CloudButton

WP5 – Programming Abstractions for Serverless Computing

Final review

September 15, 2022

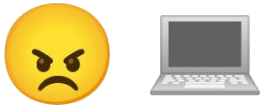
Peter Pietzuch

Professor, Imperial College London

BIG DATA VALUE
PUBLIC-PRIVATE PARTNERSHIP

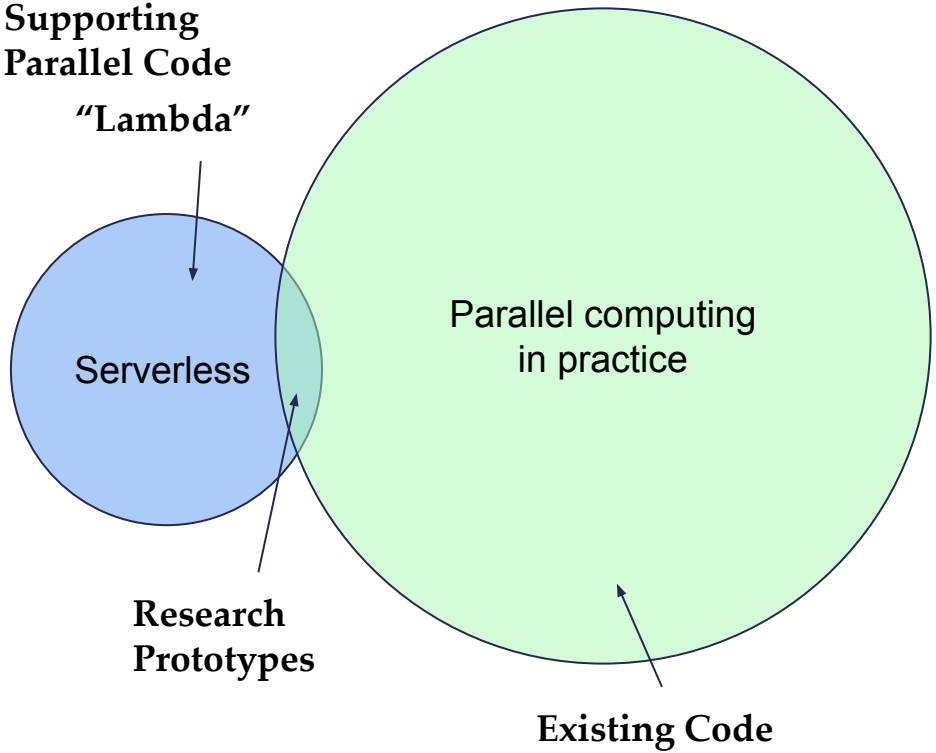
Serverless Programming Models… ?

The Programmer

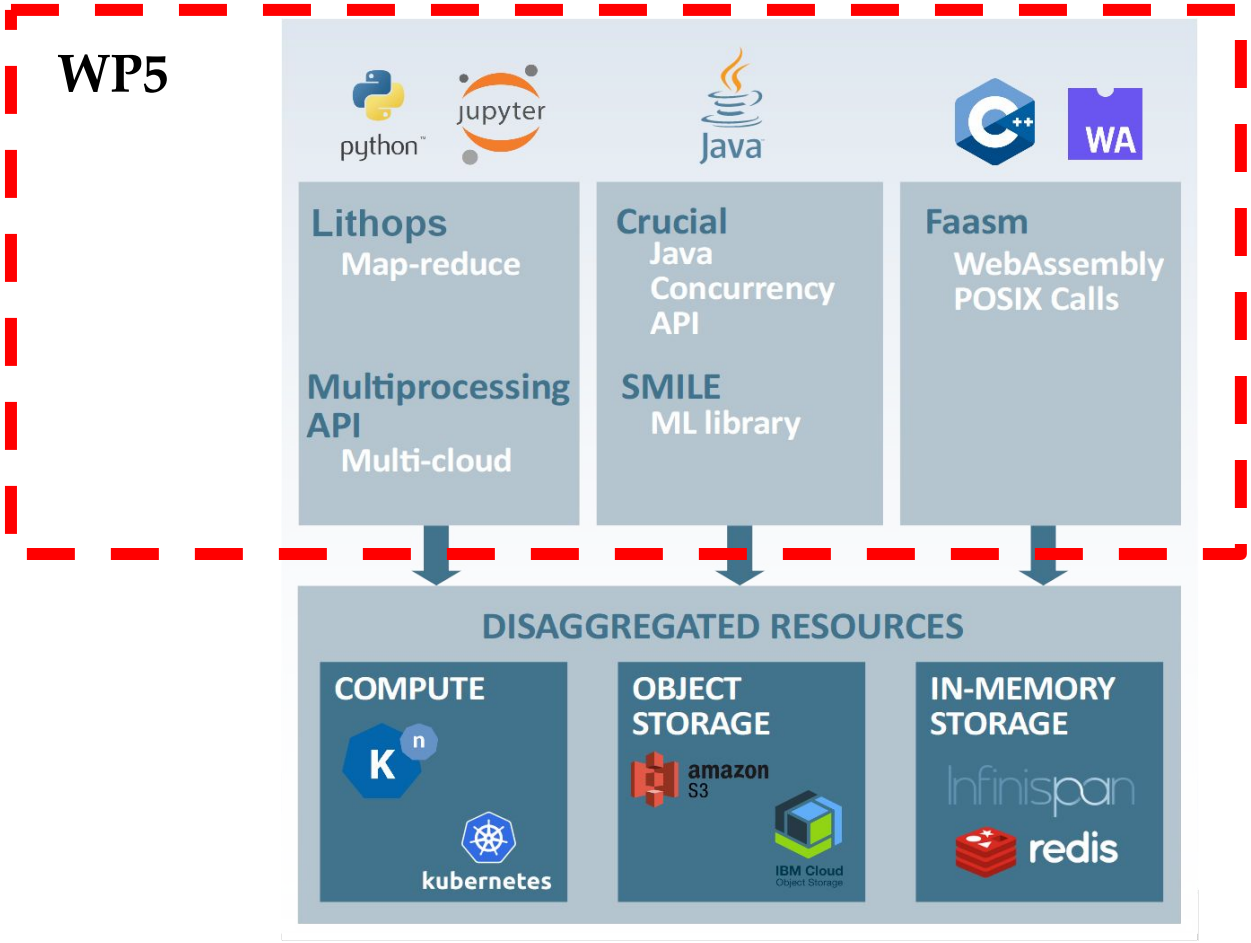


Big Data applications in the Cloud

Supporting Parallel Code



CloudButton Architecture



Work Package 5: CloudButton Toolkit

T5.1 Programming abstractions for Stateful Serverless Computing

- Porting existing Python/Java apps to the cloud

T5.2 Consistency and fault tolerance models

- Faasm: Serverless Programming with WebAssembly

T5.3 Programming tools for porting existing application

- Supporting Shared Memory HPC with FaasMP

T5.4 Application patterns and libraries

- Distributed HPC with FaasMPI

Porting Existing Python/Java Apps to the Cloud

Porting Java Applications to Serverless

Crucial: executes multi-threaded Java applications in the cloud using CloudThreads

Methodology: to port existing Java applications, the programmer must:

1. Replace keywords according to the translation table
2. Make Serializable each immutable object passed between cloud threads
3. Substitute mutable objects by the implementations offered by the DSO e.g:

`java.util.atomic.AtomicBoolean -> org.crucial.dso.AtomicBoolean`

Java API	Crucial API
`Thread`	`CloudThread`
`ExecutorService`	`ServerlessExecutorService`

Java Crucial API

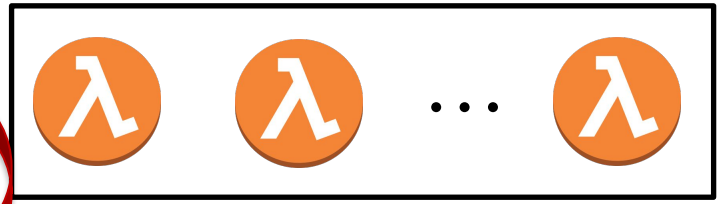
Abstraction	Description
CloudThread	FaaS functions are invoked like threads
ServerlessExecutorService	Following the Java ExecutorService interface, groups of tasks and distributed parallel *for*s are easily run on the FaaS platform
Shared objects	Linearizable distributed objects. Basic collection available: AtomicInt, AtomicBoolean, List, Map, …
Synchronization objects	Shared objects enabling primitives for thread synchronization: CyclicBarrier, Semaphore, Future, …
@Shared	Custom shared objects allowing user-defined updates on the shared data with method calls (e.g., .add(), .update(), .merge())
Data persistence	Shared objects can be replicated by setting @Shared(persistence=true).

Java Crucial API

Client application

```
Thread t = new CloudThread(new MyTask()));
t.start();
t.join();
```

FaaS invocations



Thread code (runs in the cloud)

```
public class MyTask implements Runnable {
  crucial.Object shared = new crucial.Object();

  public void run() {
    X data = compute();
    shared.update(data);
  }
}
```

Remote access

Shared objects store

crucial.Object#shared
public void update(X data) {…}

Objects…

Python Multiprocessing API

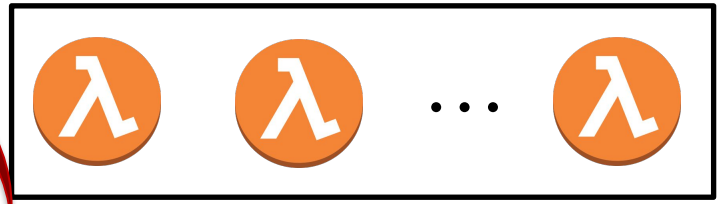
Abstraction	Description
Process	Basic function entity executed in the cloud
Pool	FaaS functions are invoked using a serverless pool of Processes
Communication objects	Objects for function communication: Pipe, Queue
Synchronization objects	Shared objects enabling primitives for function synchronization: Lock, RLock, Semaphore, BoundedSemaphore, Condition, Event and Barrier
Data structures	Custom shared objects allowing user-defined updates on the shared data: value, array
Manager	A manager object controls a server process which holds Python objects (list, dict, Namespace, Lock, RLock, Semaphore, BoundedSemaphore, Condition, Event, Queue, Value and Array) and allows other processes to manipulate them using proxies.

Python Multiprocessing API

Client Application

```
process p = Process(target=my_function)
p.start()
p.join()
```

FaaS invocations



Process code (runs in the cloud)

```
def my_function(args):
    num = Value('i', 0)
    print(num)
    ...
    num.value += 1
    print(num)
    ...
```

Remote access

Remote access

Shared KV Store (Redis)

mp-object#oid-value.set(0)

mp-object#oid-value.set(1)

Faasm: Serverless Programming with WebAssembly

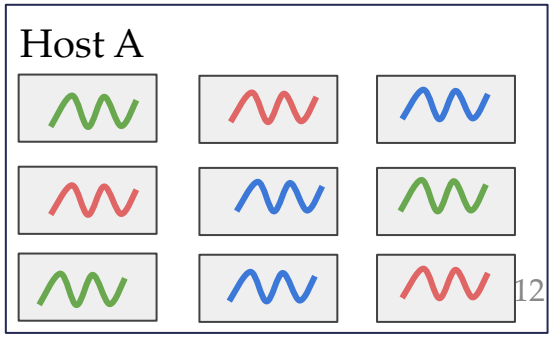
Lightweight Isolation using WebAssembly

WebAssembly

- Memory-safe intermediate language (ISA)
  - Isolation using software fault isolation
  - Preventing instructions from accessing unauthorised memory
- Initially proposed for web browsers, increasingly used in data centres
  - e.g., Fastly, Cloudflare, Krustlet, …

Python/C/C++/JS/Rust functions compiled to WebAssembly modules
- Ahead-of-time (AOT) or just-in-time (JIT) compilation
- Universal binary execution format

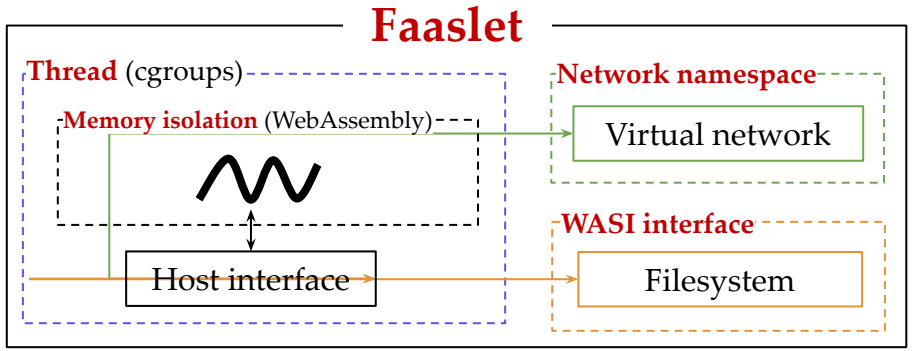
WebAssembly functions share single address space
- High function density per host



Faaslets: Memory Safety & Resource Isolation [USENIX ATC'20]

