Cluster Computing (2021) 24:799–817
https://doi.org/10.1007/s10586-020-03158-3

Self-tuning serverless task farming using proactive elasticity control

Stefan Kehrer[1] · Dominik Zietlow[2] · Jochen Scheffold[1] · Wolfgang Blochinger[1]

Received: 20 January 2020 / Revised: 13 July 2020 / Accepted: 15 July 2020 / Published online: 23 July 2020

Abstract
The cloud evolved into an attractive execution environment for parallel applications, which make use of compute resources to speed up the computation of large problems in science and industry. Whereas Infrastructure as a Service (IaaS) offerings have been commonly employed, more recently, serverless computing emerged as a novel cloud computing paradigm with the goal of freeing developers from resource management issues. However, 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 and benefit from on-demand and elastic compute resources as well as per-function billing. In this work, we discuss how to employ serverless computing platforms to operate parallel applications. We specifically focus on the class of parallel task farming applications and introduce a novel approach to free developers from both parallelism and resource management issues. Our approach includes a proactive elasticity controller that adapts the physical parallelism per application run according to user-defined goals. Specifically, we show how to consider a user-defined execution time limit after which the result of the computation needs to be present while minimizing the associated monetary costs. To evaluate our concepts, we present a prototypical elastic parallel system architecture for self-tuning serverless task farming and implement two applications based on our framework. Moreover, we report on performance measurements for both applications as well as the prediction accuracy of the proposed proactive elasticity control mechanism and discuss our key findings.

Keywords Cloud computing · Parallel computing · Function-as-a-service · Parallel cloud programming · Elasticity · Programming model

1 Introduction

Serverless computing can be seen as a natural evolution of former cloud service models and is heavily influenced by microservices, container virtualization, and event-driven programming [53]. 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 Function as a Service (FaaS) functions[1] and the storage tier is given by backend services (Backend as a Service) such as databases, message queues, and caching systems [29]. Because FaaS functions themselves are not individually addressable (point-to-point communication is not supported), they can only communicate via shared backend services [24].

✉ Stefan Kehrer
stefan.kehrer@reutlingen-university.de

Dominik Zietlow
dominik.zietlow@tuebingen.mpg.de

Jochen Scheffold
jochen.scheffold@student.reutlingen-university.de

Wolfgang Blochinger
wolfgang.blochinger@reutlingen-university.de

[1] Parallel and Distributed Computing Group, Reutlingen University, Alteburgstr. 150, 72762 Reutlingen, Germany

[2] Autonomous Learning Group, Max-Planck-Institute for Intelligent Systems, Max-Planck-Ring 4, 72076 Tübingen, Germany

[1] We refer to a function in the serverless computing context with the term FaaS function not to be confused with programming-level functions.

Prominent serverless computing platforms include AWS Lambda[2] and Azure Functions[3] as well as open source solutions such as Apache OpenWhisk[4]. Exemplary applications of serverless computing include data filtering and transformation, log file analysis, or object recognition in images [29]. 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 [9, 28, 51, 54]. Several challenges related to the development and operation of parallel applications arise from the specific characteristics of serverless computing platforms (e.g., how to implement communication based on shared backend services). Additionally, pay-per-use and elasticity are fundamentally new concepts that have to be considered [23, 32–35]: In traditional parallel execution environments such as clusters and grids, users had no visibility of the monetary costs of a computation and were not able to make use of elasticity by adapting the number of processing units at runtime.

In this work, we enable parallel cloud programming by introducing *self-tuning serverless skeletons*, which separate functional application development from non-functional aspects of the execution. Self-tuning serverless skeletons are based on the concept of algorithmic skeletons [14, 19], which has been introduced to structure parallel computations as a set of higher-level functions that abstract from complex coordination patterns inherent to parallel processing. We argue that by following a skeleton-based approach, developers are relieved of parallelism and resource management issues while an elasticity controller is able to make use of elastic compute resources to automatically handle non-functional requirements related to parallel processing. For instance, more resources can be employed to speed up the computation when the results are required immediately. At the same time, because using more processing units leads to higher monetary costs, a moderate number of processing units is automatically provisioned when the deadline is further away, thus saving money otherwise unnecessarily spent for additional compute resources.

