tools like `wasm-objdump`[92] that can read WebAssembly symbols in Wasm modules, we also verify that all the expected private library symbols are available and what we expect based on the `libomp` compiler reference [75].

6.4.2 OpenMP toolchain

We must ensure that it is seamless for users to cross-compile their existing OpenMP application to use our runtime. FAASM already integrates with non OpenMP applications by shipping a sophisticated C/C++ WebAssembly toolchain as well as a library providing helper features for the user code (e.g. `VectorAsync` to push elements to the state [25]). We modify the Clang configuration in the toolchain to include the OpenMP features required for the shipped compiler to support the OpenMP code transformations and provide a static library released with FAASM in the toolchain system root.

Static library. We add a new library, `faasmp`, to the libraries distributed by FAASM to handle OpenMP symbol resolution at compile time and provide helper and debugging features.

There are two types of function declaration that the compiler needs to be aware of. First, OpenMP prescribes that the public runtime functions are declared in the `omp.h` header [74]. Since we do not modify the OpenMP API as per our objectives, we simply include the LLVM's standard OpenMP header to the toolchain. That header contains all the declarations up to the version 5.0 of OpenMP and useful extensions that existing OpenMP programs might rely on so providing them helps with compatibility. Second, the private runtime library calls are injected into the AST and thus do not require additional changes to the toolchain.

The Wasm linker needs to be aware of the compiler's public and private library symbols found in the IR. LLVM's `-fopenmp` flag does not only instruct the compiler to apply the OpenMP transformations but also allows for the dynamic linking of the compiler's shared library `libomp.so` that provides those symbols. There is no WebAssembly mechanism hitherto for the linker to identify symbols that will be available at runtime in the same way native dynamic compilation allows for. Dynamic WebAssembly linkers however provide a method to import global function symbols from static libraries. The unresolved symbols are placed in a `faasmp.import` file along with the static library `faasmp.a`. The linker uses the common file basename to match the two and add all the functions listed in the import file to the final Wasm module as WebAssembly import functions.

Unfortunately, Clang adds data symbols to the transformation to be ABI compatible with the GNU library and they were not documented in the `libomp` reference [75]. Resolving those symbols in WebAssembly is a non-trivial exercise. Those symbols identifier cannot be resolved using the

[4]Support for Wasm started in LLVM-7 but was only stable from 8.0.0 onwards, although OpenMP's `libtarget` is crashing for this version. We have had issues with LLVM 9.0.0, crashing LLVM's front-end when compiling certain OpenMP programs to Wasm, but not with LLVM 9.0.1 and 10.0.0 so far when using our modified Faasm toolchain.

import file technique because their name is not a valid identifier and because they are data symbols of unknown sizes to the linker. Since it is not possible to prevent the compiler to add those symbols during the transformation, we add definition for them in an assembly file that we integrate to our static library archive, such that the user does not have to do additional linking to support those symbols. Let us consider the example code given in Figure 6.4.2 that contains a `reduce` clause that causes the generation of such a symbol symbols. Listing 15 shows the corresponding assembly which is interpreted by the Wasm linker which is similar to how LLVM resolves those symbols [88].

```
    .hidden  .gomp_critical_user_.reduction.var
    .type    .gomp_critical_user_.reduction.var,@object
    .section .data..gomp_critical_user_.reduction.var,"",@
    .globl   .gomp_critical_user_.reduction.var
    .p2align  4
.gomp_critical_user_.reduction.var:
    .int64 42
    .size  .gomp_critical_user_.reduction.var, 8
```

Listing 15: GNU ABI LLVM compatibility symbol for a reduce variable

Compilation example. Figure 5 shows how the OpenMP symbols resolution works when targeting WebAssembly in the first step of the FAASM compilation process. The code contains a parallel section with a `reduce` clause, and makes use of the runtime library function `omp_set_num_threads` to set the desired number of threads for the next parallel section. The `faasmp` library header files are included in the usual way before the OpenMP transformations are applied to the code to declare public library functions and helpers. We will later use the early inclusion of the headers in the compilation process to replace the program reduction types with ones backed up by the FAASM state when distributing the application. The linker then finds the FAASM libraries, including `faasmp`, and lists all the public and private OpenMP functions from the `faasmp.imports` file as import functions in the final WebAssembly module. The GNU compatibility symbols are added during the OpenMP transformation compilation stage and are resolved during the linking stage by the `z_Linux_asm.S` assembly file.

6.5 Local library runtime implementation

In this section, we describe our implementation of a high-performance WebAssembly OpenMP runtime, which also implements the necessary isolation and multi-tenancy requirements for its integration into a serverless environment. Our runtime is capable of efficiently running multiple unmodified OpenMP applications within each Faaslet, and does not require users to provision and configure the compute layer but instead relies on underlying serverless platform to dynamically scale based on the demand from the Faaslets. The implementation of FaasMP requires creating parallel section (§6.5.1), supporting work-sharing (§6.5.2) and synchronisation constructs (§6.5.3), and necessitates efficient thread pooling (§6.5.4).

6.5.1 Forking with Wasm threads

As shown previously in Figure 4, the compiler translation layer is responsible for gathering the necessary informations to fork the process and hand them over to the chosen library backend, in this case, the local backend. The user application is originally running inside a Faaslet controlled by the `WavmModule` class. That Faaslet can itself be an OpenMP thread that was previously forked by our runtime.

OpenMP forking with WebAssembly threads. We perform the intra-Faaslet forking using the current WebAssembly thread proposal mechanisms. Figure 6 shows how the stack allocation and memory mapping that takes place within the original faaslet memory when spawning two OpenMP threads. Two stacks (labeled `Stack1`, `Stack2`) are allocated from the original module heap to be used by the two WebAssembly threads. Those threads are then executed given the `function` and `args`