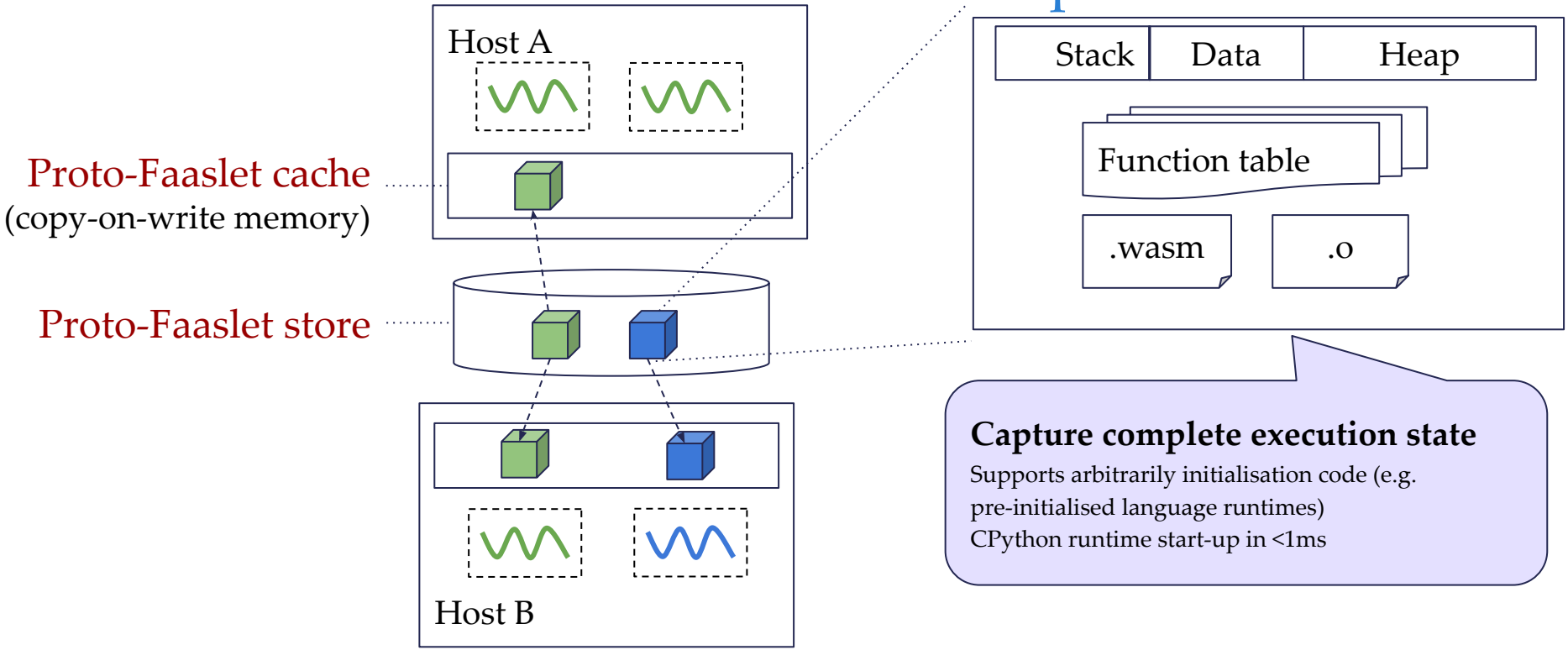
- **Faaslets:**

- **Thread** per function
- **WebAssembly** for memory isolation
- Linux **cgroups** for resource isolation



Host interface:
Serverless-specific virtualised API (with POSIX calls)

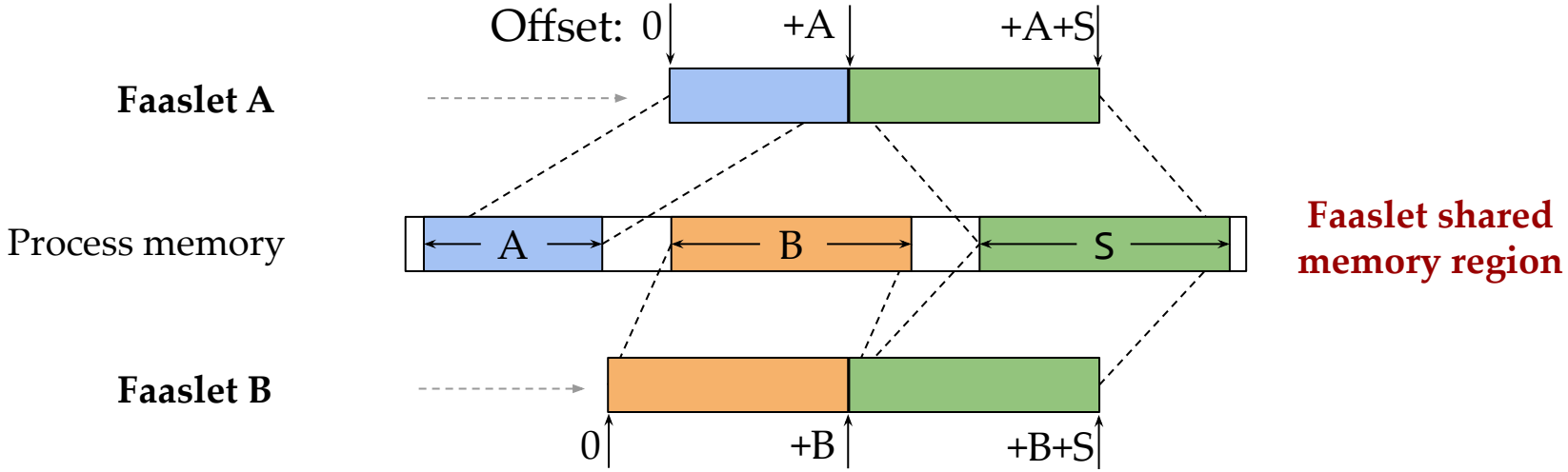
No Cold-Start Problem: Fast Creation of Faaslets from Checkpoints



Faaslets support snapshot and restore operations

Efficient State Sharing with Faasm

- Each Faaslet occupies linear memory region
- Map shared state into linear memory of multiple Faaslets

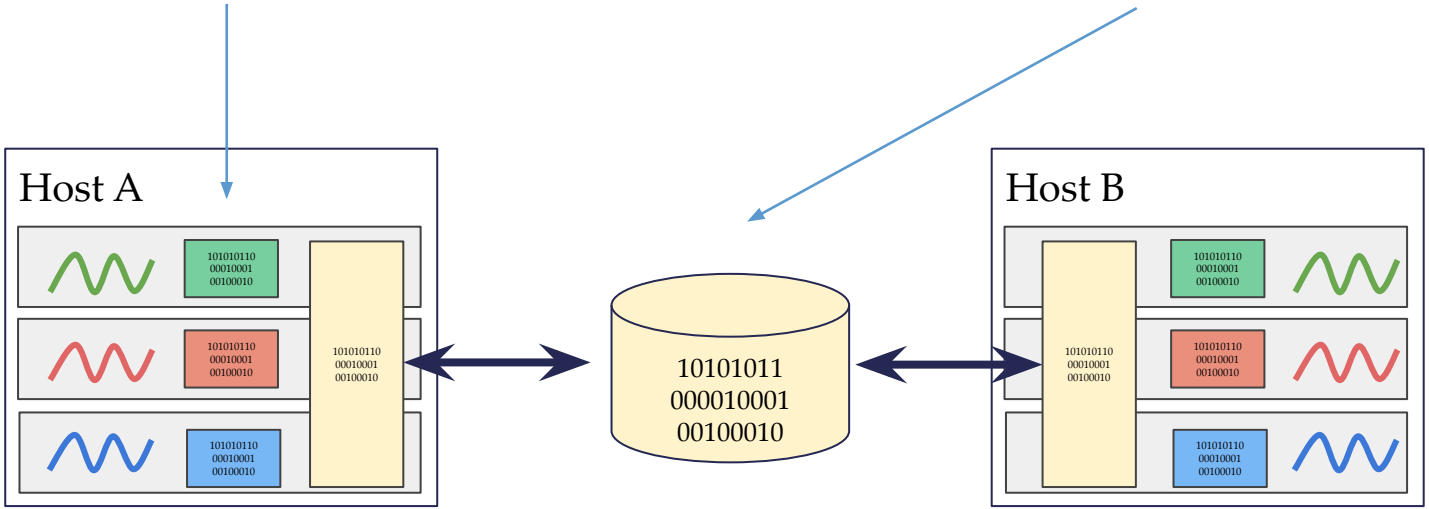


Faaslets allows state to be shared efficiently without page table changes

Distributed State Sharing with Faaslets

Faaslet shared regions - shared memory without breaking isolation

Two-tier state - global & local state tiers with synchronisation



Faaslet Programming Model

Distributed machine learning with SGD

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

@serverless_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()

@serverless_func
def sgd_main(n_workers , n_epochs):
    for e in n_epochs:
        args = divide_problem(n_workers)
        c = chain(weight_update, n_workers, args)
    await_all(c)
```

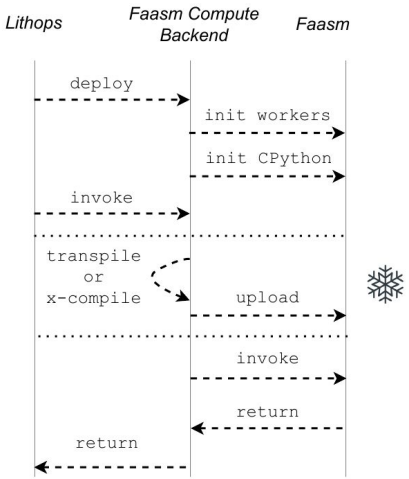
High-level object-oriented abstractions
Read-only matrices
Asynchronous vectors
Flexible consistency

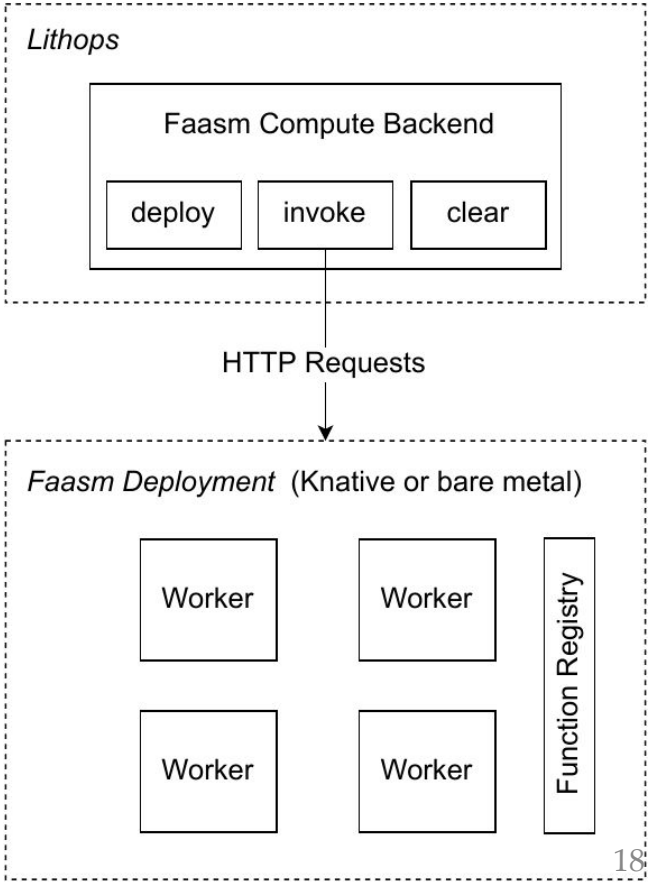
Standard programming constructs
Transparent optimisations
Direct access to shared memory

Intuitive mark-up
Function annotation
Fork-join parallelism

Faasm as Compute Backend in Lithops

- **Faasm** exposes HTTP endpoints to interact with workers and function registry
- Lithops integrates with Faasm deployment regardless of where it runs: locally in Docker, Knative, or bare metal



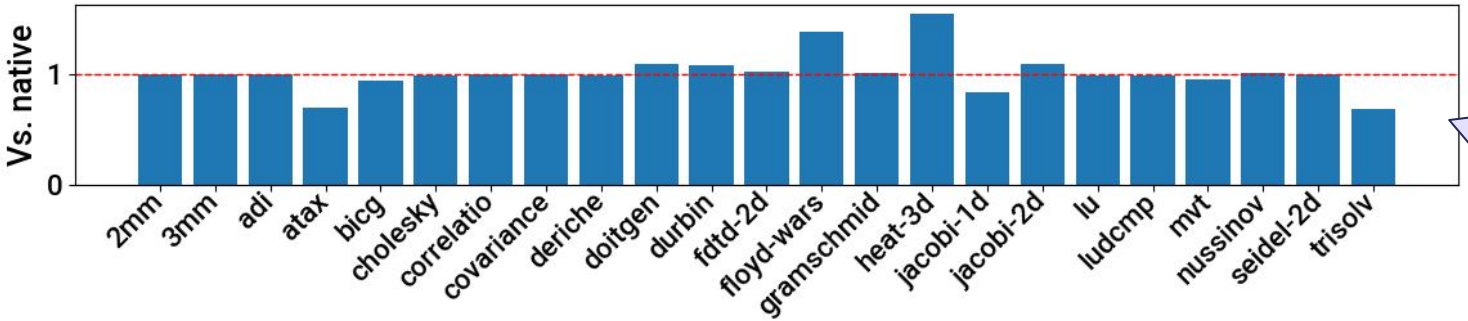


KPI: How do Faaslet Overheads Compare?

	Docker (Alpine Linux)	**Faaslets**	**Faaslets (snapshot)**	**vs. Docker**
Initialisation time	2.8 s	5.2 ms	0.5 ms	5K x
Overhead (CPU cycles)	251M	1.4K	650	385K x
Memory footprint	1.3 MB	200 KB	90 KB	15 x
Max. density on worker	~8K	~70K	>100K	12 x

Lower overheads mean lower latency and costs

KPI: Faaslet Performance with C and Python?



Mostly native performance for C applications

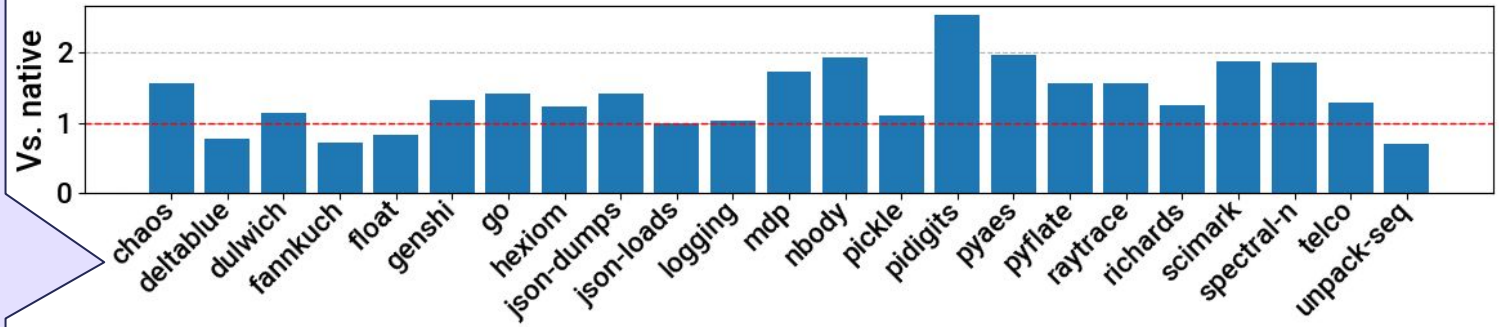
WebAssembly loses certain loop optimisations

More pronounced overhead with Python

e.g. for big integer arithmetic

More instructions, branches, cache misses

Depends on WASM advances



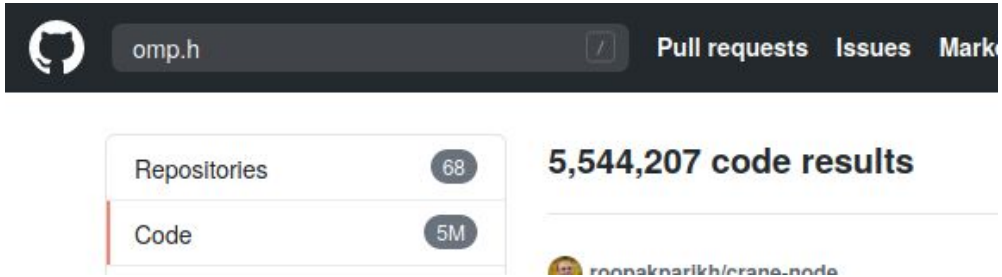
Faasm: Serverless OpenMP + MPI

Why OpenMP in Serverless?

- De-facto HPC ***parallel*** programming framework
- Pragma-based markup for C/C++ and Fortran
- >20 years of development