We specifically focus on the class of parallel task farming applications because it comprises many applications of practical relevance and is well-suited for serverless computing [38]. Parallel task farming applications split the total workload and distribute multiple tasks across a set of processing units to speed up the computation [47]. Our main contributions are a proactive elasticity controller that employs non-linear regression techniques to control the number of FaaS functions per application run according to a user-defined execution time limit as well as a corresponding elastic parallel system architecture for self-tuning serverless task farming based on a serverless computing platform. This work is based on previous research contributions presented in [38], where we describe a novel approach to parallel cloud programming called serverless skeletons that separates functional application development from non-functional aspects of parallel execution. We extend our former work by discussing a proactive elasticity controller to automatically handle non-functional requirements, an elastic parallel system architecture, as well as a corresponding prototype and new experimental results. Additionally, we discuss the applicability and requirements of proactive elasticity control mechanisms and describe our key findings.

The remainder of this work is structured as follows. In Sect. 2, we discuss the cost/efficiency-time trade-off inherent to parallel processing in the cloud. In Sect. 3, we present our approach to parallel cloud programming with serverless skeletons. We describe a serverless task farming framework as well as a corresponding prototypical implementation in Sect. 4. In Sect. 5, we describe two implemented example applications for serverless task farming. The proactive elasticity controller and a corresponding elastic parallel system architecture are described in Sect. 6. The results of an extensive experimental evaluation are provided in Sect. 7. In Sect. 8, we discuss the presented approach, its underlying assumptions, as well as our key findings. In Sect. 9, we describe related work. Finally, in Sect. 10, we conclude our work.

2 Parallel processing in the cloud

The consumption of compute resources changed drastically with the emerging cloud computing paradigm: The cloud provides metered resources on-demand, which have to be payed on a per-use basis. Consequently, the monetary costs of parallel computations have to be explicitly considered per application run [21, 31, 48]. In this regard, Fig. 1 compares three different application runs with different numbers of processing units for an (ideal) perfectly scalable and two (realistic) non-perfectly scalable parallel systems. The areas shown for each application run

[2] https://aws.amazon.com/lambda.

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

[4] https://openwhisk.apache.org.

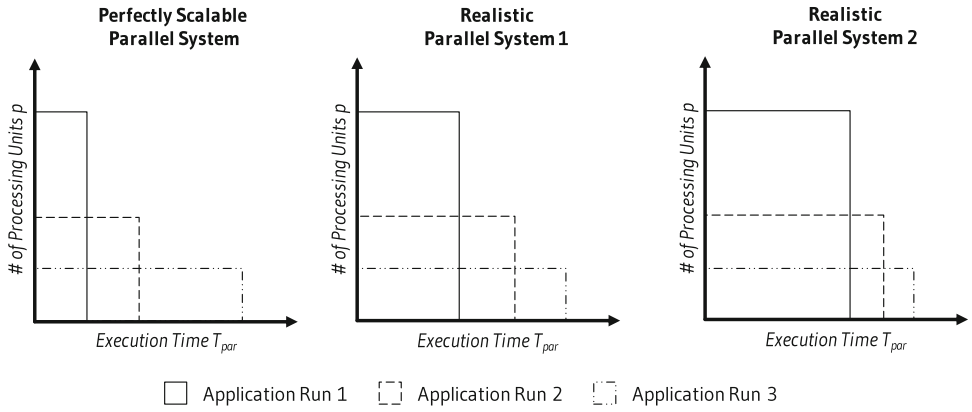

Fig. 1 Whereas the monetary costs of a perfectly scalable parallel system are independent of the number of processing units employed, for parallel systems that are not perfectly scalable, the monetary costs increase with the number of processing units

visualize the numbers of processing units as well as how long they are employed for the computation. As we can easily see, the areas shown for the perfectly scalable parallel system all have the same size, whereas the sizes of the areas shown for the non-perfectly scalable parallel systems increase with an increasing number of processing units. This can be explained by the parallel overhead that increases with the number of processing units [31]. The parallel overhead O_{par} is defined as the total time collectively spent by all processing units over and above the sequential execution time T_{seq} and modeled as

$$O_{par} = p \cdot T_{par} - T_{seq}, \tag{1}$$

where p is the number of processing units employed and T_{par} is the parallel execution time [20].