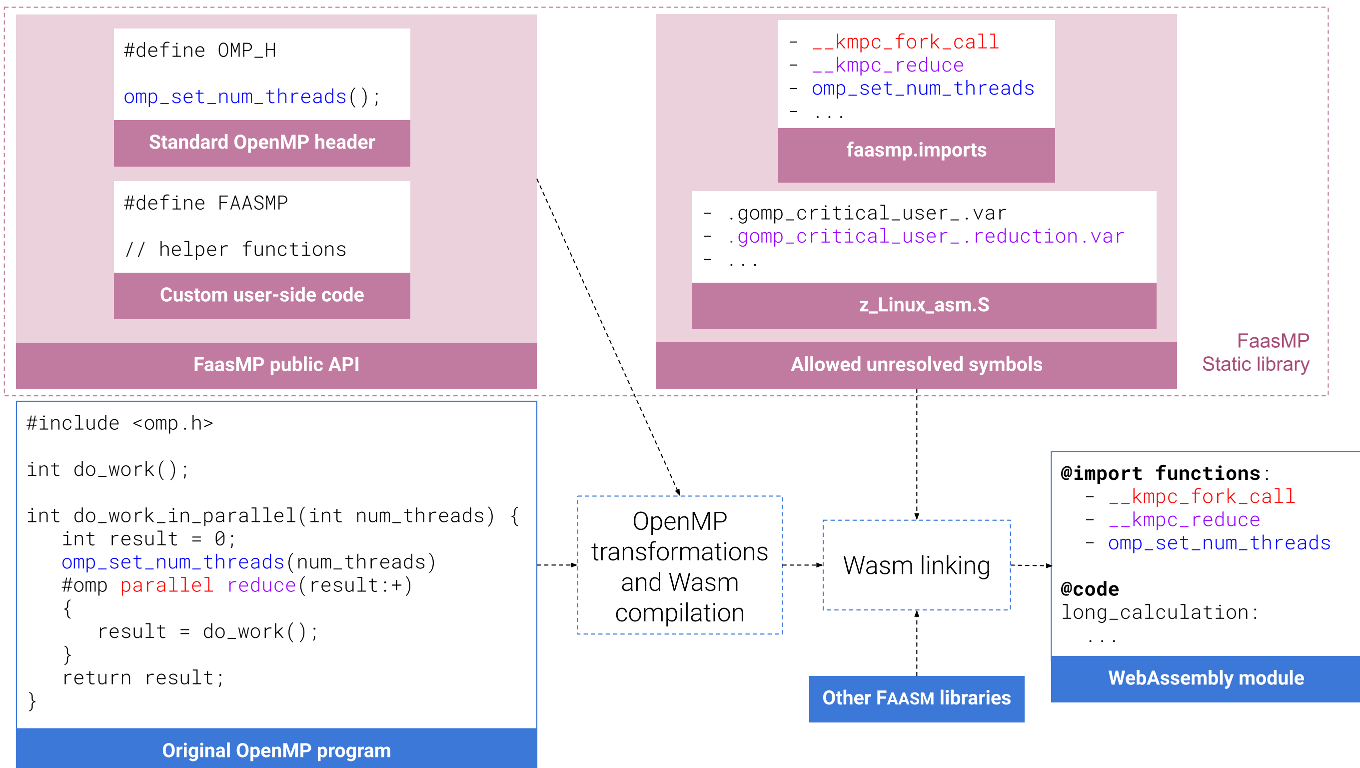


Figure 5: Overview of the building mechanism and symbols resolution.

provided by the translation layer within the same `WavmModule` context as their parent to share the environment of the forked Faaslet.

Nested parallel sections. We control the parallel section through a class called `Level`, named that way because parallel sections can be nested inside one-another. The library needs to manage meta-information about the tree formed by the nested parallel sections. We keep track of this hierarchy because users can query about it at runtime with functions like `omp_get_level` that returns the number of nested parallel regions enclosing the calling thread. OpenMP runtimes such as `libomp` aggressively optimise non-active levels (i.e. levels with just one thread), and the API prescribes users can be aware of when such optimisations are taking place, e.g. using `omp_get_active_level` that returns the number of active parallel regions enclosing the calling thread. The `Level` class is shared by all the threads but is frequently accessed and thus cannot be synchronised. Instead of having to manage a lock-free concurrent tree to control such information, the relevant data is passed down to the children threads on creation of a new parallel section, and each `Level` is locally aware of its current depth and active depth.

Runtime context information. The threads must be aware of their parallel context at runtime. `Level` is also responsible for storing the parallel section information that are in majority immutable and thus do not present synchronisation issues. A common pattern for OpenMP programs is to dynamically create an array of the size of the next parallel section's number of threads to hold the results they will be producing. The users can either query the runtime for the next number of threads, or set the desired number of threads for the next parallel section. The OpenMP API allows for several ways to set the next number of threads which take precedence over one another and might apply to all future children parallel sections or only the next parallel section of this level. `Level` again is made locally aware of those informations and we tested our implementation to mimic the behaviour of `libomp` to not surprise the users or break their optimisations because of a different handling of levels.

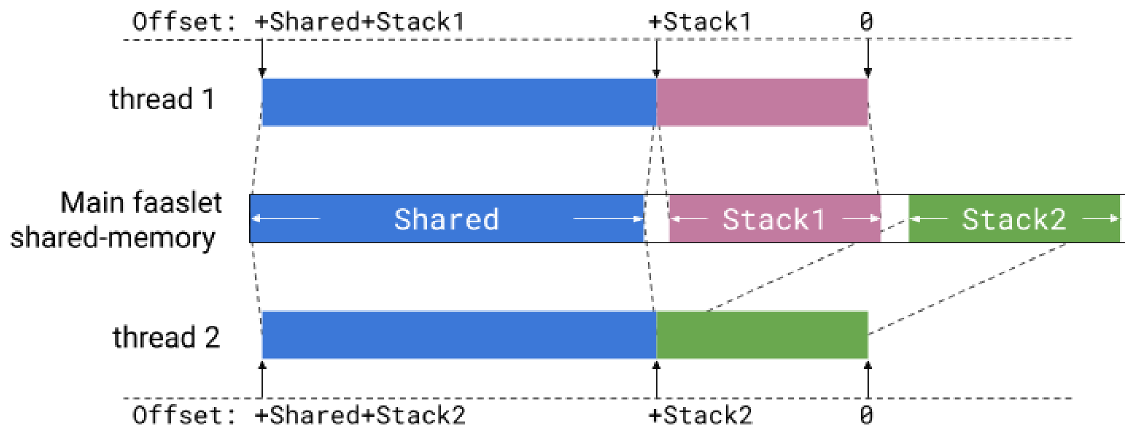


Figure 6: WebAssembly thread shared memory and stack mapping

Multi-tenancy. Unlike existing OpenMP runtimes, our library serves multiple users simultaneously, an idea based CloudFlare's suggestion of sharing libraries [36] to save resources. We use thread local storage (TLS), a mechanism FAASM already trusts for managing its runtime tools between Faaslets, to handle the user's OpenMP state. This shields us from some memory optimisations and implementation tricks that `libomp` can perform thanks to its shared static lifetime with the program, but instead we gain on users isolation and resource efficiency.

Resource access and scheduling. Internally, FAASM spawns a number of worker OS threads on each node it is deployed on, and multiple Faaslets can be safely executed concurrently on a single machine. The local WebAssembly threads share the forked Faaslet's fair resource access mechanisms by inheriting the CPU and network `cgroup` of the process. The Linux completely fair scheduler [58] handles the local scheduling under the `cgroup` constraints but still provides the same concurrency guarantees as usual and thus will not change the expected behaviour of the application by the user.

Security. The compiler translation layer cannot trust its inputs which may have been fabricated by malicious users calling the compiler's private functions itself. This is indeed possible because the private compiler library symbols are added in phase 1 of the faaslet building process that is not trusted (§5.4). Function or data pointers can be handled safely thanks to the WebAssembly software fault isolation guarantees, however, an important recurring argument provided to the library is the current thread number (TID). Existing runtimes like `libomp` commonly use this number to index into arrays, which if we also did so could lead to easy exploits. We instead use TLS and set the TID on thread initialisation to retrieve it when needed. This mechanism is trusted because the TLS is outside of the WebAssembly memory control and thus not in the user's control.[5]

6.5.2 Loop support

We carefully mirror the behaviour of LLVM's `libomp` [88] to handle the `for` pragma, OpenMP's flagship work-sharing feature, in order to preserve the fine tuning applications might have done around them. The main part of the loop scheduling process in OpenMP boils down to calculating the loop block limits assigned for each thread.

