2019 IEEE 5th International Conference on Big Data Intelligence and Computing (DATACOM)

Serverless Skeletons for Elastic Parallel Processing

Stefan Kehrer, Jochen Scheffold, Wolfgang Blochinger
Parallel and Distributed Computing Group, *Reutlingen University*, Germany
firstname.lastname@reutlingen-university.de

***Abstract*—Serverless computing is an emerging cloud computing paradigm with the goal of freeing developers from resource management issues. As of today, serverless computing platforms are mainly used to process computations triggered by events or user requests that can be executed independently of each other. These workloads benefit from on-demand and elastic compute resources as well as per-function billing. However, it is still an open research question to which extent parallel applications, which comprise most often complex coordination and communication patterns, can benefit from serverless computing.**

In this paper, we introduce serverless skeletons for parallel cloud programming to free developers from both parallelism and resource management issues. In particular, we investigate on the well-known and widely used farm skeleton, which supports the implementation of a wide range of applications. To evaluate our concepts, we present a prototypical development and runtime framework and implement two applications based on our framework: Numerical integration and hyperparameter optimization - a commonly applied technique in machine learning. We report on performance measurements for both applications and discuss the usefulness of our approach.

***Index Terms*—cloud computing, parallel computing, function as a service, parallel cloud programming, elasticity**

I. Introduction

Serverless computing is an emerging cloud computing paradigm that frees users from resource management issues. Therefore, serverless computing platforms enable the execution of user code in form of stateless *functions*. Compute resources are provisioned on-demand and scaled in an automated manner leading to two fundamental benefits: Elasticity by design and per-function resource accounting (and billing). Prominent serverless computing platforms include AWS Lambda[1] and Azure Functions[2] as well as open source solutions such as Apache OpenWhisk[3]. Whereas functions are stateless, serverless computing platforms also provide backend services to store data [1].

Exemplary applications of serverless computing include data filtering and transformation, log file analysis, or object recognition in images [1]. In all these cases, computations are triggered by an event or user request and can be executed independently of each other. This enables these applications to benefit from elastic auto-scaling in a straightforward manner.

More recently, serverless computing platforms have become of interest for parallel applications, which comprise most often complex coordination, communication, and synchronization patterns [2]–[4]. However, it is still an open research question to which extent parallel applications can benefit from serverless computing in form of on-demand and elastic compute resources as well as per-function resource accounting. Moreover, developing serverless parallel applications requires novel approaches to parallel programming.

In this paper, we introduce the concept of serverless skeletons to enable parallel cloud programming. Algorithmic skeletons [5], [6] have been introduced to structure parallel computations as a set of higher-level functions that abstract from complex coordination patterns inherent to parallel processing. We show how to use serverless skeletons to transparently ensure parallel coordination and communication based on serverless computing platforms while developers are able to implement functional code without considering parallelism and resource management issues. We specifically address the farm skeleton, which can be applied to implement a wide range of applications. We make the following contributions:

- We present a novel approach to parallel cloud programming based on serverless skeletons that enables developers to benefit from elastic compute resources without dealing with parallelism and resource management issues.
- We describe the design and implementation of a serverless farm skeleton and present a prototypical development and runtime framework based on Apache OpenWhisk.
- We evaluate the serverless farm skeleton with two example applications: numerical integration and hyperparameter optimization, which is a commonly applied technique in machine learning. We report on performance measurements for both applications.

This paper is structured as follows. In Section II, we discuss related work. In Section III, we present our approach to parallel cloud programming with serverless skeletons. We describe a development and runtime framework for the serverless farm skeleton as well as a corresponding prototypical implementation in Section IV. In Section V, we describe two example applications that we employ in our experimental evaluation in Section VI for performance measurements. Finally, in Section VII, we conclude our paper.

II. Fundamentals and Related Work

In this section, we discuss related work on serverless computing, parallel processing in the cloud, and skeletons.

A. Serverless Computing

Serverless computing can be seen as a natural evolution of the cloud computing paradigm and is heavily influenced by microservices, container virtualization, and event-driven

[1]https://aws.amazon.com/lambda.
[2]https://azure.microsoft.com/en-us/services/functions.
[3]https://openwhisk.apache.org.

978-1-7281-4117-6/19/$31.00 ©2019 IEEE
DOI 10.1109/DataCom.2019.00036

IEEE computer society

programming [7]. Whereas the microservices architectural style propagates the development and operation of fine-grained services, container virtualization helped to back this trend from a technological side. Following these developments, serverless computing enables function-level elasticity by decoupling compute from storage, which is a common approach to build cloud-native applications. The compute tier is represented by stateless FaaS functions[4] (Function as a Service) and the storage tier is given by backend services (Backend as a Service) such as databases, message queues, and caching systems [1]. Because FaaS functions themselves are not individually addressable (point-to-point communication is not supported), they can only communicate via shared backend services [8].