How much overhead occurs for which number of processing units depends on the specific scaling behavior of the parallel system considered. In cloud environments, because one pays processing units per time unit, both using more processing units as well as using processing units for a longer period of time increases the monetary costs of a parallel computation C_{par}, which can be defined as

$$C_{par} = T_{par} \cdot p \cdot c_{\pi}, \tag{2}$$

where c_{π} the price of one processing unit per time unit.

As a result, whereas elasticity enables users to control the number of processing units by means of an elasticity controller, a *cost/efficiency-time trade-off* has to be considered for all but the ideal (perfectly scalable) case [31]: Whereas adding more processing units effectively reduces the execution time, a higher number of processing units also leads to higher monetary costs due to a lower efficiency. This leads to two conflicting optimization goals: Reduce monetary costs & maximize efficiency vs. shorten the execution time of an application run.

In this work, we propose an approach to handle the cost/efficiency-time trade-off in an automated manner. Therefore, we first introduce the concept of serverless skeletons to separate functional application development aspects from non-functional aspects of parallel execution. Then, we show how to construct and integrate a proactive elasticity controller that automatically adapts the number of processing units (in form of FaaS functions) per application run according to a user-defined execution time limit while minimizing the associated monetary costs of the computation. Parallel applications based on serverless skeletons provide two essential benefits: Simplified application development without considering resource management issues and specific insights into the structure of an application that can be exploited by an automated elasticity control mechanism. This is also the key difference of our approach and existing work, which is discussed in Sect. 9 in more detail.

3 Serverless skeletons for parallel cloud programming

A major benefit of using skeletons is that coordination, communication, 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 [17]. In this work, we specifically address serverless computing platforms - a novel parallel execution environment with benefits such as 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 and 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 aspects: 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 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 serverless task farming (cf. Sect. 4.1).

Communication via backend services Whereas user and framework functions that are executed by 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. To relieve developers of the burden of implementing and adapting code 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. Two exemplary backend services supported by our prototypical implementation (cf. Sect. 4) are MinIO and Redis.

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 [37]. A system that automates the delivery process is called *continuous delivery pipeline* [26]. Figure 3 summarizes the integration of the aforementioned concepts to create a continuous delivery

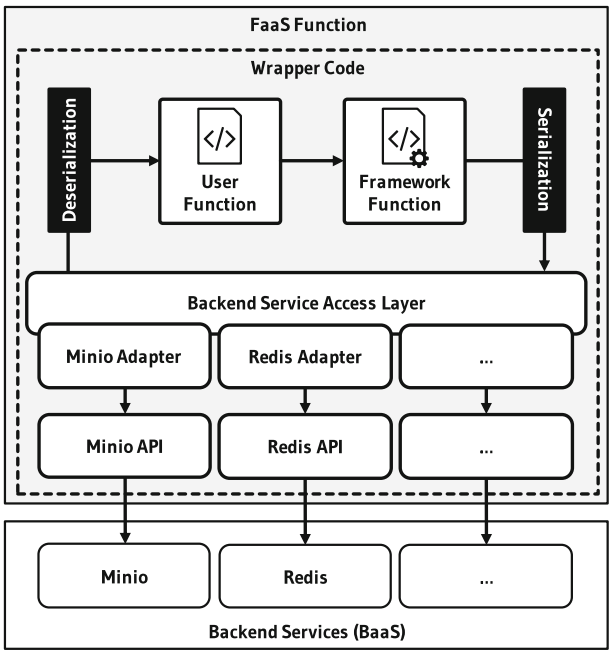

Fig. 2 FaaS functions can be automatically generated by combining user and framework functions. Generated wrapper code handles the communication via backend services as well as the serialization and deserialization of data