Static scheduling. The exact semantic of static loop scheduling is dependent on many factors. The compiler's private library API provides the overall `lower` and `upper` limits of the loop, the stride size (`incr`) and the optional user-provided `chunk_size` that we also support. For a parallel section of n threads with TIDs $t_1, \ldots, t_n$, when the `chunk_size` is provided, the ith block limits, $i \in [0, n)$ are given by $\text{lower}_i = \text{chunk} \times \text{incr} \times t_i$ and $\text{upper}_i = \text{lower}_i - (\text{incr} \times \text{chunk})$.

When `chunk_size` is not provided, the direction of the loop, the size and the signedness of the

[5]Such security concerns could have been an issue for alternative design (see §6.3.3), since `libomp` was not designed with security requirements in mind.

iteration variable must be taken into account[6] to avoid overflows in calculations and the loop is split evenly between the threads by simple division of its overall size. A special case arises if the loop is smaller than the number of threads in the current parallel section, in this case some threads are not given any work to do while the others are given a single item of the loop.

Other concerns. OpenMP has many features to allow users to optimise parallel for loops, for example the `collapse` clause allows to parallelise multiple nested loops automatically and users can use different scheduling options like `dynamic` scheduling using pooling. We scrupulously test our implementation to provide a similar behaviour to Clang and implement static scheduling since it is the only scheduling we have encountered in existing programs. We leave implementing alternative scheduling strategies as future work.

6.5.3 Threading and synchronisation support

A crucial part of a shared-memory parallel processing API is its synchronisation constructs. OpenMP provides thread synchronisation mostly on a a per parallel section basis. The `Level` class can thus be further extended to handle the level synchronisation. WAVM is using pthreads to execute the underlying threads so the usual POSIX synchronisation constructs can be used to implement the local OpenMP constructs.

Barriers are used substantially throughout the OpenMP API. The API has the explicit `barrier` pragma and many other pragmas like `for` or `single` include implicit barriers in their semantics. We design an efficient barrier using C++ mutex and condition variables which wakes up all threads simultaneously when the last thread synchronises. Our barrier only needs to be allocated once, at the start of the parallel section, and can be reused throughout it safely. We further optimise this by allocating our barrier as part of the `Level` class that is always allocated at the same time as the barrier to reduce the forking latency by removing an allocation.

Critical sections are handled with a unique lock for the parallel section. Locks are normally generated by the compiler every time they are needed throughout the application, and then locked by the runtime library in corresponding sections (e.g. critical and reduction code blocks). Clang's `libomp` uses many custom types of locks, including futex, ticket locks, and atomic variables or flags[7] for performance purposes. This complexity serves little our future intentions of distributing the library and as such all the locking purposes are handled with unique mutex allocated as part of the `Level` class similarly to what we did with the barrier to limit allocations.

OpenMP API offers more mature memory ordering than WebAssembly. The compiler private library support a plethora of atomic instructions for a multitude of sizes and signs of integer, floating points and complex variables; operations that must be used when the compiler does not itself inline the atomic instructions because it cannot ensure the validity of this operation for the target architecture. To safely support those operations, we would require the finalisation of the WebAssembly threading proposal to outline memory ordering semantic of the modules and appropriately match it to OpenMP's relaxed-consistency memory model [74]. Instead, we follow the stronger sequential consistency before the proposal is finalised and implemented in LLVM which would allow us to ensure the code generation for the OpenMP and WebAssembly combination respects the semantics of both memory models for the target architecture.

We can still implement `omp flush` to synchronise the thread's local view of the memory using the memory fence compiler intrinsic `__sync_synchronize`[88].[8] OpenMP flush can be used to implement spin locks, and depending on the architecture FAASM runs on, it might be necessary to make the flushing Wasm thread yield the CPU[88]—this is possible because yielding is in the WebAssembly threading proposal and is implemented by WAVM [64].

[6] This can be safely templated with C++ `std::make_unsigned<T>::type`

[7] The GNU compatibility symbols are notably used for this purposes.

[8] A WebAssembly compiler might directly generate an `atomic.fence` instruction instead in the future

6.5.4 WebAssembly thread pool

After running experiments on a variety of existing code, we realised that our library palled in comparison to the native libraries on a specific pattern used in certain kinds of scientific applications [93]. Listing 16 shows this pattern which we identified to be an excessive creation of small parallel sections. In line 3, the parallel section is in an outer for loop causing 100 fork-joins of 10 threads each. This could be refactored instead into an efficient `omp for` construct causing a single fork to be generated. We initially assumed in our design this would not be an issue because OpenMP, and the fork/join model in general, is notoriously slow at forking and thus assumed such a code would be an anti-pattern not found in existing applications. However, it is often employed (probably by accident), when functions containing a parallel sections are called in a loop.

```
1 int main {
2     for (int i = 0; i < 100; i++) {
3         #pragma omp parallel num_threads(10)
4         do_work();
5     }
6 }
```

Listing 16: Excessive parallel section creation

Optimising Wasm stack allocation. The majority of the forking time is spent dynamically allocating stacks from the Wasm memory and setting up the threads arguments. We reduce most of the thread spawning overheads by using a thread pool instead. Two competing designs were implemented, one using a condition for each worker, and one using a shared synchronised queue. While both reduced the runtime of those worst case programs by several orders of magnitude, the design with the work queue was consistently faster.

```
1 class PlatformThreadPool {
2 public:
3   PlatformThreadPool(size_t numThreads, WAVMWasmModule *module);
4
5   friend int64_t workerEntryFunc(void *_args);
6
7   std::future<int64_t> runThread(openmp::LocalThreadArgs &&threadArgs);
8
9   ~PlatformThreadPool();
10 ...
```

Listing 17: Thread Pool API

We outline in the Listing 17 the API of the thread pool exposed to the OpenMP library runtime. The constructor takes in the number of threads in the pool, which can be either fetched from the user configuration or defaulted to the machine's number of cores or limited by the runtime to a lower number. The other argument is the `WAVMWasmModule` which, as shown in Figure 4, handles the module's memory and thus is used to allocate the thread's stacks during the module initialisation.[9] The stacks will be reused throughout the duration of the program by reseting the stack pointer to the beginning for each new job. The friend function `workerEntryFunc` is the wrapper for the workers to handle the internal work queue management, set up the TLS multi-tenancy mechanisms explained previously and pass through the `_args` to the WAVM [64] threading API that handles thread arguments as a raw `void` pointers of arguments.

To submit a job, the runtime library efficiently pass-in an r-value reference of the arguments to set-up the thread (containing `function` and `args` from Figure 4). The pool sets up a `promise` that

[9] If necessary, lazy initialisation of the workers could also be implemented with little modifications.

