Native DAG Execution in OpenWhisk

Harshita Gupta
Harvard University
hgupta@college.harvard.edu

Eric Lu
Harvard University
ericlu01@g.harvard.edu

Abstract

Function-as-a-service offerings, such as AWS Lambda, are becoming increasingly common building blocks for server-side computation. Instead of running computations on one's own VMs or physical hardware, "serverless" functions are combined to create the building blocks of complex computations on matrices and large data-sets. These uses of serverless functions, however, are constrained by the limited orchestration and function composition abilities of serverless infrastructures.

We modify Apache OpenWhisk to interpret a domain-specific language, DAGular, for serverless orchestration and function composition. DAGular allows programmers to specify data dependencies between serverless function invocations in a functional syntax. We execute DAGular code as a directed acyclic graph of computations that allows us to optimize function invocation order and scheduling in the OpenWhisk controller. DAGular and its accompanying OpenWhisk executor satisfy the serverless trilemma and go beyond, providing native orchestration that is dynamic, highly parallel, and easily programmable by developers.

1 Introduction

Serverless functions allow developers a low-cost way to deploy and run computations in the cloud without managing VM provisioning. While programmers in the past have had to manage scaling to respond to load bursts and minimize idle time themselves, serverless functions offload the responsibility of burst management and scaling down onto the serverless provider. Since the user is charged per time unit of computation used, idle time is "free" in this model. Moreover, since pricing is not based directly on the number of function instances started, handling load bursts by increasing concurrency is also "free" in this model. This makes functions-as-a-service (FaaS) a desirable model for programmers whose computations exhibit a high amount of variance in load.

As serverless functions are adopted for more complex use cases, programmers start needing to combine the results of multiple distinct serverless function invocations into higher-level workflows. In [1], Baldini et al. propose a serverless trilemma that describes shortcomings in mechanisms for function composition supported by existing serverless function offerings. Namely, Baldini et al. claim that existing serverless function offerings each fail to satisfy one of three key properties: that time spent by a caller function waiting on the result of a callee function is not billed ("no double-billing"); that a function can be used while its source code remains hidden to its user ("black-box constraint"); and that the composition of multiple functions can again be treated as a single function in terms of blocking behavior and return value ("substitution principle").

To address the serverless trilemma, we propose that the serverless function provider expose an expressive orchestration language for controlling function scheduling and composition. Furthermore, to maximize the provider's opportunities for schedule optimization, we propose that this language be interpreted directly by the platform's scheduler. In this paper, we contribute a design of an orchestration language, DAGular, which allows the user to specify computations as function compositions. DAGular can express compositions that take the form of a directed acyclic graph (DAG) whose nodes represent function invocations and whose edges represent data dependencies between invoked functions. As we will discuss, this supports a more diverse range of computations than existing platform-side offerings. We have implemented an interpreter for a subset of DAGular as an extension to OpenWhisk, an open-source serverless computing framework.

Contributions This paper introduces:

1. A domain-specific language, DAGular, that allows a programmer to write a functional program that specifies the relationships between the inputs and outputs of multiple serverless functions in a workflow.

2. A parser that compiles DAGular into dependency information that can be used to schedule function invocations in OpenWhisk.

3. An implementation of a new action in OpenWhisk that triggers an interpreter to execute DAGular programs.

The rest of this paper is organized into the following sections. In 2, we discuss other approaches to orchestration and function composition and compare them to our contribution. In 3, we discuss the architecture of OpenWhisk and explain where our implementation fits in. In 4, we discuss the language specifics of DAGular. In 5, we explain how our implementation interfaces with OpenWhisk. We evaluate expressiveness and discuss planned benchmarks in 6, discuss future research in 7, and finally conclude in 8.

2 Related Work

Some existing serverless function platforms support scheduling for complex function compositions. AWS Step Functions are an AWS-provided service running atop AWS Lambda that addresses function composition with scatter-gather parallelism. AWS Step Functions allow developers to specify an order in which serverless functions should be executed, and also allow for triggering function invocations through other AWS services such as AWS API Gateway and AWS S3 bucket. Microsoft Azure Durable Functions also support stateful function composition with scatter-gather parallelism.

Numpywren [5] highlights one computationally-intensive workload whose communication pattern cannot be expressed directly by either AWS Step Functions or Azure Durable Functions. Numpywren is targeted toward calculations in linear algebra such as matrix multiply and Cholesky decomposition. While these might be highly repetitive, parallelizable, and therefore well-suited to FaaS, parallel implementations of these algorithms as compositions of serial functions exhibit complex communication structure. In particular, result values are multiply reused, and single function invocations refer to different sets of preceding values, sometimes sets that are determined dynamically. This highlights a primary limitation of existing orchestration offerings: they cannot directly support computational workloads with complex or dynamic communication patterns. Numpywren instead develops its own client-side coordination scheme, handling scheduling and message-passing between serverless function instances using a number of other AWS platforms. Like DAGular, numpywren is therefore able to express dependency graphs of function invocations; however, numpywren restricts programmers to computations whose communication patterns can be inferred statically. It also requires the maintenance and programming of an additional live server which coordinates function execution.