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

4 Serverless task farming framework

In this section, we discuss a serverless version of the well-known farm skeleton [47]. Many parallel applications can be implemented based on this skeleton. Prominent examples include brute-force search in cryptography, frame rendering in computer graphics, Monte Carlo simulation, and many machine learning tasks. To validate the concepts proposed in Sect. 3, we present a Java-based development and runtime framework for serverless task farming. 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 (1) discuss the user and framework functions of our serverless farm skeleton as well as their implementations, (2) the communication via shared backend services, as well as (3) delivery and deployment aspects.

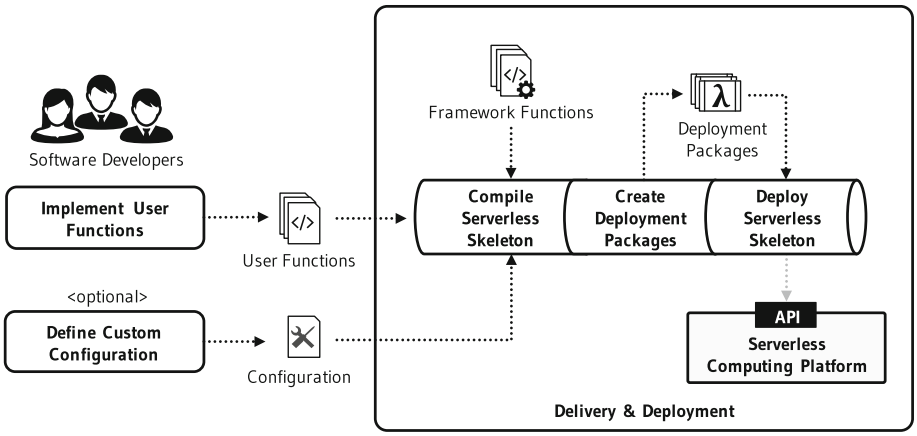

Fig. 3 A continuous delivery pipeline complements our approach for parallel cloud programming with serverless skeletons. User and framework functions are automatically compiled into a serverless skeleton instance, which is then described in form of deployment packages and deployed to a serverless computing platform

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[5] is an open source object storage that provides an Amazon S3[6] compatible API for data access. Redis[7] is an in-memory data store that can be used as database, cache, or message broker.

4.1 User and framework functions

In this section, we describe the user and framework functions of the serverless farm skeleton (depicted in Fig. 4) and discuss related design considerations. Function naming is inspired by [47]. The signatures of user functions are declared by Java interfaces, which have to be implemented by developers. Relevant Java 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. Also note that serializable objects are required to enable communication based on shared backend services. 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 description, which is 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 [20] 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 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 (for an example cf. Sect. 5.1). Developers implement any application-specific aggregation

[5] https://min.io.

[6] https://aws.amazon.com/s3.

[7] https://redis.io.