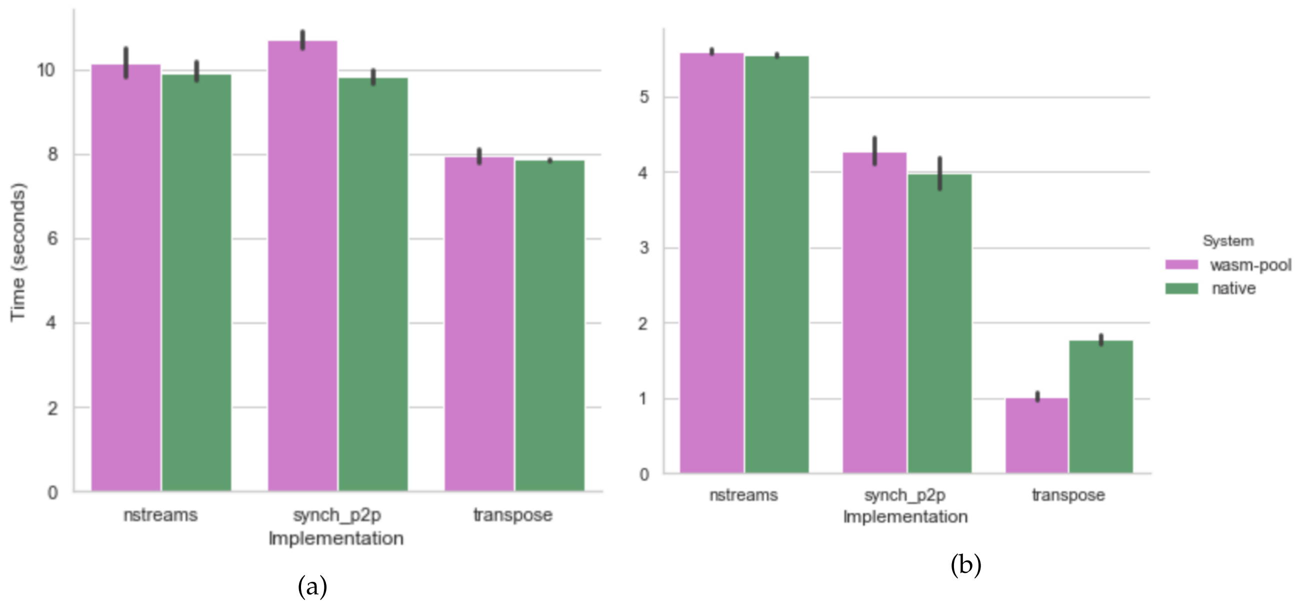


Figure 7: Existing linear algebra applications running FaasMP (wasm-pool) and `libomp` (native)

will be resolved by the worker and returns a `future` handle to the library that can use it to join the threads. The destructor gracefully waits for the OpenMP work completion before shutting down the threads.

Hence, this design fixes our two initial problems of requiring a new stack allocation for each new thread and the inefficient argument handling thanks to the reuse of the thread stacks and the use of move semantics. It also allows for more efficient use of the Wasm memory since the previous thread stacks could not be reclaimed because they were not obtained via a normal allocator.

6.6 Experimental evaluation

Native local baseline. We use Clang's native compiler local library `libomp` [88] when evaluating FaasMP against a state-of-the-art OpenMP implementation. We use the same unmodified code for both set-ups and compile the native programs using the same version of LLVM for all experiments, except for §6.6.2 where the native program runs `libgomp` and musl [76, 66], inside a Docker container running Alpine Linux [94] with full CPU access.

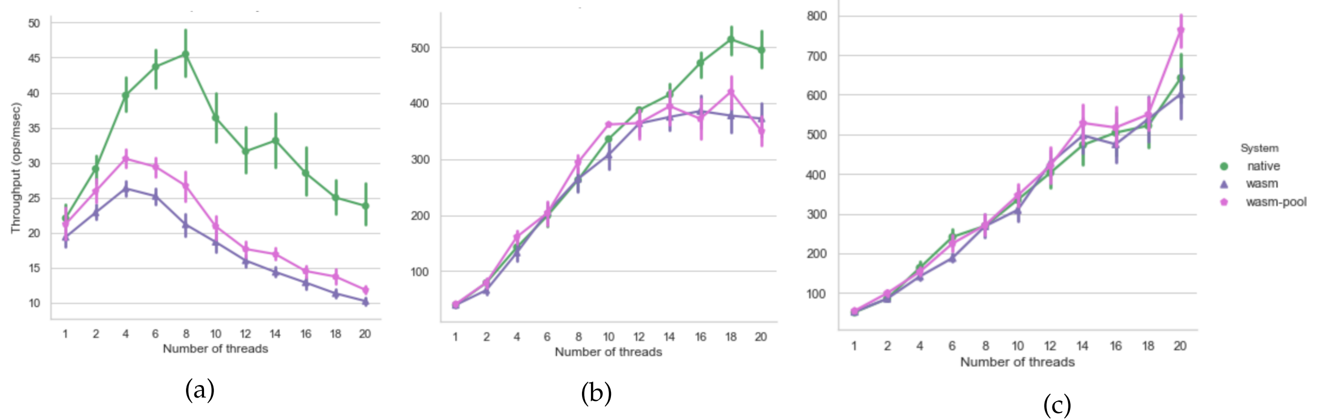
Metrics. We will use execution time, throughput and latency to evaluate the performance of our system, as well as discussing the broader usability of our work.

Testbed and set-up. All experiments are run on the same cluster made of Intel Xeon E5-2660 2.6 GHz machines with 32 GB of RAM connected by a 1 Gbps connection. We use Redis [46] for the FAASM state, deployed in the same cluster on a remote host, unless otherwise specified.

6.6.1 Linear algebra applications

This experiment compares the performance and scalability of FaasMP and `libomp` on existing matrix kernel applications. The code is taken from Intel's parallel kernels [95] and was not modified before being compiled through our toolchain. The experiments are memory intensive, and their runtime is almost entirely spent in OpenMP code, including the testing code at the end of each kernel, checking the operations completed successfully and without race conditions. They make large use of the core OpenMP API: `barrier`, `master`, `flush`, `for`, `parallel [for]`. We plot the average runtime over multiple runs and include the variance on the graphs.

Figure 7a shows how little overhead our library suffers from despite being integrated into a multi-tenant serverless platform running on WebAssembly. The performance is comparable to `libomp` and



(a) (b) (c)

Figure 8: `omp for` with different for loop sizes using dynamic threads (wasm), thread pool (wasm-pool) and `libgomp` (native).

the runtimes are within 10% of one another.

Figure 7b shows how the libraries perform when more threads are used and where the efficiency of the synchronisation is more important. Although two experiments have similar runtimes, we perform significantly better on the transpose program that makes heavy use of the `barrier` construct. Its runtime is therefore significantly affected by the performance of the synchronised variables that keep track of the threads having reached the barrier, and by the work distribution mechanism, which in this case is the thread pools of the libraries.

Our implementation is integrated inside a serverless runtime which scales thanks to its integration with platforms like KNative [33]. For running this code outside of FAASM and without having to provision machines, a user might decide to build a container and use a system such as AWS Fargate [96] that will show similar performance (up to 4 vCPUs), scalability and fairness guarantees as our approach. However, the user will be shielded from the code distribution to multiple machines that FaasMP can also provide.