```
int main () {
    int n_threads = 0;

    #pragma omp parallel
    {
        n_threads += 1;
    }
    return 0;
}
```

Large existing codebase of parallel applications

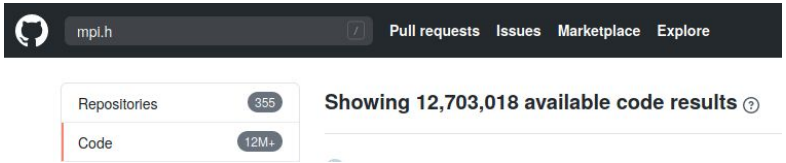


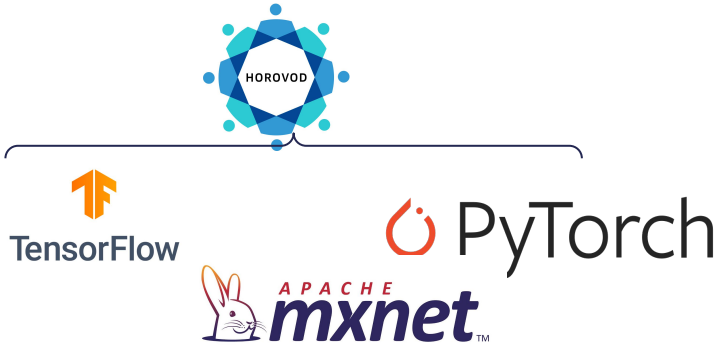
Why MPI in Serverless?

- De-facto HPC ***distributed*** programming framework
- Used *alongside* OpenMP
- Platform-agnostic API for C/C++ and Fortran
- Almost 30 years of development

```
int main () {
  if (world_rank == 0) {
    MPI_Send(...);
  } else {
    MPI_Recv(...);
  }

  return 0;
}
```

Even bigger existing codebase of distributed applications





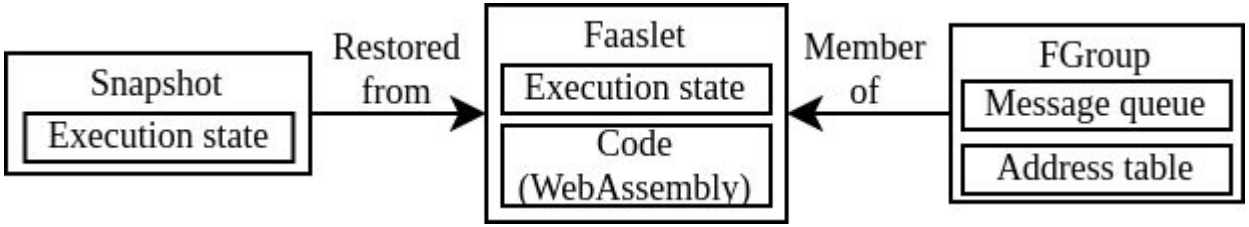
Deep Learning Applications

Porting MPI and OpenMP Apps to Faasm

- To run MPI or OpenMP application in Faasm:
  1. Cross-compile to WebAssembly (with undefined symbols like `MPI_Init`)
     http://github.com/faasm/cpp

  2. Link against custom message passing/shared memory implementation
     http://github.com/faasm/faabric

- **Faabric** library provides two key abstractions for shared memory and message passing:
  - **Snapshots** for checkpointing OpenMP threads
  - **FGroups** (Faaslet groups) for MPI communication using message pasasing



FaasMP Intercepts OpenMP Calls at Runtime

```
int main () {
      int n_threads = 0;

      #pragma omp parallel
      {
            n_threads += 1;
      }

      return 0;
}
```

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

int main () {
      int n_threads = 0;

      __kmpc_runtime_fork(
            parallel_section,
            &n_threads
      );

      return 0;
}
```

Convert this to Faaslet serverless function

Intercept this call

Use this function pointer

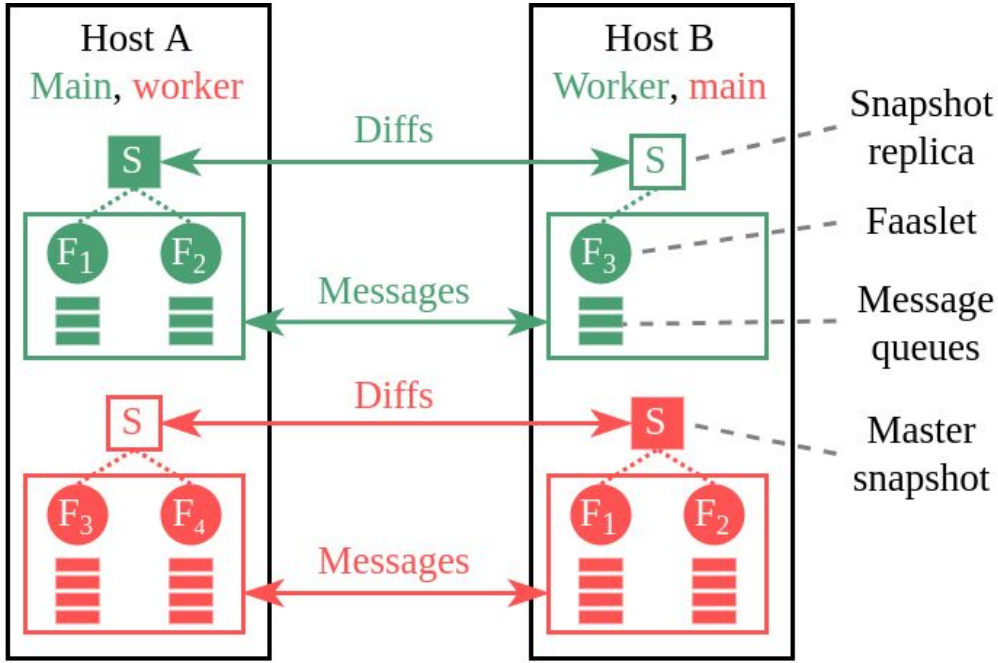
Determine shared data

Serverless Shared Memory and Message Passing

- **Shared memory support:**
  Faaslets "forked" from other faaslets by replicating parent's snapshot

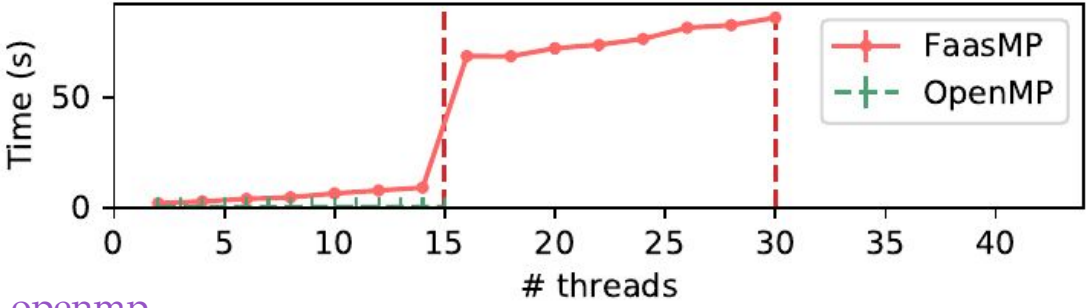
- Efficient **synchronisation** using byte-wise *Diffs* between snapshots

- **Message passing support:**
  Faaslets have unique virtual address within group, and can asynchronously send messages



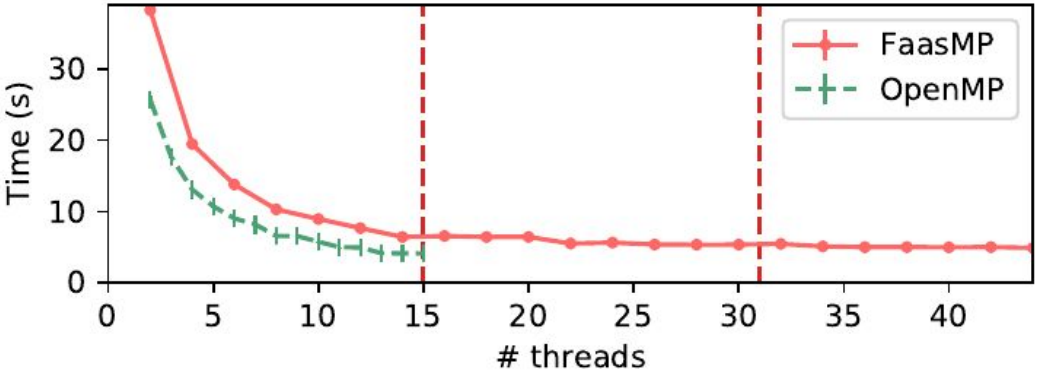
KPI: Scalability of OpenMP applications

LULESH
https://github.com/LLNL/LULESH



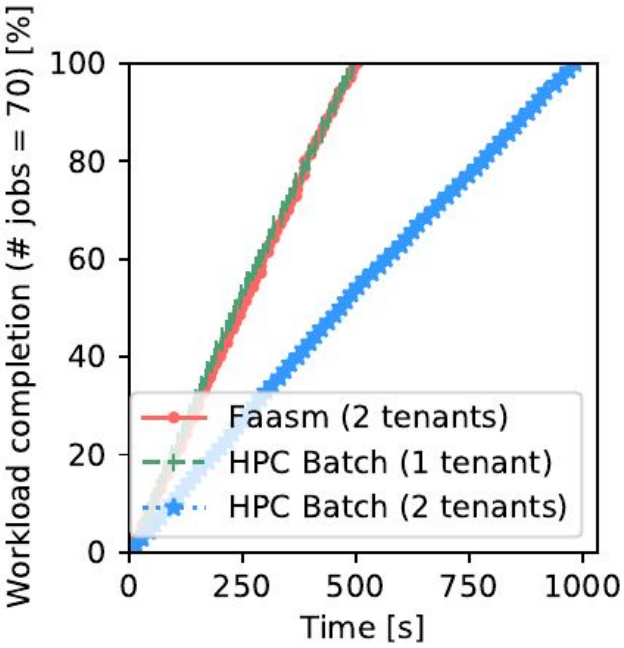
https://github.com/faasm/experiment-openmp

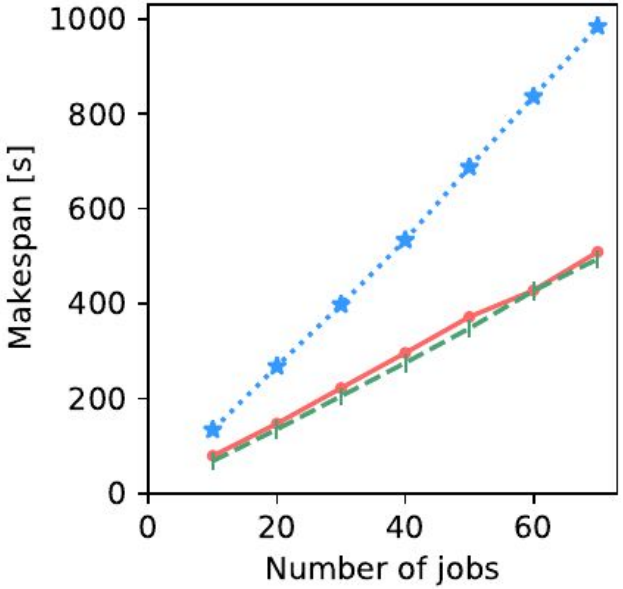
Dense Matrix Multiplication
https://github.com/ParRes/Kernels



KPI: Efficiency with Multiple Tenants

- Faasm has considerably better performance over shared resources compared to commercial batch cloud solutions



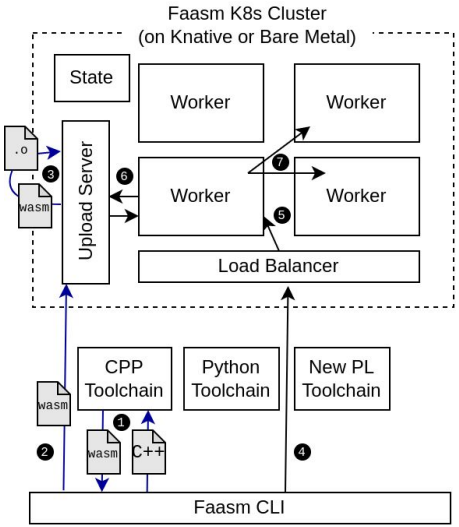


Summary:
Faasm - Lightweight Serverless with WebAssembly

- Increases serverless provider's **efficiency** through **lightweight isolation** using **WebAssembly**
- Adds **language-independence** for serverless applications **(C, C++, Python, JavaScript, Go, Rust, …)**
- **Supports simple porting** of existing applications through POSIX interface

Faasm: High-performance multi-tenant WebAssembly serverless runtime

- **Faaslet** abstraction for executing stateful serverless functions as WebAssembly threads
- Implements standard **WASI interface** for POSIX compatibility
- **Two-tier state model:** global & local state tiers with synchronisation



- Large ecosystem on **GitHub** (over 680 stars) https://github.com/faasm
- Integration with **Kubernetes**, **KNative** & **Lithops**
- More efficient than **Docker** containers for serverless

CloudButton KPIs: Efficiency & scalability for big data applications

BIG DATA VALUE
PUBLIC-PRIVATE PARTNERSHIP

CloudButton

UNIVERSITAT ROVIRA i VIRGILI

Imperial College London

EMBL

Institut Mines-Télécom

Red Hat

fundaciónMATRIX
INVESTIGACIÓN Y DESARROLLO SOSTENIBLE

The James Hutton Institute

Atos

IBM

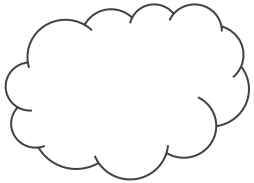
THANK YOU!