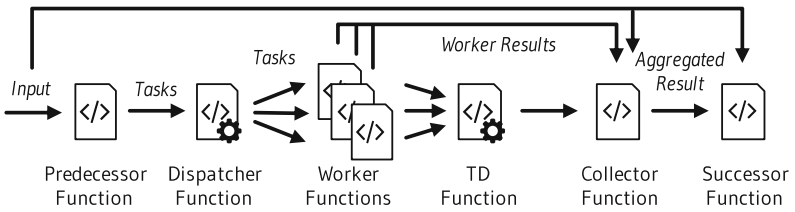

Fig. 4 User and framework functions of the serverless farm skeleton

```
Iterable<HashMap<String, Object>> predecessor(HashMap<String, Object> input);
Object worker(HashMap<String, Object> task);
Object collector(HashMap<String, Object> input, Iterable<Object> workerResults);
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

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.

4.2 Communication via backend services

Based on the communication concept described in Sect. 3 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. 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 money: For instance, if developers forget to free allocated storage resources, which are billed and payed per time unit. However, note that there are cases in which the framework cannot transparently release data stored in backend services, e.g., when a user-provided program throws an exception that forces the whole framework to terminate.

4.3 Delivery and deployment

To deliver and deploy a serverless farm skeleton instance, we implemented a delivery pipeline according to Fig. 3, which (1) groups user and framework functions to compile a serverless skeleton, (2) creates the required deployment packages, and (3) deploys the developed skeleton instance via the OpenWhisk API. The delivery pipeline is described in [38] in more detail. However, note that the user and framework functions depicted in Fig. 4 can be mapped to FaaS functions in many different ways. For instance, to execute the application sequentially, all user and framework functions can be mapped to a single FaaS function thus that no network communication or backend service access is required (cf. Fig. 2). In this case, all framework functions are implemented as regular method calls and executed by a single FaaS function. On the other hand, to parallelize the application, user and framework functions can be mapped to more FaaS functions, thus that multiple worker FaaS functions can be run in parallel to speed up the computation. Both mappings are also depicted in Fig. 6. Note that, technically, only a single container is deployed for mapping (1), whereas several containers have to be executed by the serverless computing platform to run a skeleton instance with mapping (2).

5 Implementing serverless parallel applications

In this section, we present two prototypical applications that can be easily developed and deployed with our framework: Numerical integration and hyperparameter optimization of an artificial neural network. We describe

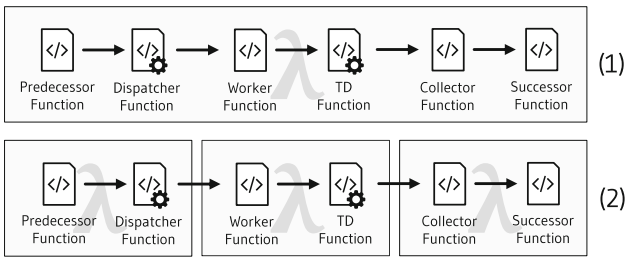

Fig. 6 Alternative mappings of the serverless farm skeleton function topology to FaaS functions for deployment. Mapping (1) represents a mapping for sequential execution of the skeleton instance and (2) shows a mapping for a parallel version of the skeleton instance

the implementation of each application based on our framework in detail. Both applications are implemented in Java.

5.1 Numerical integration

Our numerical integration application computes the numerical value of a definite integral of a user-defined real-valued function $f(x)$. We employ a commonly used technique for approximating the definite integral: The *trapezoidal rule* from the closed Newton–Cotes formulas [7]. Therefore, the region under the graph $f(x)$ is approximated as a trapezoid of which the area can be easily calculated:

$$\int_a^b f(x)dx \approx (b-a) \cdot \frac{f(a)+f(b)}{2} \tag{3}$$

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:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2} \cdot \left(f(x_0) + f(x_N) + 2 \cdot \sum_{k=1}^{N-1} f(x_k) \right), \tag{4}$$

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 Eq. (4), which is relayed to the successor function accordingly.

5.2 Hyperparameter optimization

Many machine learning techniques are configured by means of parameters that have to be determined. 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 [12].

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 [11] can be easily implemented based on our serverless task farming framework because hyperparameter configurations can be evaluated independently of each other and can thus be farmed out for distributed computation. Random and grid search are discussed more thoroughly in [11].

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[8] 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 output layer has a fixed size of 10 (representing the 10 possible numbers of the MNIST data set). 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 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[9]—a deep learning framework for the Java Virtual Machine (JVM). We employ the ND4J[10] scientific library for linear algebra operations. Whereas we do not use specific hardware accelerators in this work, ND4J also supports graphics processing units (GPU).