6.6.2 Local performance characteristics

In this experiment, we study in detail the performance characteristics of FaasMP and its local thread pool when parallelising for loops to compare it against expected OpenMP runtime behaviour. The experiment consists of three computationally intensive micro-benchmarks designed around the `omp parallel for` construct of increasing loop size (tiny, small and big). The work quantity stays constant within each experiment, even as the number of threads increases, which in turn decreases the compute to thread ratio. The source code is identical for all experiments and the native toolchain is modified to use the same libc as the WebAssembly one to eliminate the performance difference between glibc and musl.

Figure 8a considers the operating latency as we increase the number of threads that we have to fork for an almost no-op job. The thread pool design improves on our simpler initial implementation of on-demand dispatching. Both systems exhibit higher overheads than the native local runtime, although they show consistent behaviour with `libgomp` as the number of threads increases. FaasMP is designed to distribute larger jobs across hosts where the network latency dominates the cost of local operations—I/O is on the order of milliseconds compared to microseconds for OpenMP thread scheduling. As such, throughput is often optimised against latency in our implementation choices because of the distributed capabilities of our library.

Figure 8b shows a small compute/thread ratio and FaasMP reaches the peak workload with 12 threads compared to 18 threads for `libomp`. The former successfully shows constant throughput even as more threads are used to execute the same amount of work.

Figure 8c shows the linear scalability from all systems for a trivially parallelisable task, as we

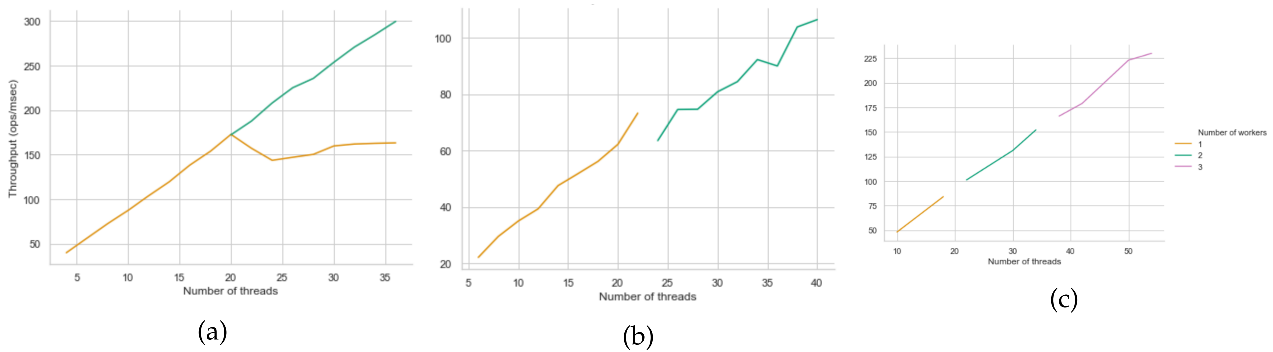


Figure 9: Scalability of different arithmetic reduction methods across hosts

would expect. The overheads for OpenMP are not meant to dominate or increase linearly with the number of threads and instead can be amortised.

Overall the behaviour of FaasMP matches previously set expectations for OpenMP work-sharing [80] in term of both linear throughput and peak work. However, as can be seen in this experiment and in §6.6.1, we suffer from higher variability in the runtimes. Our integration with FAASM increases the binary size compared to having a standalone library which can be an issue for the CPU instruction cache, but the noise in the experiments is mainly due to the WebAssembly virtualisation that was shown to entails more CPU turbulences [97] because of the poorer Wasm code generation compared to native code. As the popularity of WebAssembly continues to grow and compilers improve support for it, we expect this issue to be less prominent.

6.6.3 Distribution experiments

In this section, we show how FAASM can seamlessly scale OpenMP applications to several hosts to improve performance.

Distributed Monte Carlo methods. In this experiment, we measure the scalability of FaasMP on distributed reduce operation and compare it to existing state-of-the-art serverless platforms. We implement the same Monte Carlo simulation algorithm as found in Crucial [26], but in C++ with OpenMP rather than Java. This program is used to calculate digits of π based on the simulation of uniform random numbers in the unit square to approximate the area of the quadrant of the unit circle. This approach is similar to other existing stochastic simulation methods that are frequently implemented in OpenMP, and this application extends to further Monte Carlo integration techniques. We compare the performance of different reduction strategies. The OpenMP program uses the `reduction` clause of the parallel section to aggregate the result.

Figure 9a shows the case when Faaslets are allowed to directly push reduction data to the state which is then responsible for running an associated routine to update the data accordingly—in this case an atomic `INCRBY` operation. This mechanism is similar to the method shipping approach used by Crucial to implement the `AtomicLong` object. The counter is held in their distributed object store they used remote methods invocation to update it [26]. In our experiment, we observe that even when the scheduler starts to offload Faaslets to other available workers, the throughput of the system keeps increasing linearly, whereas, if it did not, the application plateaus at a constant throughput.

Figure 9b shows the impact of lock contention when using a remote locking mechanism to synchronise the reduction variable. We witness a pronounced drop in the throughput only just one and two workers; a gap that would worsen as more workers are running the job. We also notice significantly more jitter when several nodes are competing for the global lock. The FAASM global state consistency mechanism here only allows two Faaslets at to try and to acquire the remote lock simultaneously, but even this small amount of contention is too high for the system's network latency and the locking mechanism is severely impaired.

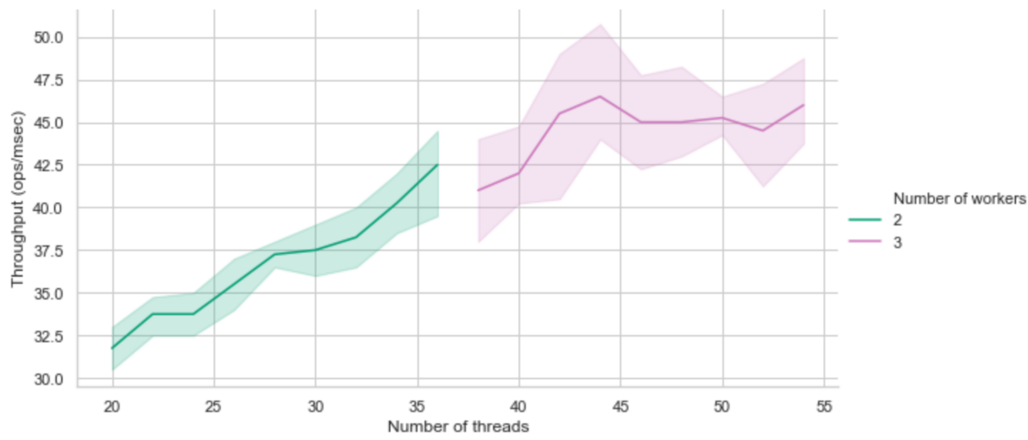


Figure 10: Peak distribution behaviour

Figure 9c shows our reduce implementation, backed by the `FaasmVector`, can achieve similar linear scalability as the method shipping experiment without relying on the consistency guarantees of the FAASM state storage layer. The final pulling of the vector to be accumulated by the main Faaslet does not impact the scaling of the program.

As discussed previously, it is expected by users that the OpenMP implementation should show such linear scalability when executing `omp parallel for [reduce]` constructs for parallelisable applications. Obtaining linear throughput as the number of threads and workers increases is also becoming the standard behaviour for stateful serverless runtimes and storage systems [98]. On this specific application, Crucial uses Infinispan [99] instead of Redis [46]. This obtains similar results [26] but requires the use of a custom Java API and code annotations to implement the program.