Typically, each FaaS function is executed in an event-driven manner, i.e., it is triggered when a specific event occurs or a user request is received. FaaS functions can also be invoked via platform-specific APIs. Technically, user code is executed in a sandboxed environment (such as a container) with a specific amount of compute resources (CPU, memory). State-of-the-art serverless computing offerings include AWS Lambda, Azure Functions[5], and Google Cloud Functions[6]. Moreover, open source platforms are available to be operated in a private cloud setting or on top of IaaS cloud offerings. Prominent examples are Apache OpenWhisk and Fission[7].

B. Parallel Processing in the Cloud

Pay-per-use and elasticity are fundamentally new concepts in the context of parallel applications [9], [10]. In traditional parallel execution environments such as clusters and grids, users had no visibility on the monetary costs of a computation and were not able to make use of elasticity by adapting the number of processing units at runtime. Existing research discusses how parallel applications can benefit from these cloud-specific properties [9]–[13]. More recently, serverless computing platforms have become of interest for parallel processing. The authors of [2] present a prototype called PyWren that enables developers to make use of AWS Lambda for parallel execution of locally developed code segments. Whereas they follow a code offloading approach, we deploy a topology of FaaS functions, which also allows parallel coordination tasks (captured by skeletons) to be executed on the serverless computing platform. The authors of [3] describe how to execute linear algebra algorithms on AWS Lambda. In [4], serverless computing platforms are evaluated for big data processing use cases based on a matrix multiplication application.

C. Algorithmic Skeletons

Algorithmic skeletons [5], [6] provide a method to structure parallel programs as a set of higher order functions that abstract over common patterns of parallel coordination. Because parallel coordination is captured by the skeleton, developers are able to implement functional code without considering parallelism issues. A major benefit of using skeletons is that coordination (such as orchestration and synchronization) is transparently handled, which substantially reduces runtime errors (e.g., due to deadlocks, starvation, and race conditions) when compared to low-level parallel programming models (such as MPI). Consequently, one can say that each skeleton comprises a built-in parallel behavior [14].

Algorithmic skeletons can be classified as either *task-parallel* with examples such as pipeline, farm, divide & conquer, and branch & bound or *data-parallel* such as map and fold [14]–[16]. Over the years, many frameworks and libraries have been developed for a variety of programming languages and parallel execution environments [17]–[20]. Whereas functional code is implemented by developers, provided compiling tools take care of automatically generating code for parallel execution to ease programming. Depending on the execution environment considered, parallel execution is based on POSIX threads, OpenMP, MPI, OpenCL, or CUDA. For instance, Muesli [15], [21] is a C++ template library that supports parallel execution on top of MPI, OpenMP, and CUDA. During the last years, skeleton-based programming models have been widely adopted. Probably the most prominent example is MapReduce [22], which has been designed to simplify data processing on large, heterogeneous clusters of commodity hardware and hides data distribution and parallel execution.

III. Parallel Cloud Programming with Serverless Skeletons

Skeletons capture common parallelism patterns and provide abstract implementations of these patterns for a parallel execution environment. In this work, we specifically address serverless computing platforms - a novel parallel execution environment with benefits such as built-in elasticity and per-function resource accounting. However, in contrast to other parallel execution environments, the specific characteristics of serverless computing platforms make the implementation of parallel coordination as well as the communication across parallel processing units challenging. In the following, we present several concepts and design principles to enable parallel cloud programming with serverless skeletons.

Skeleton-based Development with User- and Framework Functions: Algorithmic skeletons make use of the separation of concerns principle to free developers from parallelism concerns: Only the functional code is implemented by developers while code required for parallel coordination is provided by the skeleton itself. In the following, we refer to a code segment implemented by developers with the term *user function* and to a code segment provided by the skeleton with the term *framework function*. A user function essentially captures application-specific processing logic. Each skeleton declares the user functions, which have to be implemented by developers. A framework function implements a certain parallel coordination task such as task distribution or termination detection. By

[4]We refer to a function in the serverless computing context with the term FaaS function not to be confused with programming-level functions, which are used in the context of skeletons.

[5]https://azure.microsoft.com/en-us/services/functions.

[6]https://cloud.google.com/functions.

[7]https://fission.io.

following the serverless computing paradigm, parallel coordination has to be implemented based on backend services. This requires particular attention because the required consistency guarantees might not be provided by all backend services. We discuss several examples of user and framework functions in more detail for a serverless farm skeleton (cf. Section IV-A).