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

[9] https://github.com/deeplearning4j/deeplearning4j.

[10] https://github.com/deeplearning4j/nd4j.

6 Proactive elasticity control

Up to this point, serverless skeletons have been introduced as a novel approach to parallel cloud programming that separates functional and non-functional aspects. On this basis, this section shows how to make them self-tuning, i.e., how to integrate a proactive elasticity controller, which handles the cost/efficiency-time trade-off (cf. Sect. 2) based on user-defined settings in an automated manner.

6.1 Automating the cost/efficiency-time trade-off

A user that employs the serverless task farming framework to implement and execute an application has to select the number of processing units (i.e., the number of worker FaaS functions) per application run. This is implicitly accomplished by the predecessor function that generates a specific number of tasks, each of which is processed by an independent worker FaaS function (cf. Sect. 4.1). As an additional feature, we present a proactive elasticity control mechanism that is able to predict the number of processing units required to meet a user-defined execution time limit after which the result of the computation needs to be present while minimizing the associated monetary costs. This is an important scenario, e.g., when parallel processing is embedded in existing workflows. To minimize the monetary costs of the computation, one has to select the minimum number of processing units with which the computation still finishes within the user-defined execution time limit. With this approach, a user does not have to understand the scaling behavior of the application. Rather, the number of processing units is selected by the elasticity controller according to the user-defined execution time limit and thus the cost/efficiency-time trade-off is handled in an automated manner. To this end, the elasticity controller requires a prediction model that captures the application-specific scaling behavior and is able to estimate the required number of processing units upfront. We discuss how to generate such a model by following a supervised learning approach that considers monitoring data obtained from previous runs of the application.

Finding the required number of processing units can be considered a regression problem. As opposed to classification problems, which require predicting a discrete class label output, regression problems require predicting a continuous quantity output. The output in this case is the number of processing units, which should be provisioned for the computation. Note that, to employ regression techniques, the number of processing units is modeled as continuous quantity, but it can easily be rounded to a (non-negative) integer value for provisioning. Regression techniques can be used to fit a model that explains a response variable based on one (or more) explanatory variables by estimating the model parameters from data. Therefore, often the method of least squares is used to minimize the sum of the squares of the differences between the observed response variable in a given data set and the predicted response variable. The resulting model can be used to make a prediction of the response variable if values of the explanatory variable(s) are known.

In the following, we discuss how to build a model that is able to capture the non-linear scaling behavior related to parallel applications based on regression techniques and can be employed to predict the required number of processing units to meet a user-defined execution time limit.

6.2 Constructing a prediction model

To meet a user-defined execution time limit T_{limit}, the elasticity controller has to select the required number of processing units p that speeds up the processing such that the parallel execution time $T_{par} \leq T_{limit}$. The speedup S is defined as $S = \frac{T_{seq}}{T_{par}}$ [20]. However, due to the non-linear scaling behavior, the speedup does not increase linearly with respect to the number of processing units p. This behavior can be explained based on Amdahl's law [6] which says

$$T_{seq} = t_s + t_p \tag{5}$$

and

$$T_{par} = t_s + \frac{t_p}{p}, \tag{6}$$

where t_s is the execution time of the inherent sequential program part and t_p is the execution time of the parallelizable program part.

According to Amdahl's law, the increase in speedup with each additionally added processing unit decreases for an increasing total number of processing units p thus leading to a non-linear scaling behavior.

In the context of serverless task farming, the parallelizable program part t_p can be considered as the execution time of the worker FaaS functions. The execution time of the sequential program part t_s can be considered as the execution time of all other functions, which are inherently sequential (cf. Fig. 4).