Distributed peak performance. This experiment looks at the behaviour of the runtime as peak scalability is reached. Reaching peak performance happens in two cases: (1) there are no more machines available for FAASM to scale to[10] or machines are in the process of being provisioned after a sudden peak, which for VMs in the cloud can take several minutes; or (2) the program compute/thread ratio does not dominate enough the forking and scheduling overheads.

We modify the Monte Carlo simulation program above for the threads to have a smaller amount of work to do in the distributed part of the program. The compute to thread ratio thus diminishes as the number of threads increases.

Figure 10 shows how the throughput plateaus when maximum scalability is reached, similarly to what happens locally in Figure 8b when the peak parallelisation is reached but on multiple hosts. This highlights a beneficial property of our forking/scheduling mechanism: even if the number of threads and hosts increases, the application does not use more hosts than it requires. This is achieved without any tuning from the user or complicated runtime state analysis [82, 83] and saves on scarce cloud resources such as RAM and network. The distributed work-sharing implementation of FAASM can efficiently schedule those extra Faaslets without overloading the hosts and without diminishing the throughput, thanks notably to the forked Faaslets quick spawning time of $\sim 500\mu s$ [25].

We confirm in this experiment the efficient work distribution claims of FAASM that can be efficiently exploited to execute OpenMP threads. Our system makes more efficient use of the cluster resources than other serverless systems that rely on containers as their distribution mechanism like PyWren [18, 19, 20] or Crucial [26] thanks to the low resource footprint of Faaslets and the consequent high-density of threads per hosts. Through FAASM again, we offer a more fine-grained scaling and efficient use of resources than other stateful serverless platforms like Cloudburst[27] thanks to the quick creation and destruction of Faaslets.

[10] Although serverless intends to be virtually infinitely scalable, some deployments might have actual limits

Pragma	FaasMP local
`atomic`	✗
`barrier`	✓
`critical`	✓
`flush`	✓
`for [simd] [schedule=static]`	✓
`master`	✓
`parallel [reduce]`	✓
`single [nowait]`	✓

Table 8: Local runtime pragma support

6.6.4 Usability and potential

This section judges the usability of FaasMP for programmers who are either seeking to use the library or extend it. Table 8 shows the extent of our local support of the core OpenMP API, with only the `atomic` construct missing because of the immaturity of WebAssembly. We also implement static scheduling because it is the default of all known runtimes and is the only scheduler that we have encountered in existing code, probably because it is the one showing the smallest overheads out of all the schedulers [80].

We now assess how easy it is for a user to adopt our system. We have already shown that the performance behaviour of FaasMP corresponds to the user expectations from an OpenMP runtime library, thus respecting their local optimisations, and we have also shown support for most of the core OpenMP API. The main remaining obstacle for users can thus be the WebAssembly compilation that, despite differing from native compilation, can easily be undertaken in the local environment and quickly tested.

Figure 11 compares the user mental model required from the user to cross-compile their applications to Wasm compared to their current native tools knowledge. The initial role of headers are similar and the compiled objects (ELF binary for native or a Wasm module) lead to the same unresolved symbols present in the binary and to which can be attached profiling tools. The libraries are in both cases viewed as a black box, and like for the native case we only require recompilation of the user program if the compiler's internal ABI changes.

6.6.5 Other performance considerations

We explore what performance characteristics of FaasMP might ward off users.

Latency. Our system shows a higher latency for forking or joining WebAssembly threads or Faaslets than local compiler runtimes, so users should seek to avoid excessive creation of parallel sections, especially in a loop. Although OpenMP's behaviour varies based on the platform and architecture is it running on, this is an advice that has always been considered a best-practice [90, 80, 85].

Table 9 shows the latency for a minimal reduce operation, broken down by the cumulative overheads. The measured section forks, assigns a local value to the reduction variable and reduces it. FaasMP is an order of magnitude slower when there is no network communication involved and two orders of magnitude slower when using state. Upcoming modifications to the FAASM state should improve those already encouraging results by removing some of the network latency through the use of distributed shared memory. These results are also an order of magnitude better than the reduce overheads of Cluster OpenMP [85]. We know those results are not comparable because they were running a different experiment on older hardware, but this gives an order of magnitude at which

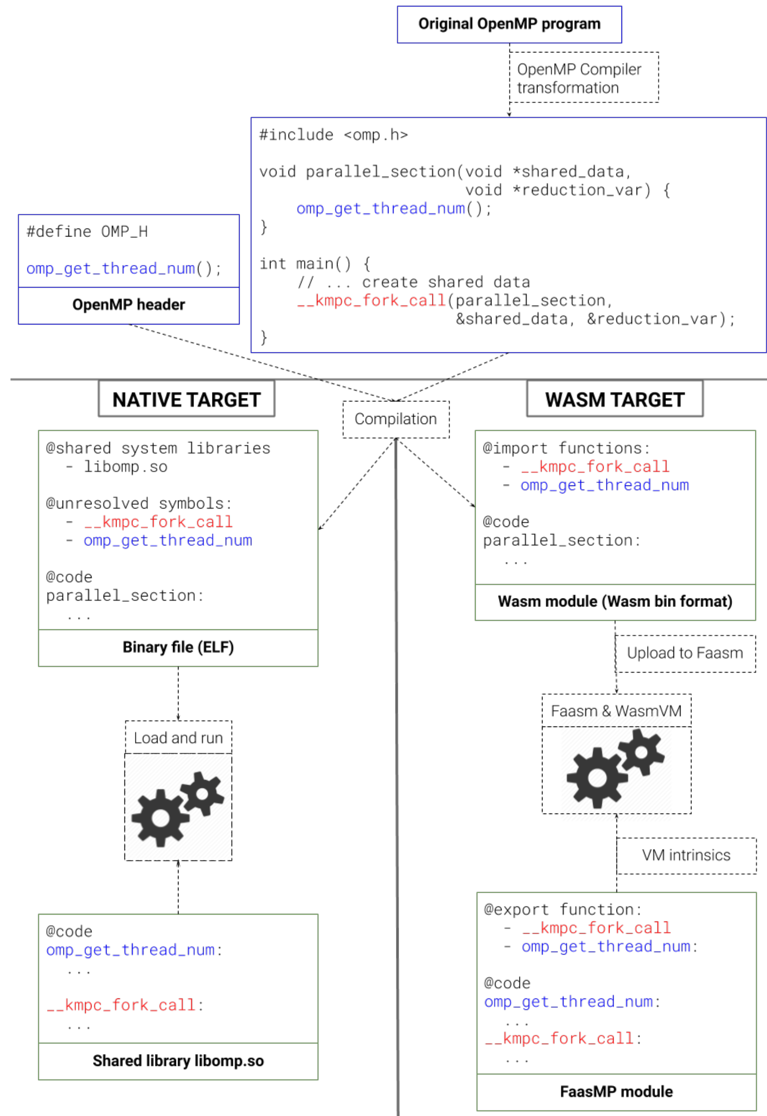


Figure 11: Comparison of the cross-compilation processes for native (left) and FaasMP (right) toolchains.

their paged-based DSM layer used to operate on.