FaaS Function Topology Mapping: For parallel execution, each skeleton instance requires a specific number of FaaS functions that have to be deployed to the target serverless computing platform. To create these FaaS functions, one has to map user and framework functions to FaaS functions, i.e., functions as seen by the serverless computing platform. As shown in Fig. 1, we distinguish between a *skeleton function topology* and a *FaaS function topology*. Each skeleton has a specific skeleton function topology consisting of connected user and framework functions, which is mapped to a FaaS function topology for deployment. Mapping two user / framework functions to the same FaaS function means that they are executed sequentially by a single FaaS function. This separation enables flexibility with respect to the deployment of a skeleton instance: On the one hand, by mapping each user / framework function to an independent FaaS function, developers have the ability to fine-tune the resource requirements of each individual user and framework function, which can be scaled-out independently. On the other hand, by grouping user / framework functions and mapping them to the same FaaS function, developers can minimize various sources of overhead. Two major sources of overhead are (1) system overhead because, technically, a container is started for each FaaS function and (2) communication overhead because FaaS functions communicate via backend services as described in the following. We discuss several mappings to FaaS function topologies and their characteristics in the context of the serverless farm skeleton in Section IV-B.

Communication via Backend Services: Whereas user and framework functions that have been mapped to the same FaaS function can communicate via shared memory, communication across FaaS functions requires additional effort. Because point-to-point communication is not supported on serverless computing platforms, communication has to be implemented based on shared backend services (cf. Section II-A). To relieve developers of the burden of implementing and adapting code

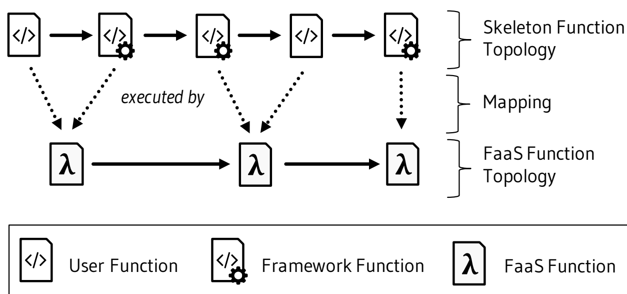

Fig. 1. For deployment, the skeleton function topology of a skeleton instance has to be mapped to a FaaS function topology.

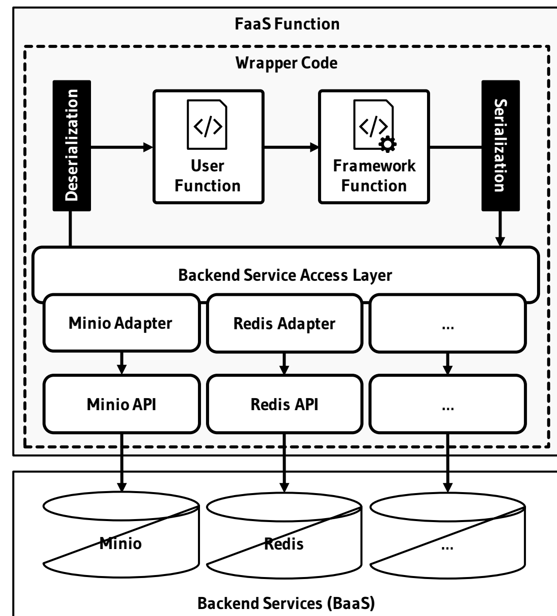

Fig. 2. FaaS functions can be automatically generated by combining user and framework functions according to the selected FaaS function topology. Generated wrapper code handles the communication via backend services as well as the serialization and deserialization of data.

for communication via backend services, the required wrapper code can be automatically generated per FaaS function. By following this approach, the interaction with backend services as well as the serialization and deserialization of data is transparent to developers and provided by the generated wrapper code. The internal structure of a generated FaaS function is depicted in Fig. 2. To support different backend services, we introduce a backend service access layer, which employs the adapter pattern. Backend services can thus be selected based on application-specific requirements and easily replaced. The selection of backend services largely depends on the type and size of data structures stored by a serverless skeleton instance as well as their access frequency. In general, frequently accessed, small data structures benefit from in-memory data stores with low access latency, whereas for huge communication volumes object storage services are a good choice.

Automated Delivery and Deployment: Delivery and deployment automation are integral concepts related to cloud programming and have been shown to effectively shorten software release cycles [23]. A system that automates the delivery process is called *continuous delivery pipeline* [24]. Fig. 3 summarizes the integration of the aforementioned concepts to create a continuous delivery pipeline for parallel cloud programming with serverless skeletons. Whereas developers have to implement the user functions required by a particular skeleton, all other steps shown in Fig. 3 can be automated including the compilation of a serverless skeleton instance (which includes the generation of wrapper code) and the deployment to a serverless computing platform by means of

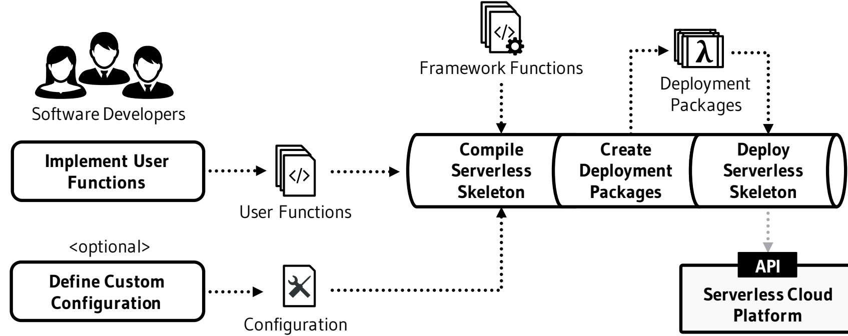

Fig. 3. A continuous delivery pipeline complements our approach for parallel cloud programming with serverless skeletons.

deployment packages. The specification of a skeleton-specific custom configuration is optional (zero configuration approach).