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825184.

Backup Slides

Serverless Programming: 2020 - 2022

2020



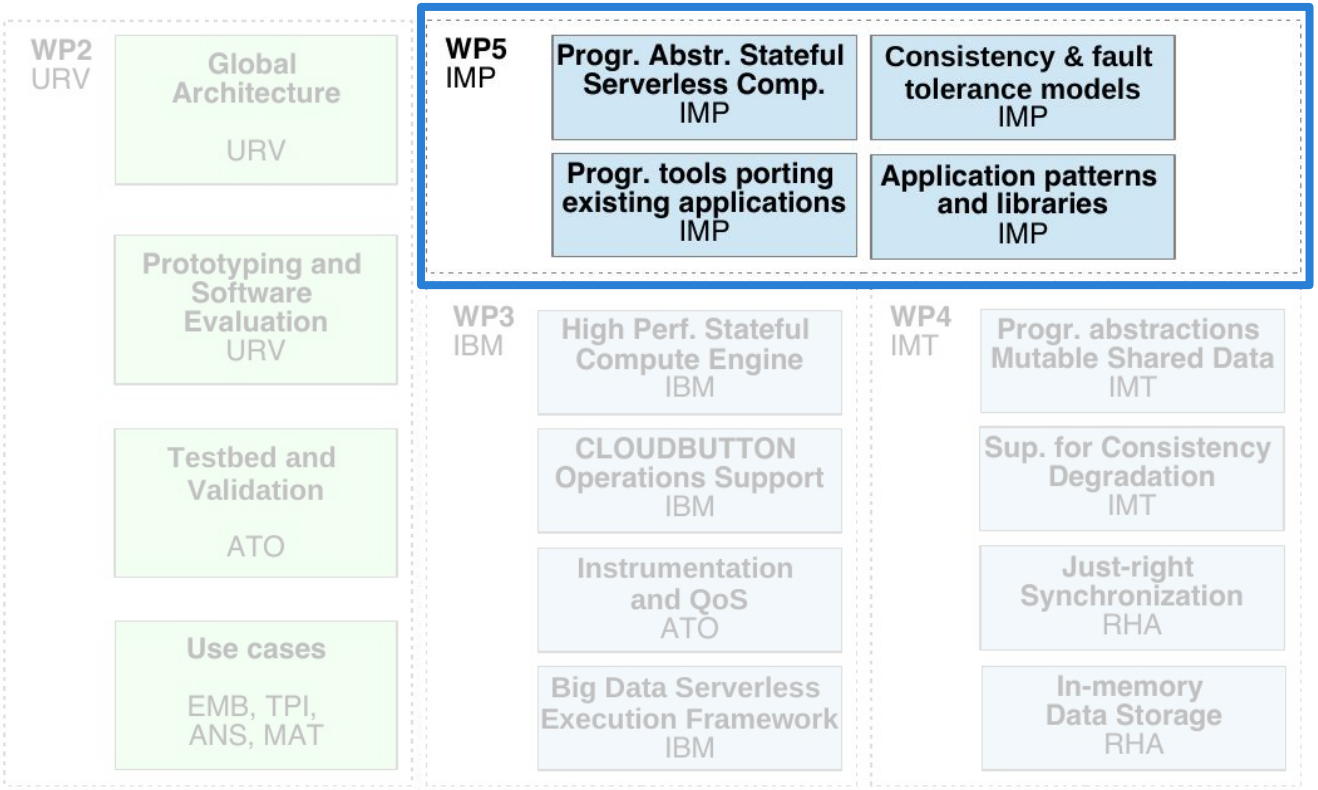
Existing work

- **Storage** (Pocket, Shredder)
- **Scheduling** (Core-granular)
- **Big Data** (PyWren)
- **ML** (Cirrus)
- **State** (SAND, Cloudburst)
- **Snapshot/Restore** (SEUSS)

Existing work

- **Actor-based** (PLASMA)
- **Cloud-Threads** (Crucial)
- **Func Execs** (Lithops)
- **DDO** (Faasm)

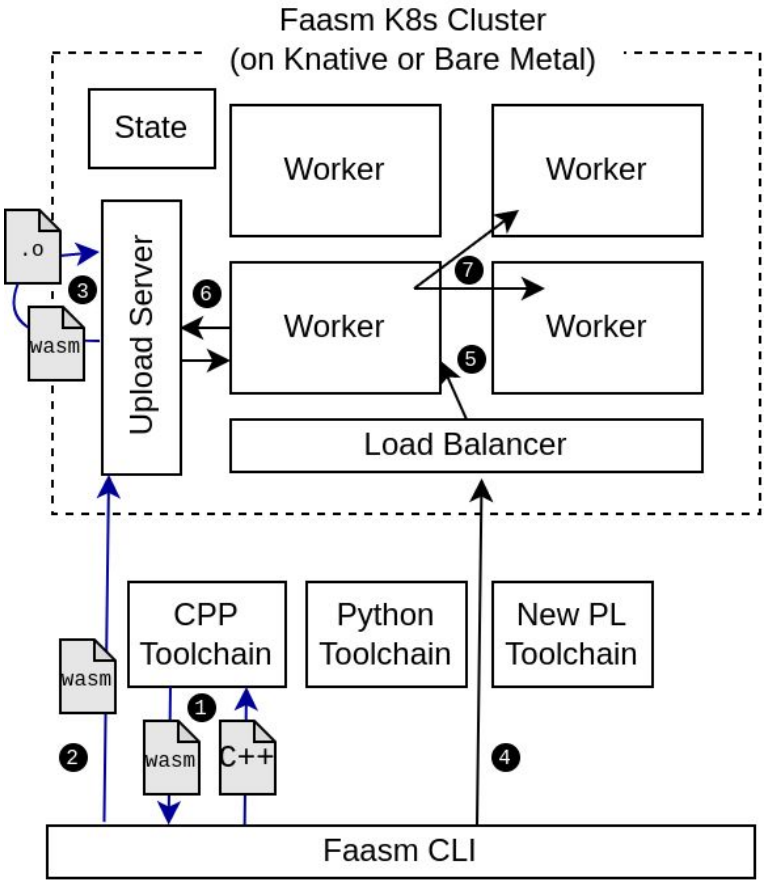
CloudButton Work Package Structure



Faasm Ecosystem

- Faasm: High-performance serverless runtime
- Faasm ecosystem consists of many components (white boxes)
- Faasm components maintained independently and released as Docker containers
- Faasm deployed on K8s and can integrate with Knative
- Low-hanging fruit: new PL (Go, Rust), new/better WASM runtime (WAVM, WAMR, WAMR + SGX)

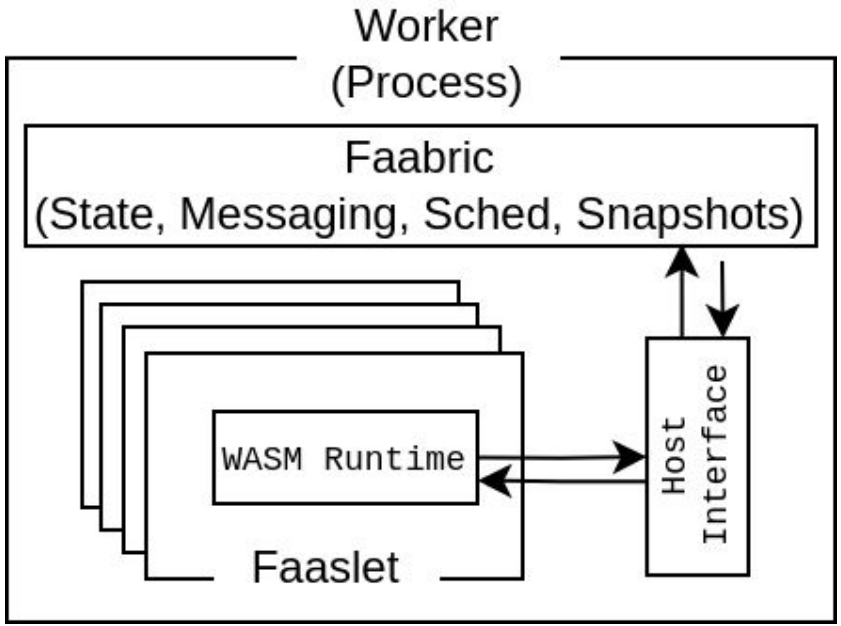
https://github.com/faasm



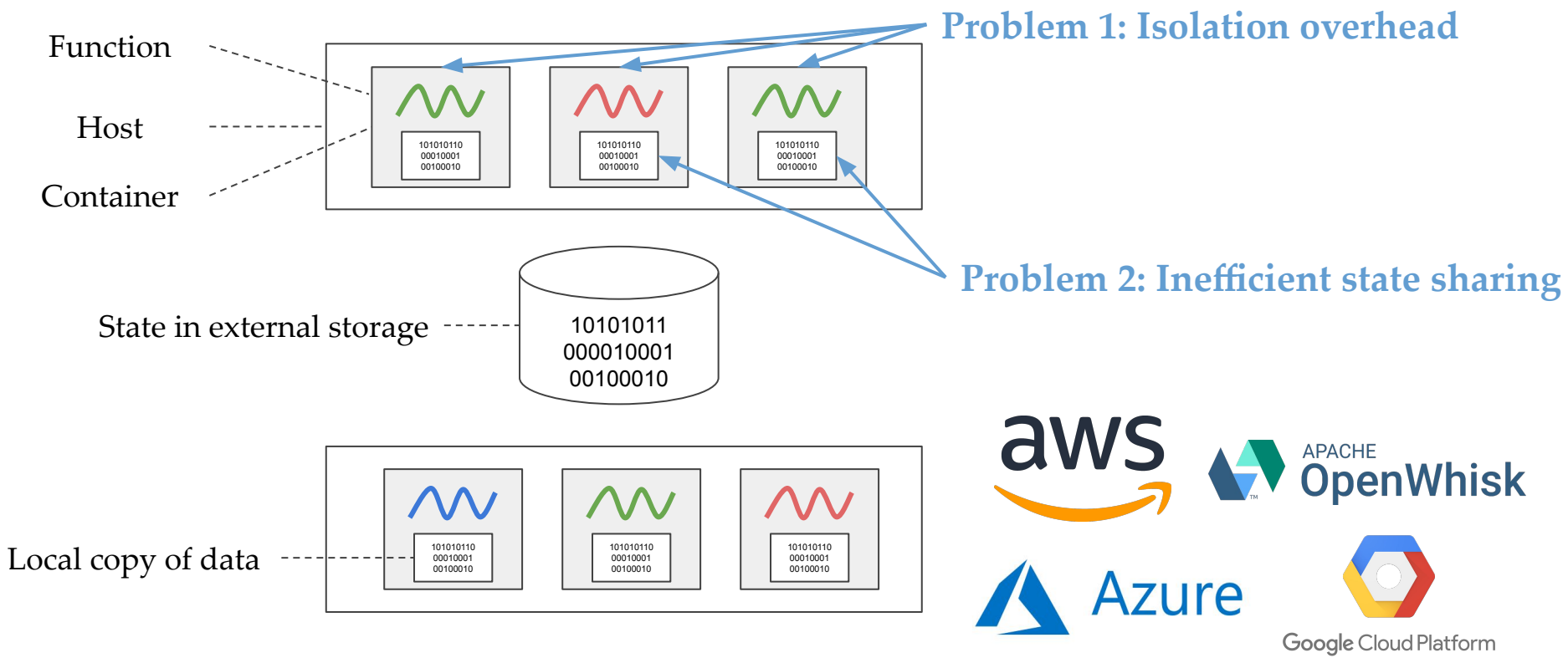
Consolidating Faaslet Abstraction

- Each worker runs in process (one per host) and has pool of Faaslets
- Faaslets are "hardened threads" that run WebAssembly code using a supported runtime (WAVM, WAMR)
- Faasm-generated WASM leaves many imported systems unresolved
- Faasm's host interface provides implementations for these symbols
- All heavyweight logic is off-loaded to new library component: Faabric

https://github.com/faasm/faabric



Problems With Container-based Serverless



Images: AWS , Azure , GCP , OpenWhisk

Reducing Isolation Overhead with WebAssembly

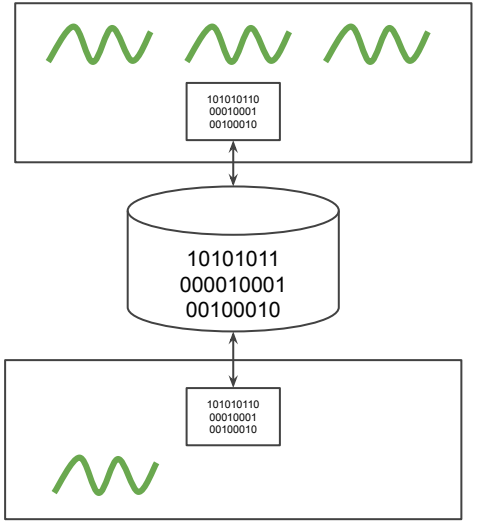
WebAssembly

- Lightweight memory safety
- Used by Fastly, Cloudflare, Krustlet

Benefits vs. Containers:

- Reduced isolation overhead
- Fine-grained control over memory

Sharing State Locally, Synchronising Globally



Local tier
Shared memory

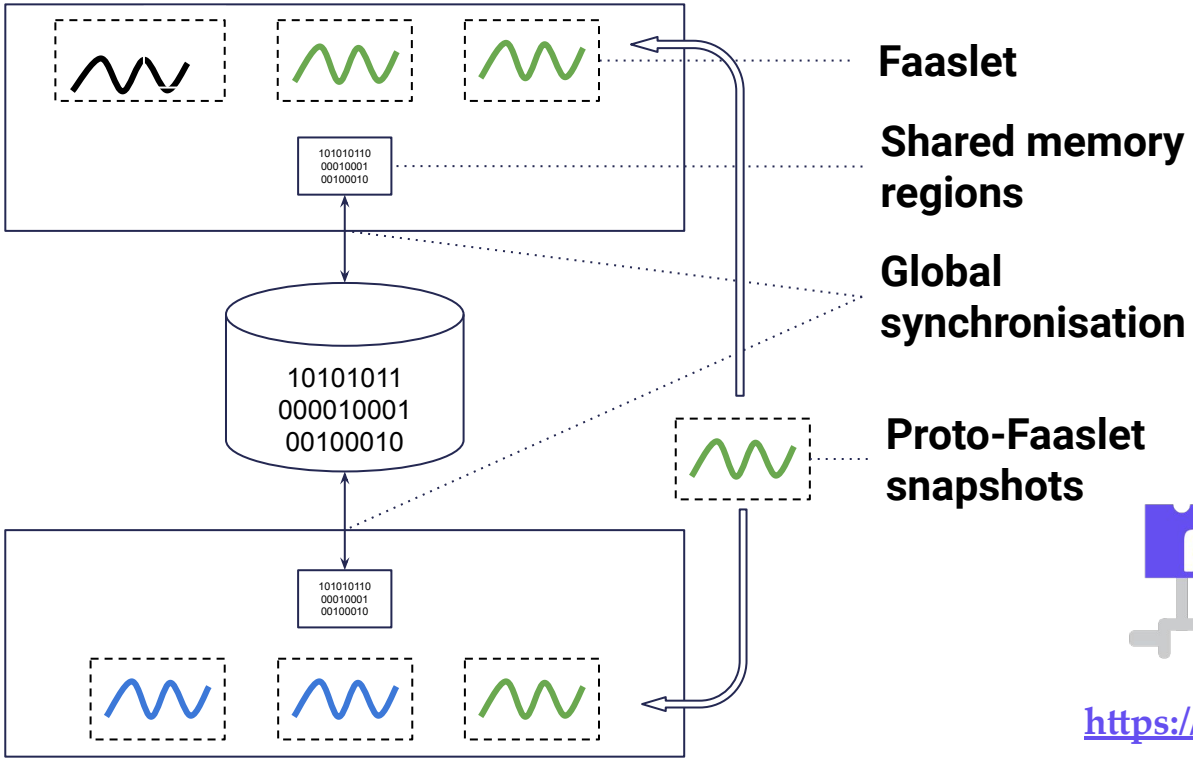
Global tier
Cross-host synchronisation

Two-tier state

Benefits:

- Local access to single copy of data
- Scaling to multiple hosts

Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing



https://github.com/lsds/Faasm

FAASM: Distributed Data Objects

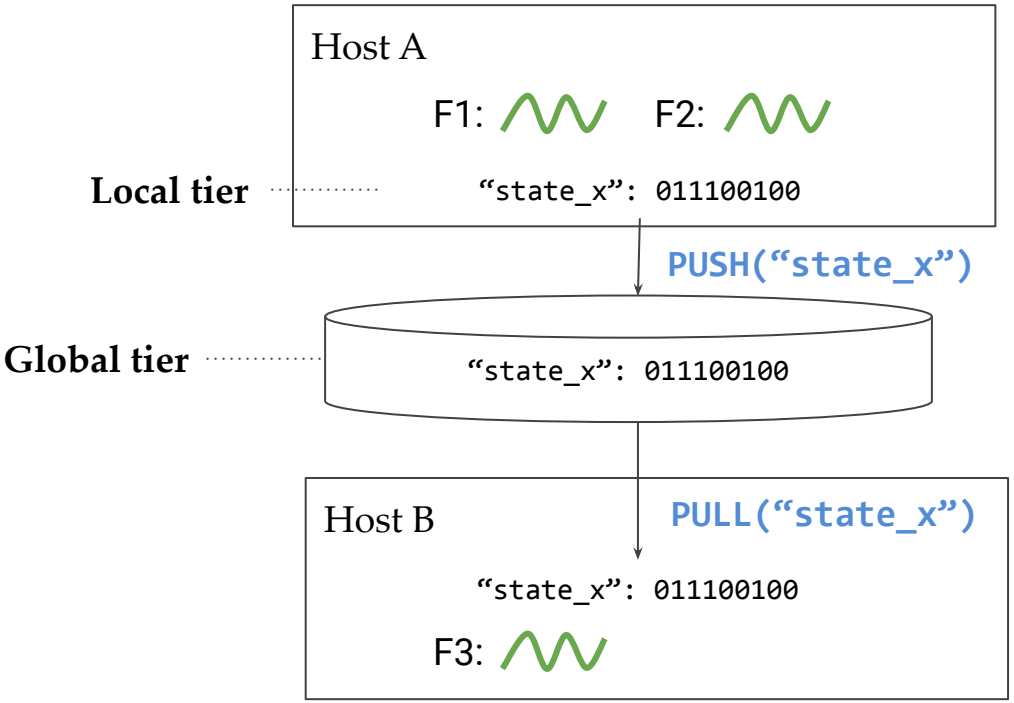
```
t_a = SparseMatrixReadOnly("training_a")
t_b = MatrixReadOnly("training_b")
weights = VectorAsync("weights")

@serverless_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()

@serverless_func
def sgd_main(n_workers , n_epochs):
    for e in n_epochs:
        args = divide_problem(n_workers)
        c = chain(weight_update, n_workers, args)
    await_all(c)
```

High-level Object-Oriented abstractions
Read-only matrices
Asynchronous vector
Flexible consistency

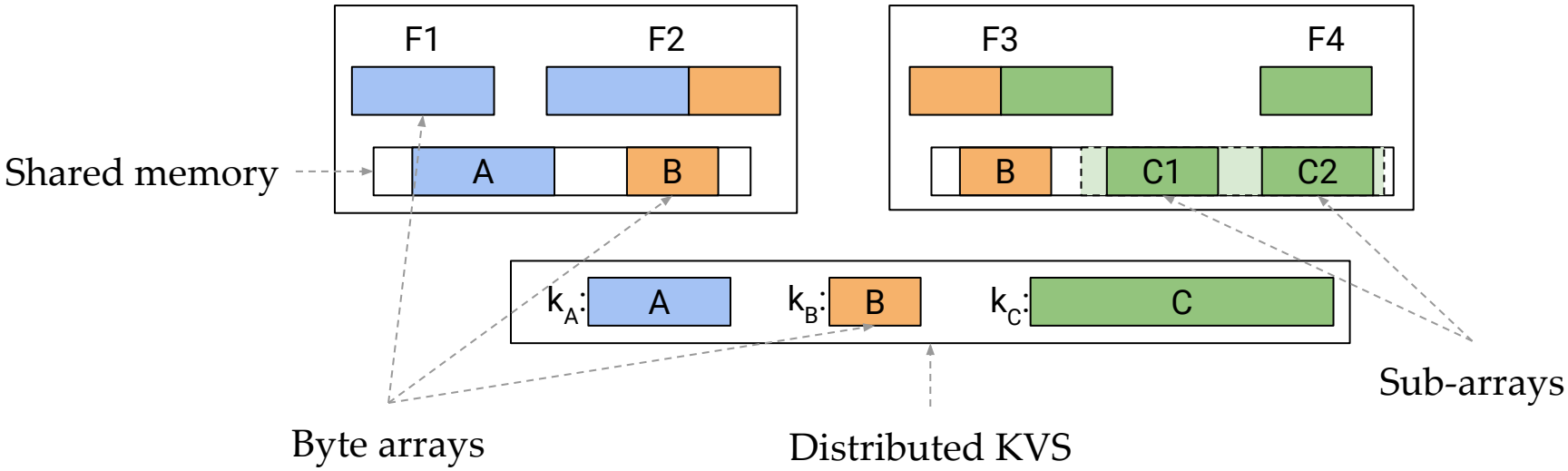
Standard Programming constructs
Transparent optimisations
Direct access to shared memory

Intuitive mark-up
Function annotation
Fork-join parallelism

FAASM: Global Synchronisation with Variable Consistency



FAASM: Serialisation-Free Transfers



Serialisation-free transfer of arbitrarily complex data structures

Serverless Fault Tolerance

Per-function FT annotations:

- `@run-only-once`
- `@run-at-least-once`
- `@fail-on-fault`

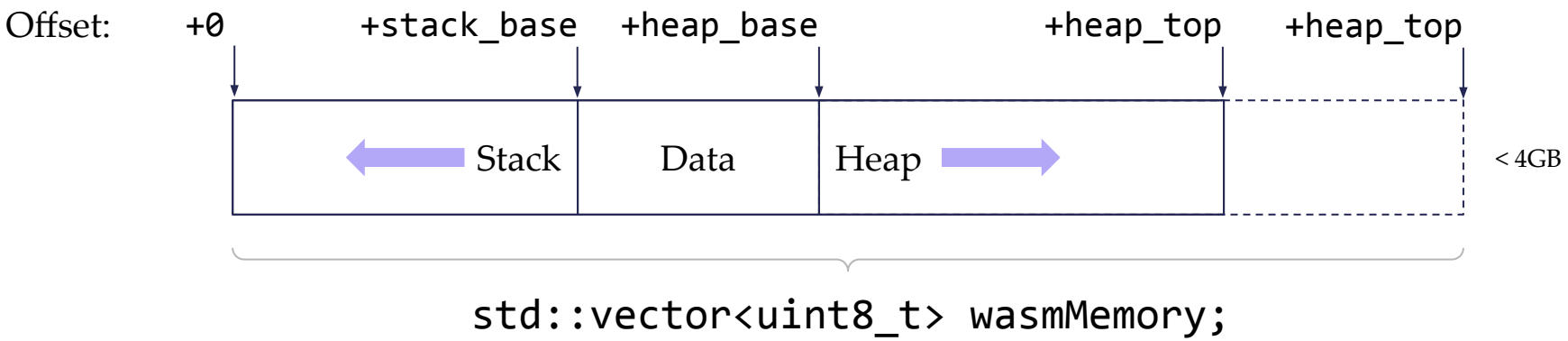
Serverless Computation Graph:

- Dynamically construct graph of chained functions
- Replay inputs on failure

Atomic state operations:

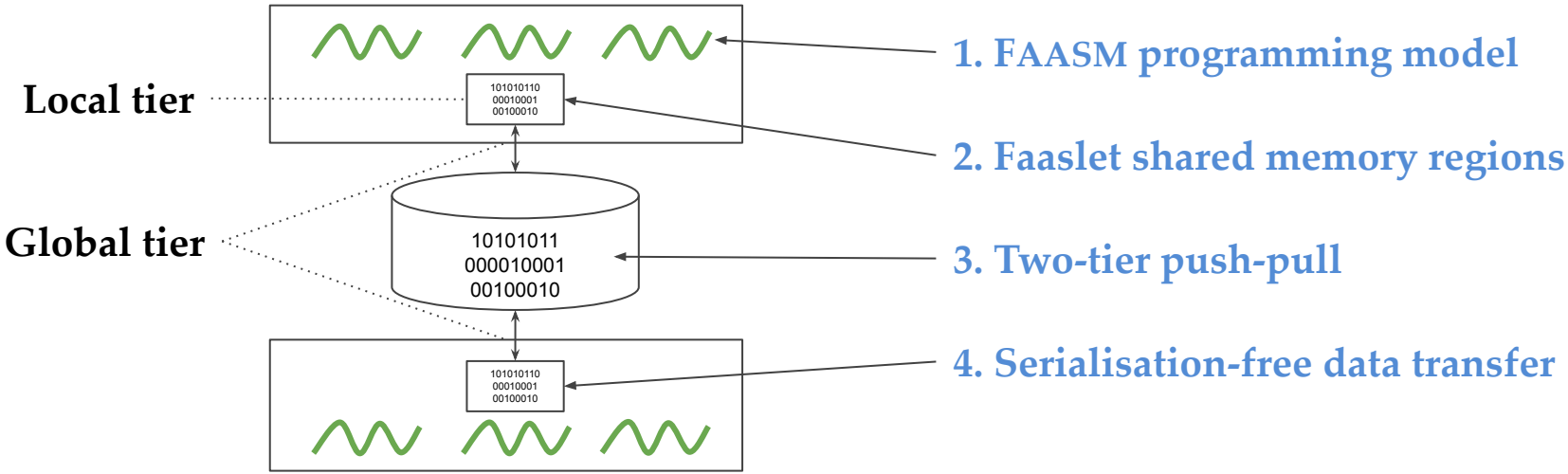
- New state API functions: `atomic_state_read(), atomic_state_write()`
- State logging and revert on failure