Similarly, ExCamera [2] runs two additional virtual machines that launch and communicate between serverless functions that serve as "workers" in a larger video encoding operation.

Jangda et al. propose an orchestration language for serverless functions, Serverless Programming Language (SPL) [3]. SPL implements serial composition of function invocations and simple platform-side operations on returned values, but does not support parallel execution. Jangda et al. also chose to implement SPL on OpenWhisk, adding two additional action types, Fork and Program; compositions are implemented using OpenWhisk's existing Sequence action type. The function composition language itself is not interpreted platform-side; setting up a workflow requires a client to make one request per function composition, representing a significant overhead which increases as the workflow lengthens.

3 Architecture

OpenWhisk is an open source serverless platform, implementing a distributed system for executing serverless functions. OpenWhisk consists primarily of a server ("controller") that distributes work to containers ("invokers"). OpenWhisk uses a CouchDB database to store programs and data. OpenWhisk supports lambda-type function invocations in the form of "actions". Action invocations generate "activations", which identify running actions; the ultimate results are persisted in a database. An OpenWhisk action may execute a script or represent a Sequence of other actions (as proposed by [1]); Sequences themselves are considered actions, thereby obeying the substitution principle.

Actions are created and triggered via a command line interface, WSK, which communicates with the OpenWhisk controller via HTTP requests. Users may create and update actions via HTTP post/put requests; these are stored in CouchDB and retrieved for execution when invoked.

OpenWhisk is deployed on virtual machines using ansible. It can be deployed across a single machine or multiple machines; our development all occured on a single-machine deployment.

```
def cholesky (A, B, N) {
    nblocks = ceil(N/B)
    def for_j(j) {
        L[j][j] = chol(A[j][j]);
        par_for (i in range(j, j + nblocks)) {
           L[i][j] = mul(
            inv(
                L[j][j]), A[i][j]);
        }
        def for_k(k, j) {
            par_for (i in range(k, nblocks)) {
                A[k][l] = sub(
                    A[k][l],
                    mul(
                        T(L[k][j]), L[l][j]));
            }
        }
        par_for (k in range(j, j + nblocks)) {
            for_k(k, j);
        }
    }
    seq_fold (L = A, j in range(0, nblocks))
        {for_j(j)};
    return L;
}
```

Figure 1: A DAGular Program Executing Cholesky Decompositions

4 Design

4.1 DAGular Language Design

DAGular is designed to support complex function compositions and data dependencies that ultimately allow a user to build a computation out of many worker serverless functions. The language syntax is intended to be intuitive and familiar for a developer. Despite an imperative appearance, because the language ultimately describes a dependency graph, it is interpreted functionally; this also renders synchronization unnecessary. While we do not support unbounded conditional looping, our language contains parallel map and sequential reduce, allowing many programs on variable-size data to be expressed and safely executed.

An example DAGular program, Cholesky, is provided in 1. This example relies on the lambda functions `chol`, `mul`, `inv`, `sub` and `transpose`. The example uses subgraph definitions via the `def` keyword, and runs computations in sequence or parallel via `seq_fold` and `par_for`. Apparent updates are functional and scoped; future references will

```
store_results = STORE_METADATA(INPUT["link"]);
rekognition_results = REKOGNITION(INPUT["link"]);
status1 = DB_PERSIST(
    store_results["status"]["code"]);
status2 = DB_PERSIST(
    rekognition_results["status"]["code"]);
insights = rekognition_results["data"];
return {
    "persistence_status": [status1, status2],
    "rekognition_insights" : insights
    };
```

Figure 2: A DAGular program that operates on a file uploaded to S3

pick up the last bound value. This allows the developer's code to remain brief while also retaining the DAGular nature of the dependency structure. Unlike numpywren, DAGular does not attempt to analyze the program for its dependency structure before execution.

DAGular also makes it easy for the programmer to define how partial outputs from one program can be consumed as inputs to another program. 2 shows a simple example of our projection language, which is very similar to that proposed in [3]. The projection features of DAGular allow a DAGular invocation to return a hybrid response to the caller once the results have returned.

5 Implementation

5.1 Parallelism and asynchronity

To maximize parallelism, the DAGular interpreter is implemented using asynchronous callbacks. Variable names refer to results of evaluation or function invocation that may not be complete yet; the final return value is bundled as a typical OpenWhisk activation, such that users may invoke DAGular programs asynchronously or register event handlers for completion. The value of a variable is not retained once all its consuming functions have completed execution; this reduces the memory footprint of interpretation. DAGular expressions that depend on the results of other DAGular expressions or function invocations wait on these results before themselves returning a value. Elements of arrays generated by parallel maps are retained as separate values so that later maps are able to begin work before all elements of the array have been completed. Though the current implementation only invokes a function, allocating a container, once the arguments to the invocation are ready, extensions could pre-allocate containers to reduce cold start overhead.