IV. Design and Implementation of a Serverless Farm Skeleton Framework

In this section, we discuss a serverless version of the well-known farm skeleton, which can be used in any pipeline to speed up the computation [25]. Many pleasingly parallel applications can be implemented based on this skeleton. Prominent examples include frame rendering in computer graphics, brute-force search in cryptography, and Monte Carlo simulation. To validate the concepts proposed in Section III, we present a Java-based development and runtime framework for the serverless farm skeleton. The remainder of this section is structured as follows. First, we describe the serverless computing platform addressed upon which we built our prototypical implementation. Subsequently, we discuss (1) the user and framework functions of our serverless farm skeleton as well as their implementations, (2) potential mappings of the skeleton function topology to FaaS function topologies, (3) the communication via shared backend services, as well as (4) debugging, delivery, and deployment aspects.

Serverless Computing Platform: The serverless computing platform addressed is Apache OpenWhisk - an open source serverless computing platform that executes FaaS functions based on events from external sources or API calls. Technically, functions are deployed as Docker containers. The functional logic implemented by developers is called Action in OpenWhisk jargon and can be written in one of the following programming languages: NodeJS, Swift, Java, Go, Scala, Python, PHP, Ruby, or Ballerina. In addition, we employ two backend services: MinIO and Redis. MinIO[8] is an open source object storage that provides an Amazon S3[9] compatible API for data access. Redis[10] is an in-memory data store that can be used as database, cache, or message broker.

A. User and Framework Functions

In this section, we describe the user and framework functions of a serverless farm skeleton (depicted in Fig. 4) and discuss related design considerations. Function naming is inspired by [25]. The signatures of user functions are declared by Java interface methods, which have to be implemented by developers. These Java interface methods are shown in Fig. 5. Note that framework functions are transparent to developers.

Predecessor (User) Function: The predecessor function receives a set of input key-value pairs and initiates the farm skeleton by creating a set of tasks. Each task is described by a set of key-value pairs with the key being a `String` and the value being an `Object`. Finally, the predecessor function returns the tasks that should be processed in parallel.

Dispatcher (Framework) Function: The dispatcher function is provided by the framework and enacts task distribution. Therefore, it invokes the implemented worker function once per task created by the predecessor.

Worker (User) Function: A worker function receives a task defined as a set of key-value pairs as input and computes a result value being an `Object`. Developers are free to implement any application-specific processing logic that maps the input to a result value.

Termination Detection (Framework) Function: To detect the termination [26] of all worker functions, the termination detection function is invoked by each worker function when its computation has been completed. Because point-to-point communication is not supported by serverless computing platforms and FaaS functions are stateless, termination detection has to be implemented based on a shared backend service. As termination is a persistent property of the global system state, which means that once detected it should never be changed again, the implementation of termination detection based on a backend service requires particular attention. False positive or false negative termination detection signals can compromise

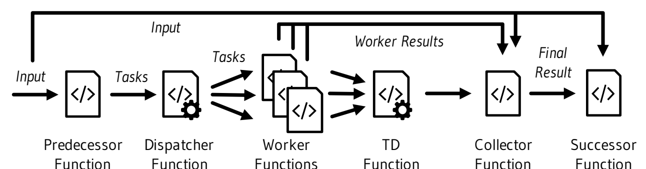

Fig. 4. User and framework functions of the serverless farm skeleton.

[8]https://min.io.
[9]https://aws.amazon.com/s3.
[10]https://redis.io.

Predecessor Function:	`Iterable<HashMap<String, Object>> predecessor(HashMap<String, Object> input);`
Worker Function:	`Object worker(HashMap<String, Object> task);`
Collector Function:	`Object collector(HashMap<String, Object> input, Iterator workerResults);`
Successor Function:	`HashMap<String, Object> successor(HashMap<String, Object> input, Object result);`

Fig. 5. Signatures of the serverless farm skeleton user functions declared by Java interface methods.