Based on Eq. (6), the number of processing units can be expressed as

$$p = \frac{t_p}{T_{par} - t_s}. \tag{7}$$

According to Eq. (5), the execution time of the parallelizable program part can be expressed as $t_p = T_{seq} - t_s$.

Consequently, the number of processing units p can also be described as

$$p = \frac{T_{seq} - t_s}{T_{par} - t_s}. \tag{8}$$

As a result, the number of processing units can be explained by means of the sequential execution time T_{seq}, the execution time of the sequential program part t_s, and the parallel execution time T_{par}.

As the goal is to find the required number of processing units to meet a user-defined execution time limit T_{limit}, one can set $T_{par} = T_{limit}$. However, to determine the sequential execution time T_{seq} and the execution time of the sequential program part t_s additional measurements and program analyses are required, which are not profitable in a practical scenario. To deal with this issue, we propose the use of regression techniques to estimate T_{seq} and t_s from labeled performance measurement data, i.e., monitoring data obtained from previous application runs. In the following, we explain how to construct a corresponding regression model that allows the prediction of the required number of processing units. As input for the model, i.e., as explanatory variables, we consider the input size I_{size}, which simply describes the size of a given input in form of a numeric value, and the user-defined execution time limit T_{limit}, which are both known initially.

To enable predictions of the number of processing units, it is required that the sequential execution time T_{seq} and the execution time of the sequential program part t_s can be described as a function of the input size I_{size}, which can be fitted based on labeled performance measurement data. Note that, according to Amdahl's law, both the sequential execution time T_{seq} and the execution time of the sequential program part t_s are independent of the number of processing units. Here, we model the sequential execution time T_{seq} as nth degree polynomial of I_{size}, i.e.,

$$T_{seq} = \sum_{i=0}^{n} \alpha_i \cdot (I_{size})^i, \tag{9}$$

where $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_n)$ is a parameter vector and n controls the size of this vector.

Similarly, we model the execution time of the sequential program part t_s as mth degree polynomial of I_{size}, i.e.,

$$t_s = \sum_{j=0}^{m} \beta_j \cdot (I_{size})^j, \tag{10}$$

where $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_m)$ is a parameter vector and m controls the size of this vector.

Polynomials are commonly used in curve fitting due to their flexibility of shapes [1, 13]. This modeling also covers linear and quadratic relations depending on how n and m are selected. In Sect. 7, we show that this modeling is sufficient to enable accurate predictions for task farming applications. However, in the context of other application classes, other models might be used for T_{seq} and t_s (e.g., logarithmic or exponential models).

Based on Eqs. (8), (9), and (10), a non-linear regression model that enables the prediction of the required number of processing units can be constructed as

$$\hat{p} = \frac{\sum_{i=0}^{n} \alpha_i \cdot (I_{size})^i - \sum_{j=0}^{m} \beta_j \cdot (I_{size})^j}{T_{limit} - \sum_{j=0}^{m} \beta_j \cdot (I_{size})^j}, \tag{11}$$

where $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_n)$ and $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_m)$ are the parameter vectors of the model and n and m control the sizes of these vectors.

To instantiate a concrete prediction model by means of supervised learning, n and m have to be defined and the parameter vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ have to be estimated from data. Note that increasing n and m increases the number of parameters of the model. However, a general design goal of regression models is to keep the number of model parameters small. This has several reasons: simple models are easier to understand, avoid the curse of dimensionality, and reduce the risk of overfitting [1, 13]. Therefore, the prediction model is generated for different, increasing values of n and m (cf. Eq. 11) until the accuracy of the resulting model in terms of the root-mean-squared-error (RMSE) cannot be significantly increased anymore. Technically, this can be evaluated by comparing the increase in accuracy to a defined threshold. A prototypical implementation is discussed in more detail in Sect. 6.3.

The resulting model provides the required number of processing units $\hat{p}$ based on a given input size and a user-defined execution time limit. After estimating the model parameters from measurement data, an elasticity controller is able to employ such a model for predictions. An elastic parallel system architecture based on the concept of serverless skeletons as well as its implementation, which also integrates such an elasticity controller, is described in the following.