WebAssembly: Fine-Grained Memory Safety



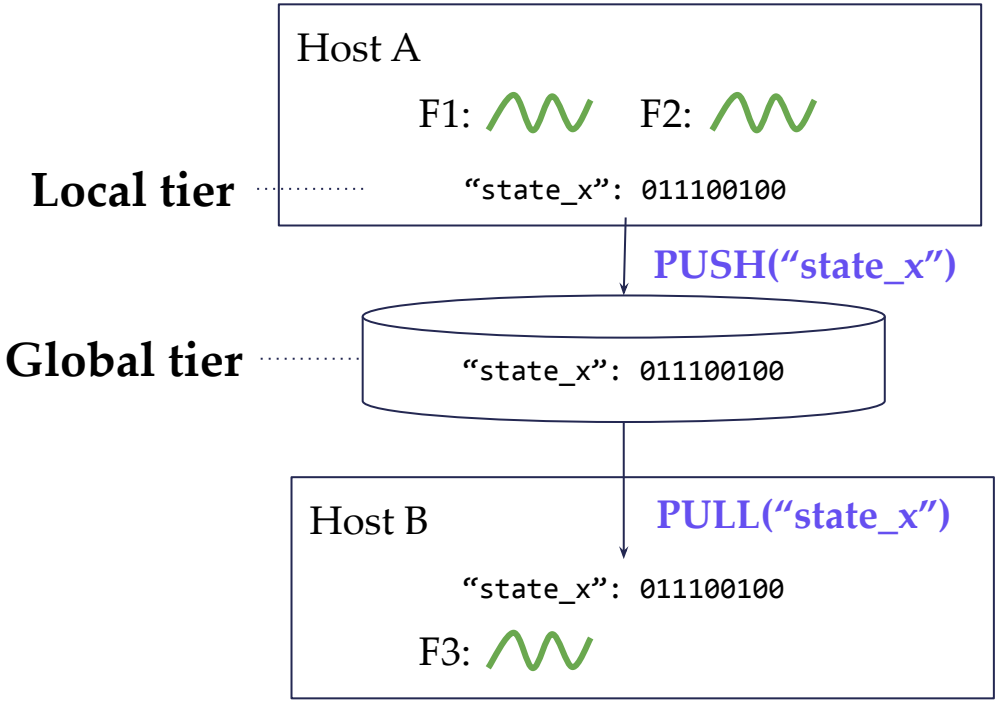
WebAssembly memory model: Simple linear address space

FAASM - Two-Tier State



Distributed architecture to support local sharing and global synchronisation with flexible consistency

State Synchronisation between Local/Global Tiers



Support for different **consistency models** between state tiers

Two-tier push/pull for state synchronisation

Flexible OS Interaction

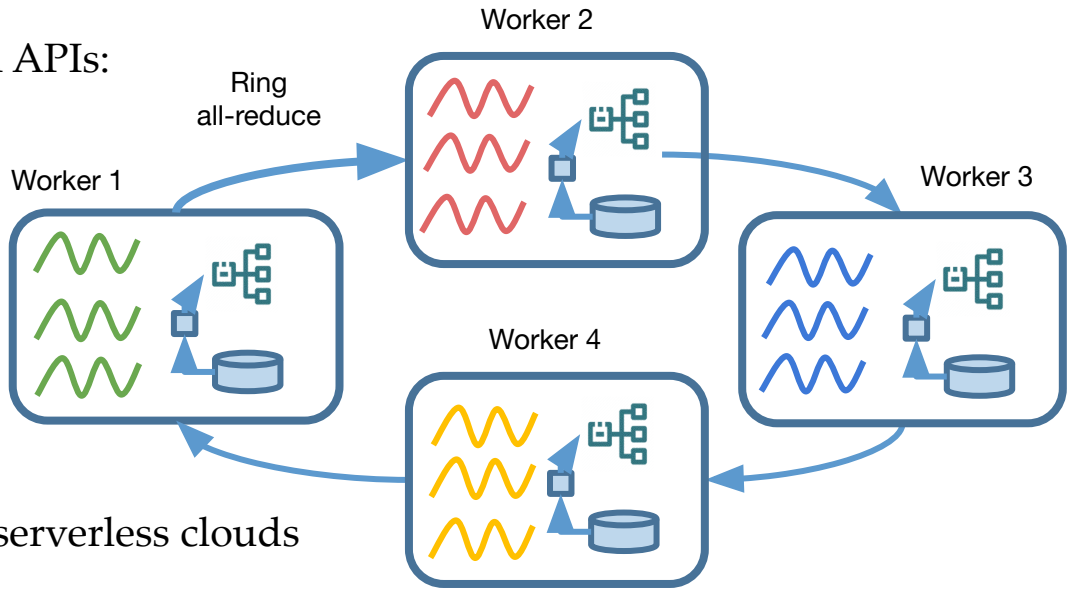
Faaslet host interface

Category	Sub-category	API
Serverless	Chaining	chain_call(), await_call(), ...
	State	get_state(), set_state(), ...

Faaslets supports serverless and POSIX APIs for ease of porting

Example: Collective Communication APIs

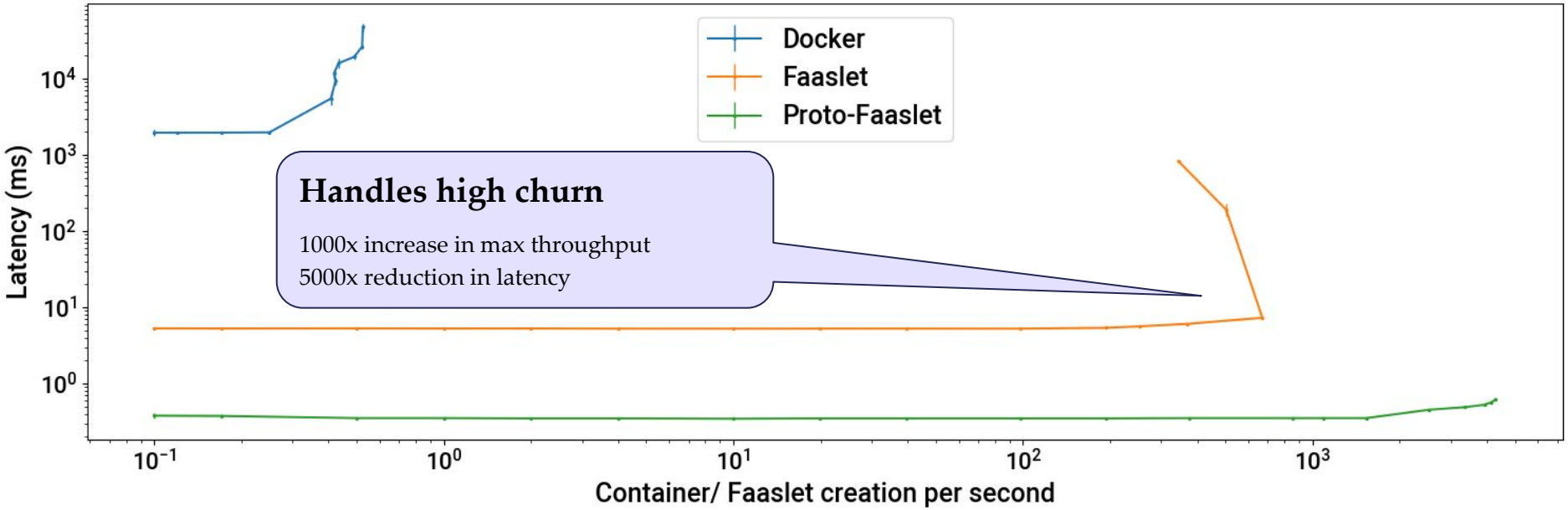
- Support message passing between Faaslets
  - Synchronous & asynchronous communication
- Provide range of communication APIs:
  - Multicast & broadcast
  - Distributed aggregation
  - Synchronisation barriers
- Example: all-reduce for ML



- Support for MPI applications in serverless clouds

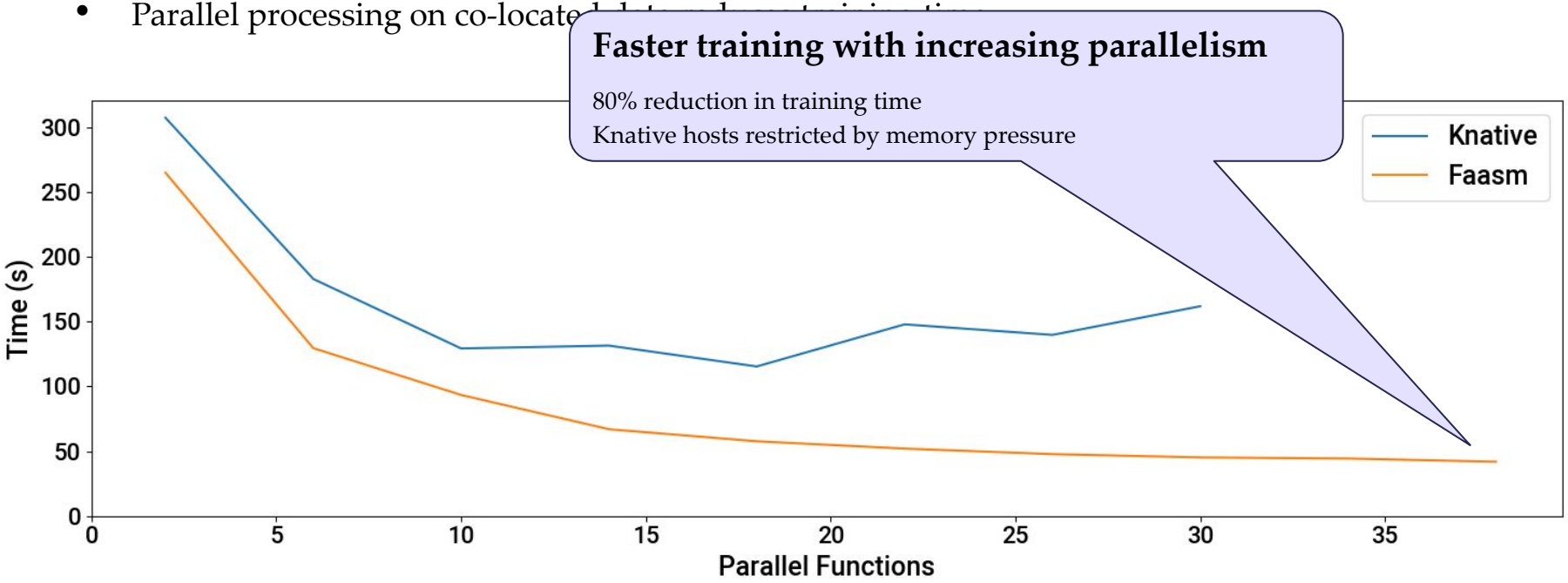
How do Faaslets Compare to Containers?

- Higher churn means higher utilisation of shared infrastructure



Can Faaslets Improve Machine Learning Training?

- Parallel processing on co-locate

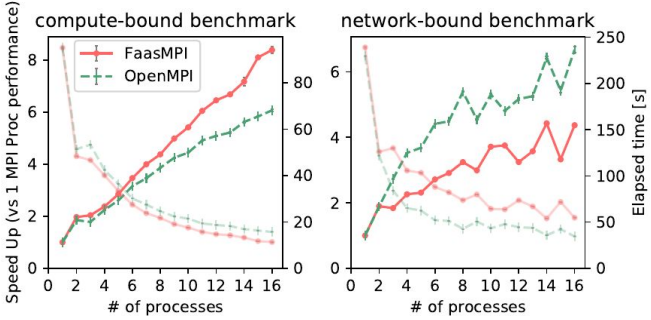


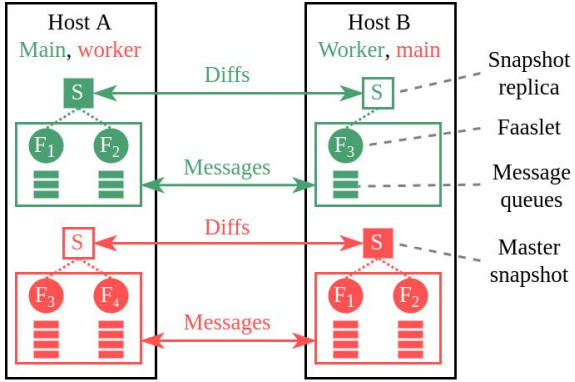
Faabric: Support for HPC Applications

- Supports **shared memory** & **message passing** abstractions for serverless
- Implements **OpenMP** & **MPI** interfaces for data & compute intensive HPC applications
- Provides **scheduling**, **state management**, and **snapshots** using Faasm

Faabric library for distributed shared memory & message passing on serverless

- Two key abstractions: **Snapshots** & **FGroups**
- For **shared memory**, Faaslets are forked and their state is synchronised using diffs
- For **message passing**, Faaslets have group address for asynchronous communication

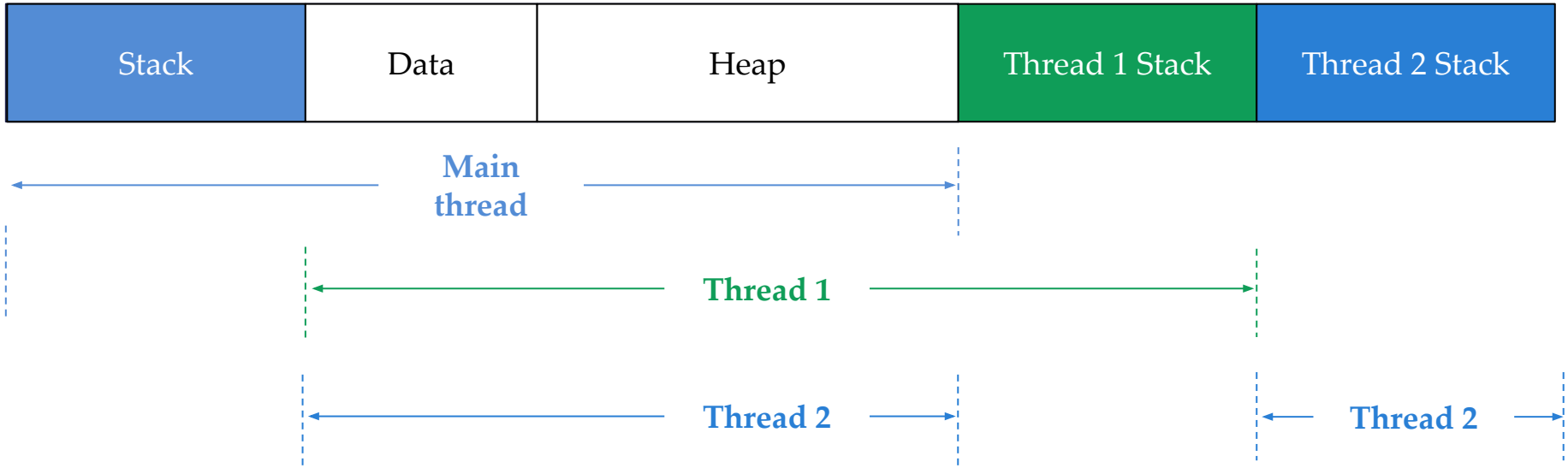




- Available on **GitHub** as Faasm extension: https://github.com/faasm/faabric
- Better **performance** compared to commercial batch cloud solutions