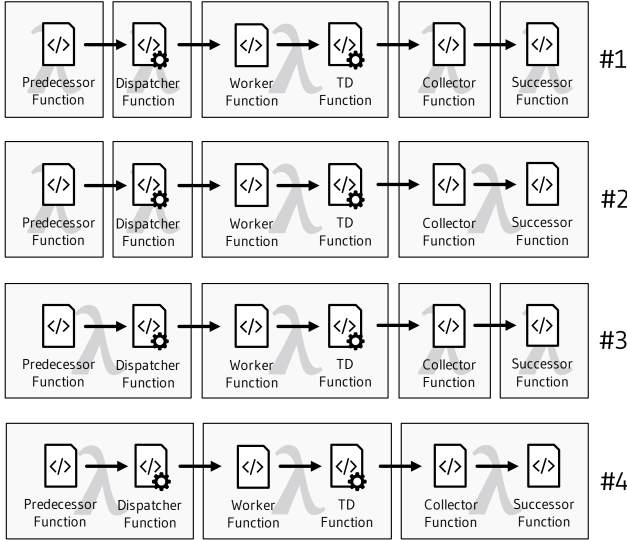

Fig. 6. Four alternative mappings of the serverless farm skeleton function topology to FaaS function topologies for deployment.

the execution by detecting termination more than once or never. Our implementation is based on Redis. We employ Redis' atomic increment operations to implement a counter, which is incremented atomically once per completed worker function. Termination is detected when the counter has reached the total number of worker functions. In this case, the collector function is invoked.

Collector (User) Function: The collector function receives a set of result values, where each result value has been computed by one worker function. Additionally, we pass the set of input key-value pairs, originally received by the predecessor function. This is required by applications for which the implementation of the collector function depends on the original input. Developers implement any application-specific aggregation logic that merges together these result values, e.g., summing up all values. The aggregated result value computed by the collector function is an arbitrary `Object`.

Successor (User) Function: The successor function receives the result value computed by the collector function. Developers are free to implement any application-specific result handling such as storing the result in a database or sending an email to inform a user about the completed computation. A successor function can also invoke other FaaS functions.

B. FaaS Function Topologies

Fig. 6 shows potential mappings of user and framework functions to FaaS functions. The first mapping offers the highest flexibility, whereas the last mapping is designed to minimize overhead (cf. Section III). Mapping #4 (cf. Fig. 6) has the least achievable overhead. Further reduction of FaaS functions is not possible because worker functions have to be scaled out to make use of parallel processing. Our framework provides a `YAML` configuration file that exposes different FaaS function topology mappings as configuration options. Mapping #4 is selected by default. Other mappings should only be selected if more flexibility is actually required.

C. Communication via Backend Services

Based on the communication concept described in Section III and depicted in Fig. 2, our framework transparently ensures the communication across FaaS functions. Our prototypical implementation of the backend service access layer supports two different backend services, namely MinIO and Redis. To implement the MinIO adapter, we rely on the MinIO client Java SDK[11] version 5.0.6. To implement the Redis adapter, we rely on the Redis Java client jedis[12] version 3.0.1. More adapters can be easily added.

Note that the framework transparently allocates and releases data stored in backend services and thus ensures that these services are only used when they are actually required. In contrast, programming serverless parallel applications in an ad hoc manner can lead to huge waste of costs: For instance, if developers forget to free allocated storage resources, which are payed per time unit.

D. Debugging, Delivery, and Deployment

For debugging purposes, we developed a testing tool, which can be used to test an implemented farm skeleton. The testing tool runs the farm skeleton on the developer's local machine and does not require a serverless computing platform to be installed. Therefore, we provide shared memory implementations of all framework functions and utilize multiple threads for parallel execution. Note that the physical parallelism and thus also the performance obtained heavily depends on the processors / cores available locally. However, we found the testing tool to be an adequate means to validate the implementation of user functions with small input data before deploying the skeleton instance to a serverless computing platform.

To deliver and deploy a serverless farm skeleton instance, we implemented a delivery pipeline as depicted in Fig. 3:

[11] https://github.com/minio/minio-java.
[12] https://github.com/xetorthio/jedis.

Compile Serverless Skeleton: User and framework functions are automatically grouped according to the FaaS function topology mapping selected. To enable communication across FaaS functions, we automatically generate the required wrapper code for each FaaS function. Note that communication via backend services requires the objects that should be stored to be serializable. Depending on the selected FaaS function topology, we automatically check if this is the case before deploying a skeleton instance to avoid runtime errors.

Create Deployment Packages: A deployment package in form of a JAR (Java Archive) file is created per FaaS function. Deployment packages contain all required dependencies of included user and framework functions as well as their third-party dependencies such as libraries used by the developer to implement user functions or libraries used by the provided framework functions. Moreover, the generated wrapper code is contained.

Deploy Serverless Skeleton: In the last step, deployment packages are used to automatically deploy the developed skeleton instance via the OpenWhisk API. To access OpenWhisk, we created a wrapper library that provides simple operations such as `createFaaS`, `deleteFaaS`, and `invokeFaaS` upon which our framework manages the lifecycle of a serverless skeleton instance or FaaS functions, respectively.