Physics simulation. A natural question that arises from Table 9 is what happens when an application is comprised exclusively of latency-critical patterns such as the ones showed in Listing 16. LULESH [93] is a complex hydrodynamics modelling application containing 44 OpenMP pragmas that we were able to compile without modification.[11] An initial `iterations` parameter allows for running the program during a certain logical epoch of time, with a single iteration creating over 10,000 threads. Since most parallel section are called in loops, the theoretical latency to fork and join those sections is of squared complexity to the number of threads, and thus the single order of magnitude of overhead difference shown in Table 9 should result in a two orders of magnitude difference of the program runtime. This was verified empirically and is what motivated the creation of the thread pool, since the dynamic stack allocation overheads was giving results that were five orders of magnitude worst originally. We believe the pinning of local workers to CPU cores, similarly to what `libomp` does [88], would further help with the local performance improvement.

[11] LULESH uses C++ `iostream` for logging, which is not supported by the host interface, and so logging was removed.

System	Time (ms)
Native	2
Local library (thread pool)	13
Faaslet local snapshot	73
Faaslet remote snapshot	140

Table 9: Comparison of reduce latency (no-op)

Other concerns. Other considerations include being currently limited to 32-bit applications by WebAssembly, the lack of a better fault tolerance mechanisms apart from relying on idempotent threads and not having a mechanism to automatically decide between the local or distributed backend. In addition, our solution shares the benefits and issues of FAASM in general [25].

7 FaasMPI: Bridging the gap between HPC and the 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 [28], and HPC frameworks like MPI and OpenMP expose users to hardware-specific features such as SIMD instructions and GPU offloading [74]. Newer HPC languages such as Chapel [100] and Charm++ [101] introduce high-level programming constructs, with NumPy-like arrays implemented with Chapel in Arkouda [102]. However, these are yet to see wide adoption outside the HPC community. In contrast, frameworks commonly used in Cloud environments like Spark [103] and Flink [104] offer high-level APIs [1] in dynamic languages such as Python. Serverless providers most commonly target Python and Javascript support [35, 105, 106], 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 [31] and Microsoft's CNTK [107] and Alchemist [108] 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 [109].

7.2 FaasMPI and 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 [28]. 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; (v) remote memory access and one-sided communication. FAASM also supports custom types and custom reductions. A full list of the MPI functions supported by FaasMPI is given in Table 10.

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 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 [110]. As with all FAASM functions, the output of this compilation is a

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`.
Remote memory	`MPI_Win_create()`	Create region of shared state.
	`MPI_Win_free()`	Delete region of shared state.
	`MPI_Get()`	Pull data from shared state.
	`MPI_Put()`	Push data to shared state and notify receiver.
	`MPI_Win_get_attr()`	Get attribute of shared state.
	`MPI_Win_fence()`	Wait for operations on shared state.

Table 10: MPI functions supported in FaasMPI

WebAssembly file that can be uploaded and invoked on a FaasMPI cluster.

FAASM itself is built on *Faaslets*, a lightweight isolation mechanism which uses WebAssembly for memory safety [32]. 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 architecture

MPI applications operate in the context of a *world*, which encompasses a set of distributed processes that can send messages to each other. Each process addresses others using a numeric *rank*, with ranks ranging from zero to one less than the *world size*. The underlying MPI runtime assigns ranks to processes as part of the initialisation process triggered by a call to `MPI_Init`. Each subsequent MPI call may specify one or more ranks, which the runtime must resolve to determine which processes on which hosts a given message must be delivered to.

In FaasMPI, an MPI world is made up of a number of serverless functions, each of which has its own rank. When the initial call to `MPI_Init` is made, FaasMPI uses the standard FAASM scheduler to invoke the required number of functions. Each FAASM host includes an *MPI Broker*, which performs rank resolution and uses FAASM's distributed state to share data across hosts.

The FaasMPI architecture is shown in Figure 12, which outlines a function with `Rank 0` on one

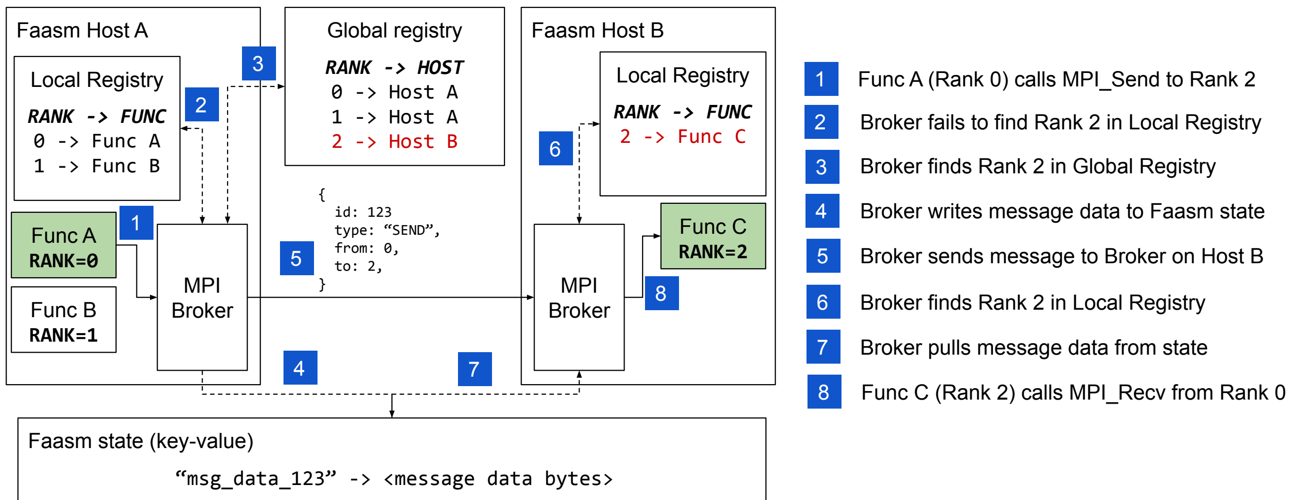


Figure 12: Resolving `MPI_Send` and `MPI_Recv` calls in FaasMPI

host (`Func A` on `Faasm Host A`), sending a message to a function with `Rank 2` on a different host (`Func C` on `Faasm Host B`) using the `MPI_Send` function. The MPI Broker on `Host A` first checks if the given rank is located on the same host by querying the *Local Registry*. If this is the case, the message and its associated data can be transferred purely through FAASM's in-memory shared state. As this is not the case, the MPI Broker queries the *Global Registry* to determine which host the given rank is located on (`Host B`). Before sending the message, the MPI Broker writes any relevant message data to FAASM's global state, making it accessible to all other hosts in the cluster. The MPI Broker then sends the message directly to its counterpart on the relevant host, which pulls the message data from the global state. This MPI Broker then queries its own Local Registry to determine the recipient function for the message (`Func C`). The recipient function can access incoming messages through calls to the relevant MPI function, in this case, `MPI_Recv`.