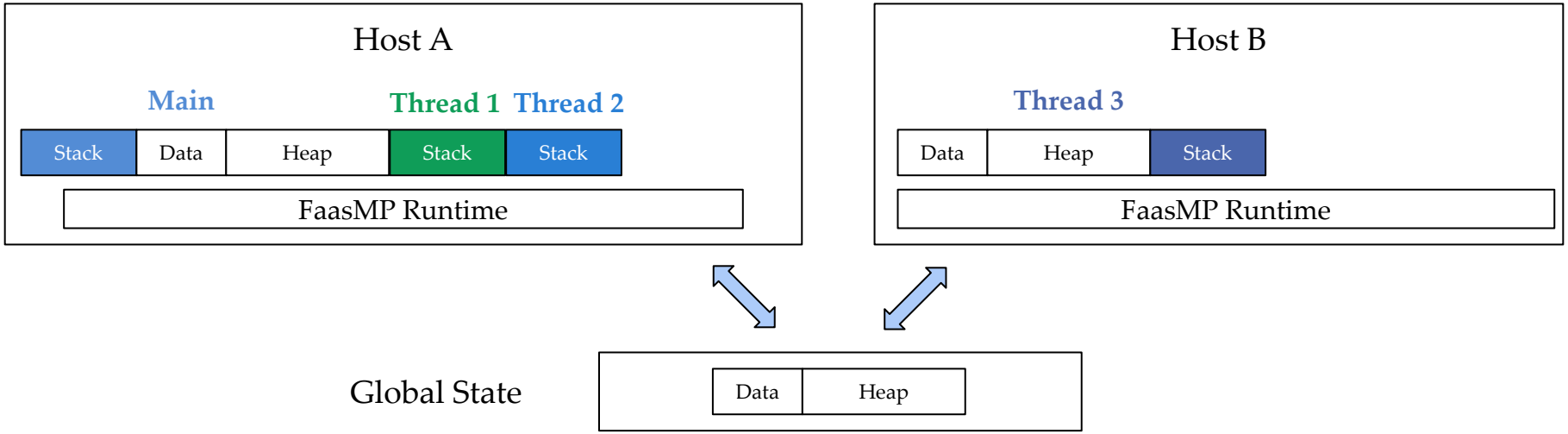
□ CloudButton KPIs: Easy-of-use & transparent HPC application support

FaasMP: Local Thread Memory Model



Easy to manipulate, simple memory model

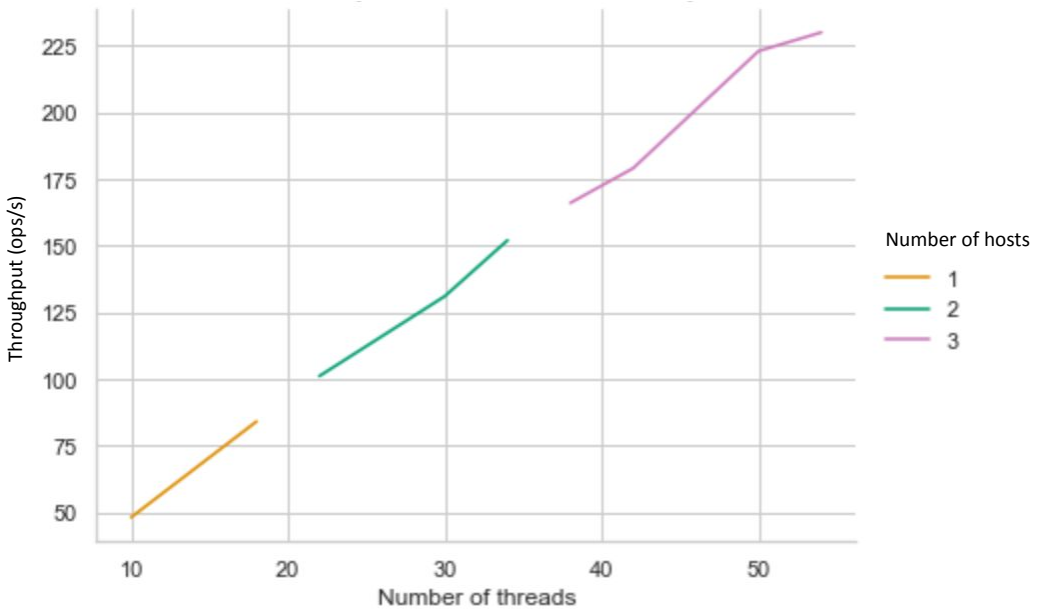
FaasMP: Distributed Thread Memory Model



Distributed thread memory via global state

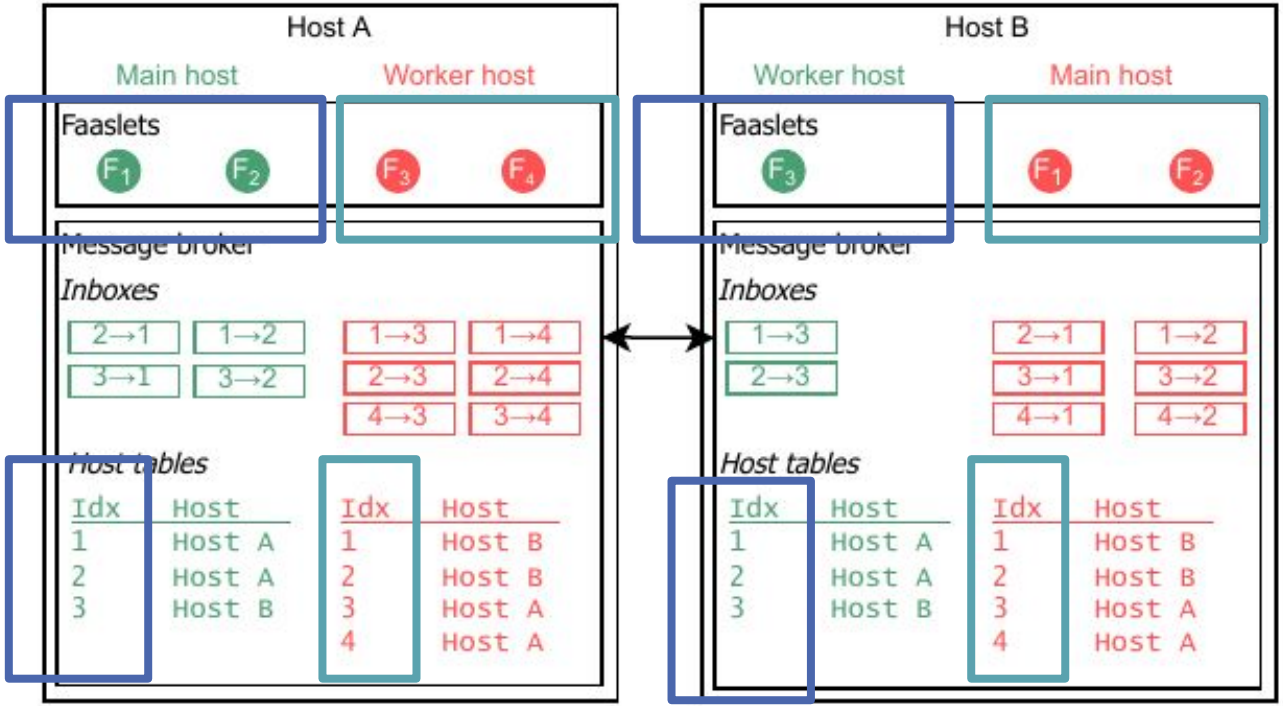
FaasMP: Performance (Distributed Reduce)

Approximately linear scaling across hosts



Message Passing for Faaslets

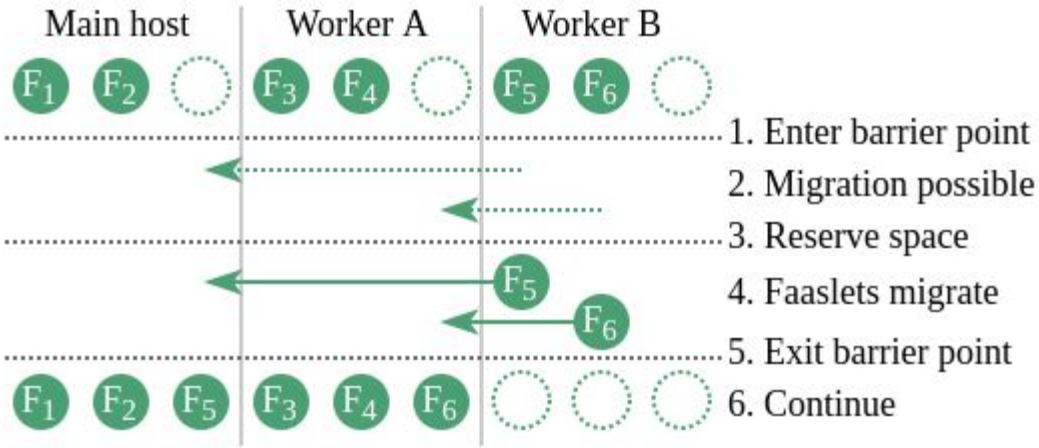
FGroup 1



FGroup 2

Overcoming Message Passing Overheads

- **Problem:** Bottleneck in message-passing application is network
- **FaasMPI's solutions:**
  - Co-located Faaslets use intra-process shared memory (see D5.3)
  - Faaslets can be migrated at runtime to improve locality
  - Optimised collective communication algorithms (see D5.3)



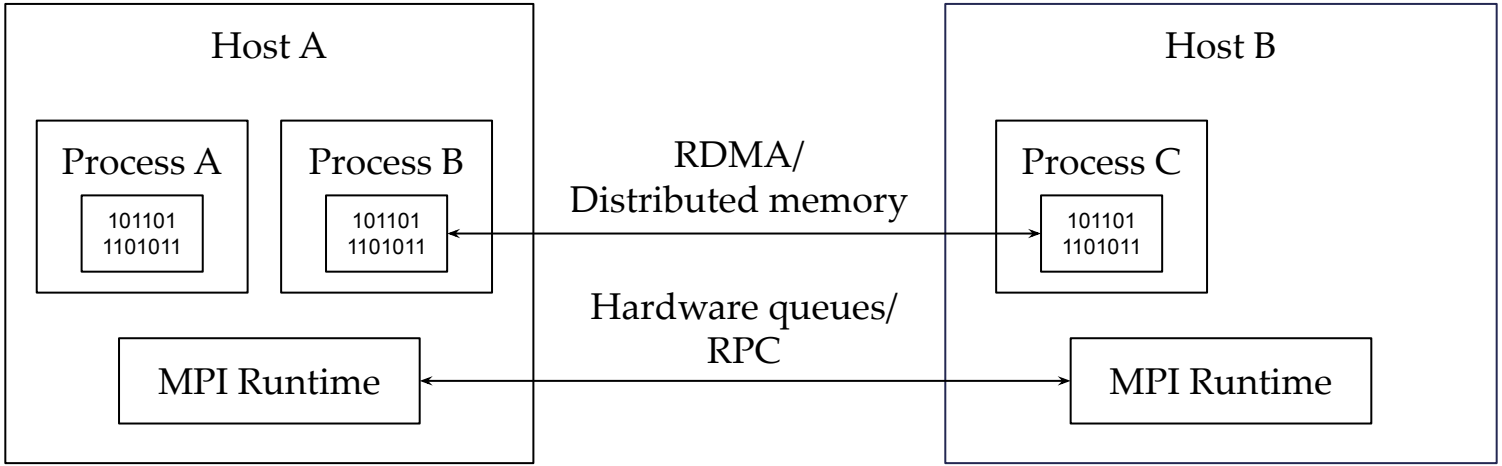
FaasMPI: Serverless MPI

Category	*Examples*	*Purpose*
Point-to-point	`MPI_Send, MPI_Recv`	Synchronous send and receive
	`MPI_Isend, MPI_Irecv`	Asynchronous send and receive
Collective	`MPI_Bcast, MPI_Alltoall`	Broadcast, all-to-all
	`MPI_Reduce, MPI_Allreduce`	Reduce
	`MPI_Scatter, MPI_Gather`	Scatter/ gather
Remote Memory	`MPI_Win_create, MPI_Win_free`	Create/ free remote memory region
	`MPI_Get, MPI_Put`	Read/ write to remote memory

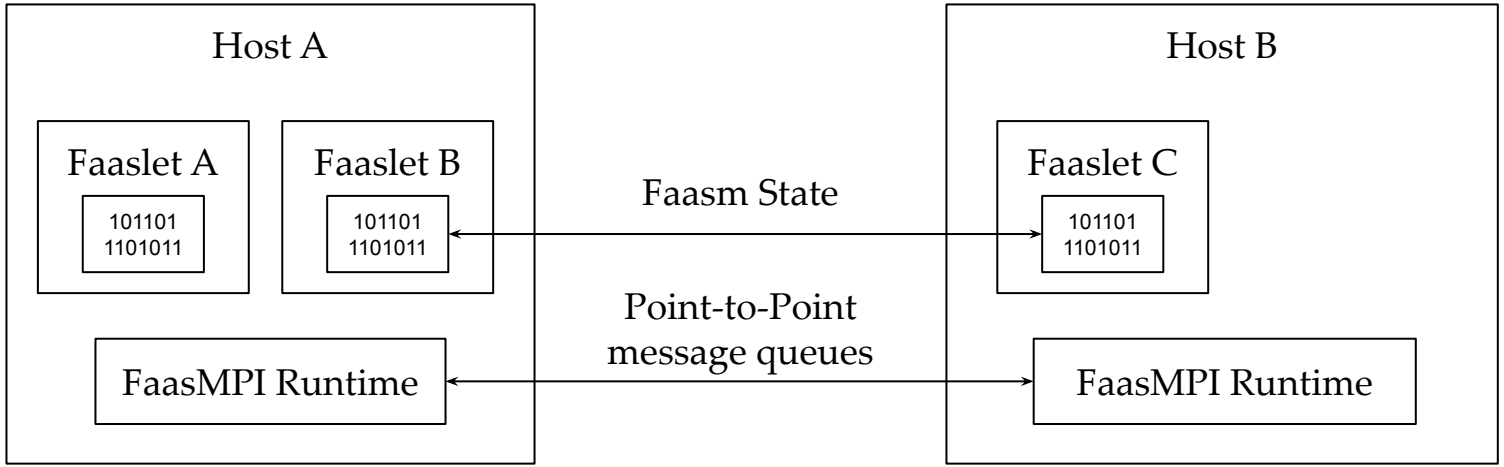
Small subset of MPI supports majority of open-source MPI codebases

Large-Scale Study of MPI Usage, Laguna et al. Supercomputing ‘19

FaasMPI: Traditional MPI Implementation



FaasMPI: Serverless MPI



Can we compete on raw performance?