V. Case Studies

In this section, we present two prototypical applications that can be easily developed and deployed with our framework: Numerical integration and hyperparameter optimization for an artificial neural network. We describe the implementation of each application based on our framework in detail.

A. Numerical Integration

Our numerical integration application computes the numerical value of a definite integral of a user-defined function $f(x)$. We employ a commonly used technique for approximating the definite integral: The *trapezoidal rule* from the closed Newton-Cotes formulas [27]. Therefore, the region under the graph $f(x)$ is approximated as a trapezoid of which the area can be easily calculated:

$$\int_a^b f(x)dx \approx (b-a) \cdot \frac{f(a)+f(b)}{2} \quad (1)$$

A better approximation can be achieved by partitioning the integration interval $[a, b]$ and applying the trapezoidal rule to each subinterval. This procedure is also called the *composite trapezoidal rule*. Therefore, the closed interval $[a, b]$ is partitioned into N equally spaced subintervals, where each subinterval has a length of $\Delta x = \frac{b-a}{N}$. Increasing the number of subintervals makes the approximation more accurate. The numerical value of a definite integral can be calculated based on the composite trapezoidal rule as follows:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2} \cdot \left(f(x_0) + f(x_N) + \sum_{k=1}^{N-1} 2 \cdot f(x_k) \right), \quad (2)$$

where the values x_0 and x_N are equal to a and b, respectively.

Implementation: A developer has to implement the partitioning of the integration integral as part of the predecessor function, which is automatically dispatched by the dispatcher (framework) function. Each subinterval is calculated independently by a worker function. Termination is transparently detected by the termination detection (framework) function. Thereafter, the collector function calculates the final value of a definite integral based on Equation 2, which is relayed to the successor function accordingly.

B. Hyperparameter Optimization

Many machine learning techniques are configured by means of parameters that have to be manually selected. These parameters are called hyperparameters. A prime example are artificial neural networks, which can be configured by a multitude of hyperparameters that influence their network architecture (number of layers, layer size) or the learning process (learning rate). The optimal configuration has to be selected from a (most often) highly multi-dimensional hyperparameter space. Finding the optimal configuration is a non-trivial process referred to as hyperparameter optimization [28].

A commonly used approach for hyperparameter optimization is grid search, which we employ in our case study. However, note that also other approaches such as random search [29] can be easily implemented based on our framework because hyperparameter configurations can be evaluated independently of each other and can thus be farmed out for distributed computation.

Our hyperparameter optimization application considers a simple artificial neural network following the multilayer perceptron (MLP) architecture and is designed to optimize the layer size of a hidden layer. The goal is to find the layer size with the highest prediction accuracy (for the data set employed). The network architecture comprises three layers: An input layer, a hidden layer, and an output layer. To train the network, we use the well-known MNIST[13] data set, a large collection of handwritten digits that is commonly used to benchmark classification techniques. The input layer of the network has a fixed size of 784, which corresponds to the number of pixels of MNIST images ($28 \cdot 28 = 784$). The fully connected hidden layer uses Rectified Linear Units (ReLU) activation functions. The output layer has a fixed size of 10 (representing the 10 possible numbers of the MNIST data set), uses softmax activation functions, and a multi-class cross entropy loss function. The learning algorithm employed is stochastic gradient descent.

Implementation: A developer has to implement (1) the generation of hyperparameter configurations as part of the predecessor function, (2) the training and evaluation of an artificial neural network based on a hyperparameter configuration for the worker function, and (3) the aggregation of results for the collector function. In this case, the collector function

[13]http://yann.lecun.com/exdb/mnist.

selects the hyperparameter configuration that produced the best accuracy. The successor function writes the output to the console. Task distribution and termination detection are transparently handled by framework functions. Our implementation of hyperparameter optimization is based on Deeplearning4j[14] - a deep learning framework for the Java Virtual Machine (JVM). We employ the ND4J[15] scientific library for linear algebra operations.

VI. Experimental Evaluation

To evaluate our serverless farm skeleton framework, we measured the speedups that can be obtained by means of parallel execution for both applications described in Section V. Moreover, we compare the execution time with respect to the two different backend services supported by our prototypical implementation and the different FaaS function topologies depicted in Fig. 6. All our measurements were executed based on an Apache OpenWhisk installation hosted in our OpenStack-based private cloud environment. Our OpenWhisk cluster is operated on two Ubuntu 16.04 virtual machines (VM) with 14 vCPUs clocked at 2.6 GHz, 20 GB RAM, and 40 GB disk each. MinIO and Redis are operated on a single Ubuntu 16.04 VM with 2 vCPUs clocked at 2.6 GHz, 8 GB RAM, and 40 GB disk.