5.2 DAGular invocation

Our implementation allows for DAGular programs to be created via the OpenWhisk command line interface (CLI) using e.g. the command `wsk action create -dagular newAction file.js`. The provided parameter contains a DAGular program after conversion to JSON representation. After creation, DAGular programs may be executed using normal action invocation mechanisms, including e.g. the command line `wsk action invoke newAction`. All functions invoked by the DAGular code that are not defined subgraphs within the DAGular program scope are assumed to refer to an OpenWhisk action and invoked from the database.

Our implementation of DAGular execution builds on top of existing OpenWhisk function execution from scratch. We register a single activation for the entire DAG in the OpenWhisk action database, and invoke function executions once the action has been triggered.

Our implementation uses significantly less network bandwidth than [3]'s extension to OpenWhisk, primarily because it relies less on the network for activations. [3] constructs an invocation pipeline from multiple sub-actions, rather than through one single action, and slows execution and activation down as a result.

Since OpenWhisk relies on Scala futures to trigger functions, our DAG execution occurs lazily and as functions become available. This requires minimal computation on the controller's part and is handled by built-in Scala language features.

6 Evaluation

Implemeting DAGular support in the OpenWhisk reactive core satisfies the serverless trilemma. Since serverless functions do not call each other, there is no double-billing; since DAGular functions are actions themselves, they satisfy the substitution principle; and since no code of called functions is required DAGular satisfies the black-box constraint. DAGular also satisfies all other metrics outlined in [4]; DAGular supports parallelization, is dynamic, has minimal overhead, and is build directly into the reactive core so that it is not client-side or a serverless function itself.

DAGular is successfully able to execute a Cholesky decomposition without the use of a scheduler or server outside the OpenWhisk controller.

6.1 Planned Benchmarks

Due to trouble getting numpywren to run on OpenWhisk, we were unable to evaluate and compare our implementation of a Cholesky decomposition to numpywren's. Below are some benchmarks we plan to collect in the future:

- Measuring completion time for identical cholesky decompositions of various sizes and block numbers. on both DAGular and numpywren.

- Tracking the number of workers executing at a given time, as a way to measure utilization and parallelization.

- Measuring CPU utilization level on the virutal machines for both numpywren and DAGular – which is working to saturation, and better?

- Measuring how much network overhead is present in passing data between functions?

- Measuring how much database access is adding performance overhead?

7 Discussion and Future Work

Future work on DAGular has much potential. DAG compilation currently occurs within the OpenWhisk controller; future work out move this compilation to the CLI in order to proactively type-check and raise user errors upon DAG creation. Preemptively compiling a data structure of function dependencies from DAGular code would also allow advanced analytics through graph and edge analysis.

Additionally, DAGular can be sped up by adding direct message passing capabilities to the function invocations within OpenWhisk. By allowing functions to communicate directly with each other, complex data dependencies will introduce less performance overhead.

DAGular's small code footprint additionally demonstrates that adding DAG execution to commerical serverless infrastructures is a manageable lift with high payoffs for the user. Our patch to OpenWhisk is only 750 lines of code.

8 Conclusion

We have reviewed the serverless trilemma and the problem of function composition and orchestration, and discussed why existing solutions are either not parallel enough or not dynamic enough. We have proposed and implemented DAGular, a serverless orchestration language that can be natively supported by a serverless controller. We have modified OpenWhisk's controller to interpret DAGular code and execute the corresponding computations.

Our project demonstrates the usefulness and feasibility of orchestration and composition frameworks for serverless functions. By supporting more complex and dynamic workloads, serverless functions make auto-scaling and auto-scheduling features more broadly available to cloud computing users, and bring the promise of "serverless computing" closer. When developers can focus on their computations and led cloud providers handle autoscaling and scheduling, FaaS grows in its capabilities and value.

References

[1] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*, pages 89–103, Vancouver, BC, Canada, 2017. ACM Press.

[2] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. page 15.

[3] Abhinav Jangda, Donald Pinckney, Samuel Baxter, Breanna Devore-McDonald, Joseph Spitzer, Yuriy Brun, and Arjun Guha. Formal Foundations of Serverless Computing. *arXiv:1902.05870 [cs]*, February 2019. arXiv: 1902.05870.

[4] Pedro García López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. Comparison of FaaS Orchestration Systems. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 148–153, December 2018. arXiv: 1807.11248.

[5] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *arXiv:1810.09679 [cs]*, October 2018. arXiv: 1810.09679.