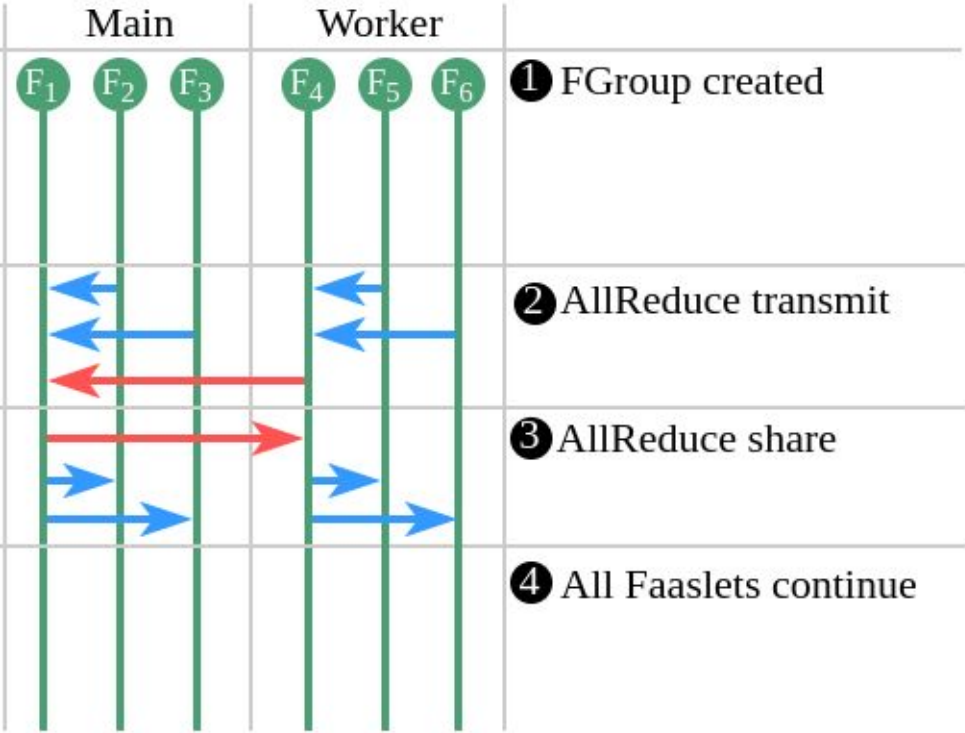
Yes, with the right hardware

Can we compete on scale?

Yes, if distributed state is fast enough

Example: MPI

```
MPI_Init();
int[] w= initWeights();
for(int i=0; i<steps; i++) {
  updateWeights(w, r, n);
  MPI_Allreduce(w, n);

  if(rank == 0)
    applyWeights(w);
}
```

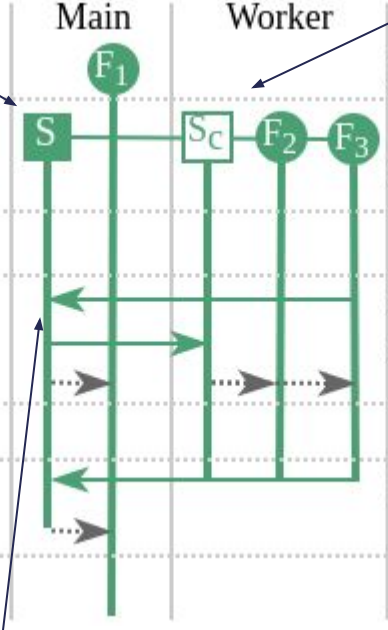


Synchronising Shared Memory in Faasm

Snapshot: map of linear memory of Faaslet.

Fork: Faaslets can be "forked" by copying their parents' snapshot

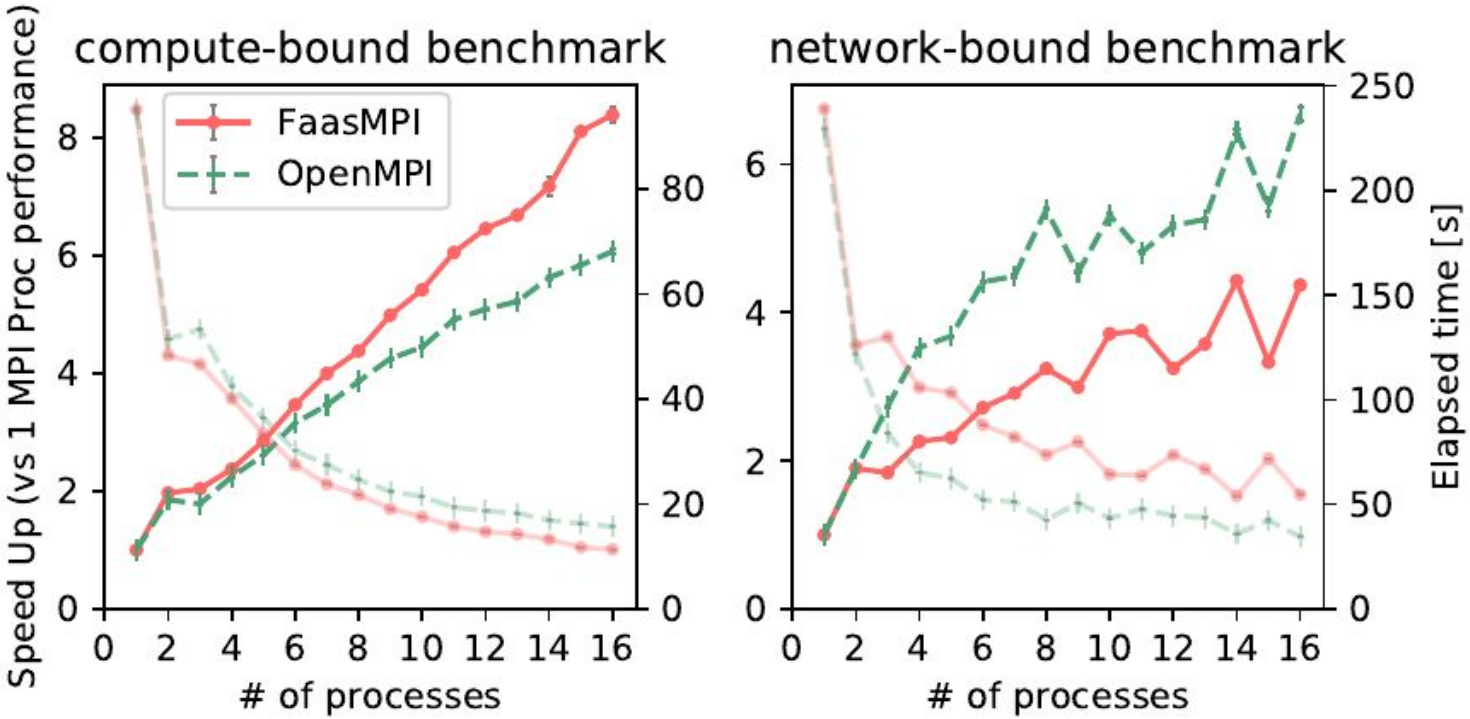
```
int* w = init();
#pragma omp parallel
{
 updateWeights(i, w);
 #pragma omp barrier
 applyWeights(i, w);
}
finaliseWeights(w);
```



1. F_1 at parallel section
   Push snapshot S to S_c
2. Restore $F_{2,3}$ from S_c
   Execute $F_{1,2,3}$
3. At barrier, update S/S_c
4. Remap $F_{1,2,3}$
   Execute $F_{1,2,3}$
5. $F_{2,3}$ finish, update S
   Remap F_1
   Execute F_1

FDiffs: snapshot copies can be synchronised with byte-granularity at syncrhonisation points

KPI: Performance with MPI Big Data Apps

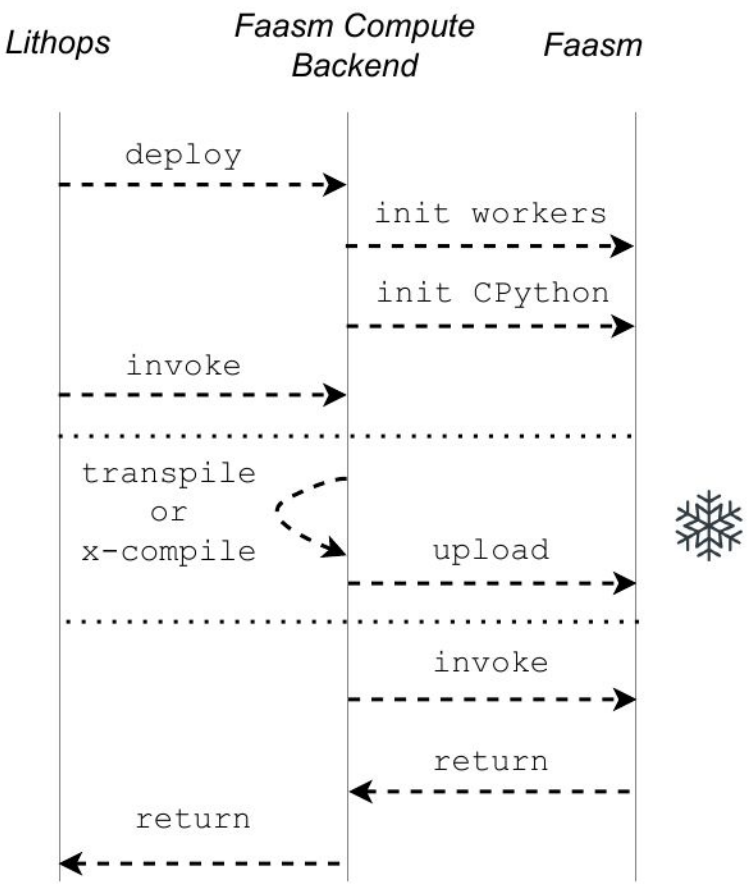


https://github.com/lammps/lammps / https://github.com/faasm/experiment-mpi

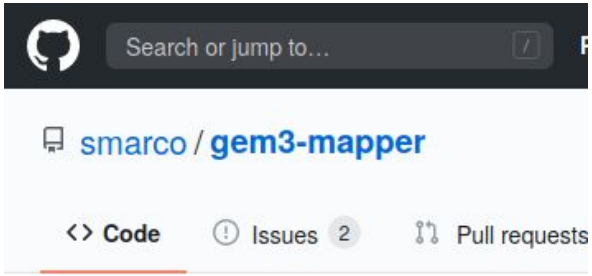
Lithops Functions with Faasm backend

- Faasm executes functions cross-compiled to WebAssembly
- Faasm provides first/class support for functions written in C/C++
- Faasm can run Python functions by cross-compiling the Python runtime, and interpret non-WASM Python functions
- First time (cold) Lithops runs new Faas function, backend will have to cross-compile (Python or C/C++) and upload it
- Successive invocations (hot) cache WASM code

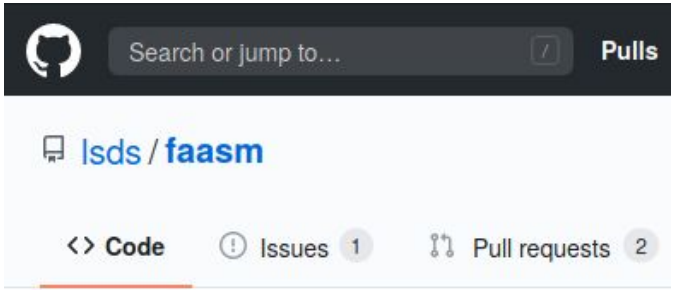
https://github.com/faasm/python



Supporting the Genomics Use Case



+



= simple, distributed mapping

Porting effort:

- Multi-threaded C codebase (simple pthreads)
- Added approx 60 lines of new C code to wrap in Faasm function
- Modified Makefile to use Faasm toolchain

WP5 + Genomics Mapping Phase

Genomics mapping phase (5 stages):

1. Create, chunk up and upload index — Client-side
2. Chunk up and upload input data — Client-side
3. **Map read chunks to index chunks — Faasm**
4. **Aggregate mappings — Faasm**
5. Retrieve results — Client-side

Genomics Mapping