FaaS Function Topologies: First, we compare the different FaaS function topologies depicted in Fig. 6. Therefore, we measured the execution time of a numerical integration application instance with a sequential runtime T_s of 89.28 seconds. We measured an execution time of 99.21 seconds for topology #1, 96.41 seconds for topology #2, 96.35 seconds for topology #3, and 93.78 seconds for topology #4 with Redis as backend service and one worker FaaS function. As expected, topology #1 has the highest and topology #4 has the lowest overhead. This is related to the number of FaaS functions employed: Topology #1 employs 5 FaaS functions whereas topology #4 employs only 3 FaaS functions. Topology #2 and #3 both employ 4 FaaS functions thus leading to similar execution times. Technically, every FaaS function is executed in a Docker container that has to be started. However, note that topology #1 offers the highest flexibility (cf. Section III).

[14]https://github.com/deeplearning4j/deeplearning4j.
[15]https://github.com/deeplearning4j/nd4j.

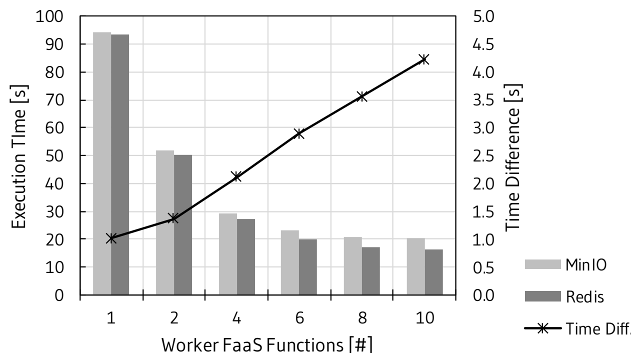

Fig. 7. Measured execution time of numerical integration application instance with FaaS function topology #4 based on MinIO / Redis backend service.

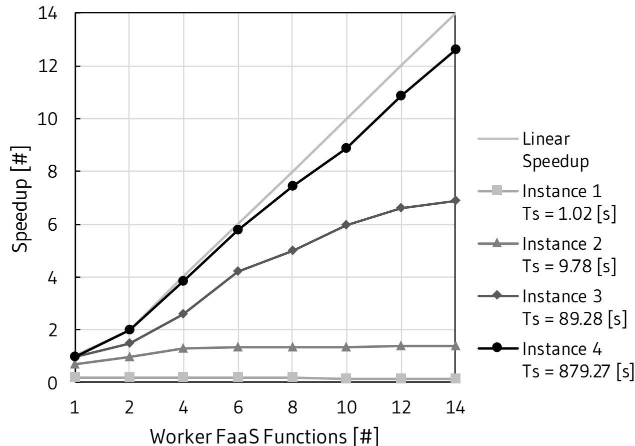

Fig. 8. Measured speedups of the numerical integration application with FaaS function topology #4 and Redis backend service.

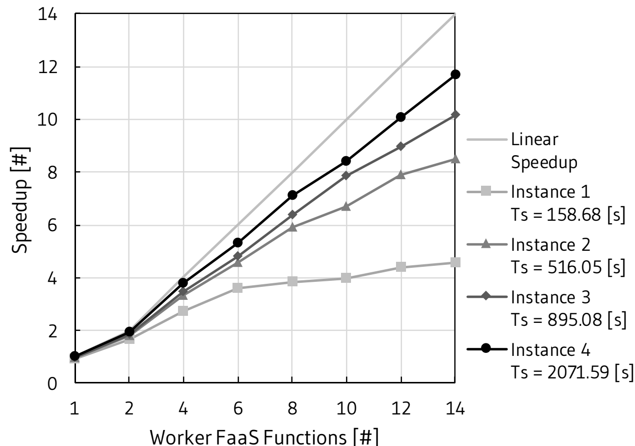

Fig. 9. Measured speedups of the hyperparameter optimization application with FaaS function topology #4 and Redis backend service.

Backend Services: To compare the performance of the two backend services supported by our prototypical implementation, namely MinIO and Redis, we executed the aforementioned instance of the numerical integration application ($T_s = 89.28$ seconds) with different degrees of parallelism. Fig. 7 compares the measured execution times based on MinIO and Redis and shows how the difference of both execution times evolves for an increasing degree of parallelism. The execution based on Redis is faster because it stores all data in memory.

Parallel Performance: We measured the parallel execution time for four instances of the numerical integration application (with different sequential runtimes) with respect to different degrees of parallelism. Fig. 8 shows the achieved speedups. For larger workloads, we achieved close to linear speedups. For small workloads, the overhead outweighs the utility of parallel execution. Speedups measured for the hyperparameter

optimization application are shown in Fig. 9. All parallel performance measurements were executed with FaaS function topology #4 and Redis backend service. To ensure reliable measurements, we executed less worker FaaS functions in parallel then vCPUs available. Moreover, we ensure that FaaS functions executed simultaneously are distributed across the OpenWhisk cluster by limiting the invoker user memory.