7.3.1 MPI one-sided memory access

To avoid coordination overhead in point-to-point MPI communication, MPI 2.0 introduced one-sided communication, or Remote Memory Access (RMA). This allows an MPI process on one host to directly access data held in the memory of another MPI process, potentially on another host. MPI is agnostic as to how RMA is implemented, but it may be backed with hardware support such as RDMA, or rely on distributed memory in a supercomputer [111].

FaasMPI supports RMA using FAASM's distributed shared, providing zero-copy access when functions are colocated, and access through FAASM's global state tier when distributed across hosts. This provides a simple abstraction and straightforward implementation, extending FaasMPI's support to a wider range of existing MPI applications.

Listing 18: Distributed SGD application with Faasm

```
t_a = SparseMatrixReadOnly("training_a")
t_b = MatrixReadOnly("training_b")
weights = VectorAsync("weights")

@faasm_func
def weight_update(idx_a, idx_b):
  for col_idx, col_a in t_a.columns[idx_a:idx_b]:
    col_b = t_b.columns[col_idx]
    adj = calc_adjustment(col_a, col_b)
    for val_idx, val in col_a.non_nulls():
      weights[val_idx] += val * adj
      if iter_count % threshold == 0:
        weights.push()

@faasm_func
def sgd_main(n_workers, n_epochs):
  for e in n_epochs:
    args = divide_problem(n_workers)
    c = chain(update, n_workers, args)
    await_all(c)
    ...
```

8 Distributed Data Objects: Object-oriented programming in Faasm

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.

8.1 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 18 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 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`

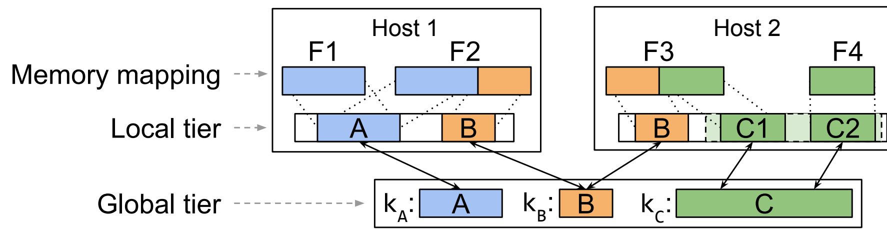


Figure 13: FAASM two-tier state architecture

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.

8.2 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 [12] or Cloudburst [15].

Figure 13 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 18 uses *state chunks* to access a subset of a larger state value. As shown in Figure 13, 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.

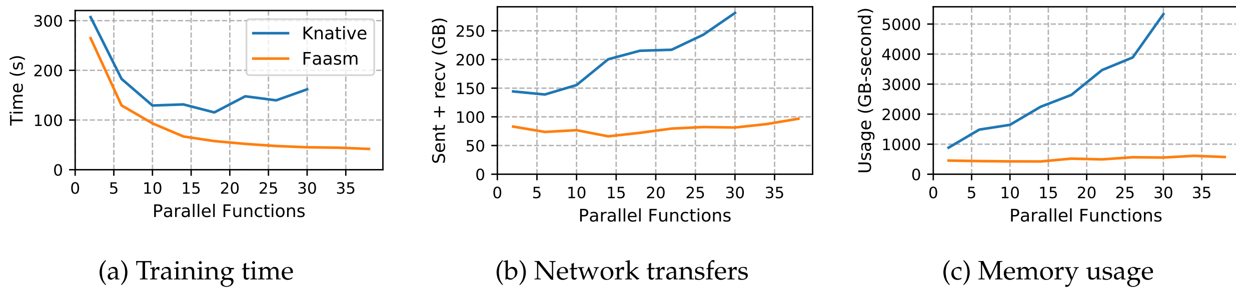
Ensuring global consistency. DDOs support varying levels of consistency between the tiers as shown by `VectorAsync` in Listing 18. To enforce strong consistency, DDOs must use *global read/write locks*, which can be acquired and released for state keys using the functions `lock_state_global_read` and `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.

8.3 Experimental evaluation

To demonstrate the use of DDOs in an experiment, we implement the same distributed *stochastic gradient descent* (SGD) algorithm as in Listing 18 in C/C++ to run text classification on the Reuters RCV1 dataset [112]. This updates a central weights vector in parallel with batches of functions across multiple epochs.

8.3.1 Experimental set-up

Serverless baseline. To benchmark FAASM against a state-of-the-art serverless platform, we use Knative [113], a container-based system built on Kubernetes [114]. All experiments are implemented



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

Figure 14: Machine learning training with SGD with FAASM and containers (Knative)

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 [59, 105, 115]. Note that all memory measurements include the containers/Faaslets and their state.

8.3.2 Experimental results

To test the scalability of the prototype we ran both Knative and FAASM with increasing numbers of parallel functions. Figure 14a 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 14b 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 14c 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.

9 Conclusion

In this deliverable, we have describes a number of new high-level programming models that we have developed in the project in order to better support the building stateful serverless applications in CloudButton. 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. This work therefore 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. 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] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] H. Yviquel and G. Araujo, “The cloud as an openmp offloading device,” 08 2017.

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

[32] 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.

[33] "Knative platform," 2020.

[34] 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.

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

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

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

[38] 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.

[39] P. G. López, M. S. Artigas, S. Shillaker, P. R. Pietzuch, D. Breitgand, G. Vernik, P. Sutra, T. Tarrant, and A. J. Ferrer, "Servermix: Tradeoffs and challenges of serverless data analytics," *CoRR*, vol. abs/1907.11465, 2019.

[40] J. Spillner, "Serverless computing and cloud function-based applications," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, UCC '19 Companion, (New York, NY, USA), p. 177–178, Association for Computing Machinery, 2019.

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

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

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

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

[49] 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.

[50] 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.

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

[52] 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.

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

[54] 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.

[55] 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.

[56] 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.

[57] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[70] 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.

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

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

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

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

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

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

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

[78] 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.

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

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

[81] 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.

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

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

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

[85] 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.

[86] 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.

[87] 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.

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

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

[90] R. Lyerly, S.-H. Kim, and B. Ravindran, "libmpnode: An openmp runtime for parallel processing across incoherent domains," pp. 81–90, 02 2019.

[91] S. Shillaker, "Faasm github repository," 2020.

[92] B. Smith, "Wabt: The webassembly binary toolkit," 2020.

[93] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[94] A. L. Downloads, "Alpine linux," 2020.

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

[96] A. W. Services, "Aws fargate," 2020.

[97] A. Jangda, B. Powers, A. Guha, and E. Berger, "Mind the gap: Analyzing the performance of webassembly vs. native code," *CoRR*, vol. abs/1901.09056, 2019.

[98] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, "Anna: A kvs for any scale," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 401–412, 2018.

[99] F. Marchioni and M. Surtani, *Infinispan Data Grid Platform*. Packt Publishing Ltd., 2012.

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

[101] 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.

[102] 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.

[103] 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.

[104] 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.

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

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

[107] 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.

[108] 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.

[109] 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.

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

[111] W. D. Gropp and R. Thakur, "Revealing the performance of mpi rma implementations," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pp. 272–280, Springer, 2007.

[112] 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.

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

[114] The Linux Foundation, "Kubernetes." `https://kubernetes.io/`, 2020.

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