VII. Conclusion

In this work, we present a novel approach to parallel cloud programming that enables elastic parallel processing without considering parallelism or resource management issues. Based on the well-known concept of algorithmic skeletons, we are able to demonstrate that also parallel applications, which require coordination, communication, and synchronization, can benefit from serverless computing platforms. Serverless skeletons do not only ease parallel cloud programming, they can also save monetary costs by employing compute resources only when they are efficiently used. For instance, with respect to the farm skeleton, not all workers might complete their computation at the same time, which typically results from non-uniform task distribution or heterogeneous processing speeds of processing units (which is the common case in standard cloud environments). By executing workers as independent FaaS functions, we ensure that compute resources are not allocated after the computation has been completed.

Whereas our experimental evaluation shows very promising results for applications implemented based on the farm skeleton, also note that many other parallel execution models (and corresponding skeletons) heavily rely on the consideration of data locality to efficiently exploit compute resources, which is not supported by current serverless computing platforms. This issue should be further investigated in future work. For instance, to retain the strict separation of stateless FaaS functions and backend services, locality-aware backend services can be offered by cloud providers, which store data in close physical proximity to FaaS functions (e.g., on the same rack).

Acknowledgment

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program *Services Computing*.

References

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," 2019.

[2] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*. New York, NY, USA: ACM, 2017, pp. 445–451.

[3] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "numpywren: serverless linear algebra," *CoRR*, vol. abs/1810.09679, 2018.

[4] S. Werner, J. Kuhlenkamp, M. Klems, J. Müller, and S. Tai, "Serverless big data processing using matrix multiplication as example," in *2018 IEEE International Conference on Big Data*, 2018, pp. 358–365.

[5] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

[6] S. Gorlatch and M. Cole, *Parallel Skeletons*. Boston, MA: Springer US, 2011, pp. 1417–1422.

[7] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. U, and A. Iosup, "Serverless is more: From paas to present cloud computing," *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, Sep. 2018.

[8] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[9] S. Kehrer and W. Blochinger, "Migrating parallel applications to the cloud: assessing cloud readiness based on parallel design decisions," *SICS Software-Intensive Cyber-Physical Systems*, vol. 34, no. 2, pp. 73–84, Jun 2019.

[10] ——, "A survey on cloud migration strategies for high performance computing," in *Proceedings of the 13th Advanced Summer School on Service-Oriented Computing*. IBM Research Division, 2019.

[11] J. Haussmann, W. Blochinger, and W. Kuechlin, "Cost-efficient parallel processing of irregularly structured problems in cloud computing environments," *Cluster Computing*, vol. 22, no. 3, pp. 887–909, Sep 2019.

[12] S. Kehrer and W. Blochinger, "Taskwork: A cloud-aware runtime system for elastic task-parallel hpc applications," in *Proceedings of the 9th International Conference on Cloud Computing and Services Science*. SciTePress, 2019, pp. 198–209.

[13] ——, "Elastic parallel systems for high performance cloud computing: State-of-the-art and future directions," *Parallel Processing Letters*, vol. 29, no. 02, p. 1950006, 2019.

[14] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While, "Parallel programming using skeleton functions," in *PARLE '93 Parallel Architectures and Languages Europe*, A. Bode, M. Reeve, and G. Wolf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 146–160.

[15] H. Kuchen, *Parallel Programming with Algorithmic Skeletons*. Cham: Springer International Publishing, 2019, pp. 527–536.

[16] M. Danelutto, F. Pasqualetti, and S. Pelagatti, "Skeletons for data parallelism in p3l," in *Euro-Par'97 Parallel Processing*, C. Lengauer, M. Griebl, and S. Gorlatch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 619–628.

[17] F. Alexandre, R. Marques, and H. Paulino, "On the support of task-parallel algorithmic skeletons for multi-gpu computing," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: ACM, 2014, pp. 880–885.

[18] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, "P3 l: A structured high-level parallel language, and its structured support," *Concurrency: Practice and Experience*, vol. 7, no. 3, pp. 225–255, 1995.

[19] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Software: Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.

[20] M. Aldinucci, M. Danelutto, and P. Teti, "An advanced environment supporting structured parallel programming in java," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 611 – 626, 2003.

[21] H. Kuchen and J. Striegnitz, "Features from functional programming for a c++ skeleton library," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 78, pp. 739–756, 2005.

[22] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[23] S. Kehrer, F. Riebandt, and W. Blochinger, "Container-based module isolation for cloud services," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 177–186.

[24] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley, 2010.

[25] M. Poldner and H. Kuchen, "On implementing the farm skeleton," *Parallel Processing Letters*, vol. 18, no. 01, pp. 117–131, 2008.

[26] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Pearson Education, 2003.

[27] K. E. Atkinson, *An Introduction to Numerical Analysis*, 2nd ed. John Wiley & Sons, Inc., 1989.

[28] J. Bergstra, D. Yamins, and D. D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13, 2013, pp. I–115–I–123.

[29] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012.