6.3 Serverless elastic parallel system architecture

To enable proactive elasticity control for serverless skeletons, several additional functions have to be introduced, which are described in the following. Figure 7 shows the resulting serverless elastic parallel system architecture in the context of serverless task farming. Whereas this architecture is independent of the technologies and programming languages used, a Java-based prototypical implementation that employs Redis as monitoring and model backend service is described accordingly.

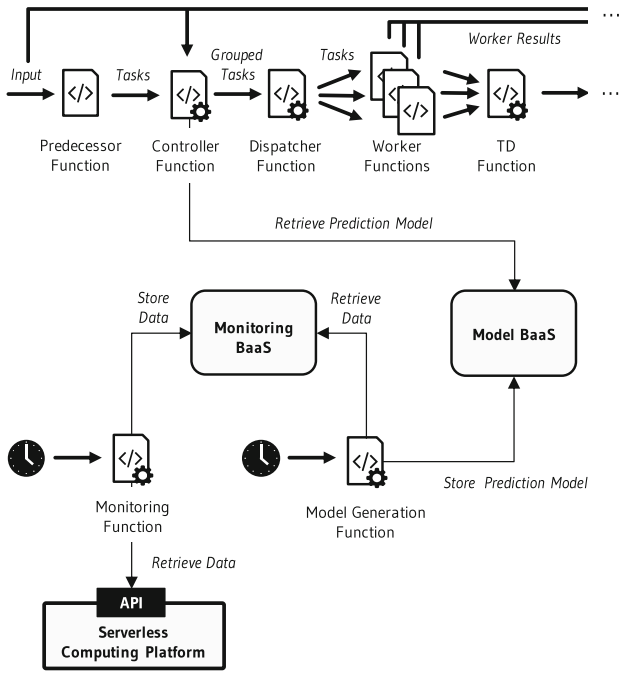

Fig. 7 Serverless elastic parallel system architecture

Monitoring (framework) function The monitoring function extracts relevant monitoring data of previous application runs from the serverless computing platform and stores them in the monitoring backend service. Alternatively, monitoring data can also be generated by the skeleton framework, e.g., by storing custom time stamps in the monitoring backend service. This can be easily accomplished by adding instrumentation code to the wrapper code of a skeleton instance's FaaS functions (cf. Sect. 3) and enables the consideration of custom metrics. The monitoring (framework) function is triggered periodically.

Model generation (framework) function The model generation function generates a prediction model by using the monitoring data stored in the monitoring backend service. Also preprocessing steps can be integrated, e.g., to only consider recent data. The model generation process itself is performed according to the concepts discussed in Sect. 6.2. Therefore, the model is fitted to the data records by adjusting the model parameters. The underlying non-linear least squares problem is solved by employing the Levenberg-Marquardt algorithm [40, 43], or more specifically a Java-based implementation[11], which uses JAMA[12] version 1.0.3 for basic linear algebra operations. As discussed in Sect. 6.2, the number of parameters is increased until the accuracy of the resulting model in terms of the RMSE cannot be significantly increased anymore. This is accomplished by comparing the increase in accuracy to a defined threshold. In this regard, our prototypical implementation employs a default threshold of 0.01. The produced prediction model is stored in the model backend service. Note that the model backend service and monitoring backend service can also be the same entity. The model generation (framework) function is triggered periodically.

Controller (framework) function The controller function uses information on the application's input as well as the user-defined execution time limit as input variables for the prediction model, which it loads from the model backend service. The outcome of the model is a predicted number of processing units, which is employed to group user-defined tasks (i.e., tasks generated by the predecessor function). These task groups are finally passed to the dispatcher function. The dispatcher function invokes the worker function once per task group. Each worker function executes the received group of tasks sequentially. To enable flexibility with respect to task grouping, developers should provide fine-grained tasks by making use of overdecomposition. Note that the maximum number of atomic tasks limits the maximum number of processing units that can be efficiently employed for the computation.

All these functions are managed by the serverless skeleton framework to relieve the user. The controller function is mapped to the FaaS function that hosts the dispatcher function. The monitoring function and the model generation function can be mapped to the same FaaS function for deployment. In this case, the corresponding FaaS function is triggered periodically to obtain new monitoring data and to generate new models. If a single FaaS function is employed, monitoring data has not to be stored separately.

7 Experimental evaluation

Our self-tuning skeleton framework is evaluated as follows. First, it is measured how the backend service used affects the execution time by comparing the two different backend services supported by the prototypical implementation (namely MinIO and Redis). Second, the scalability of both example applications described in Sect. 5 is evaluated. Finally, the proposed proactive elasticity control mechanism is evaluated by assessing the accuracy of potential prediction models, which have been generated based on labeled performance measurement data, i.e., monitoring data obtained from previous application runs.

All our measurements were executed based on an Apache OpenWhisk installation hosted in an OpenStack-based private cloud environment. Our OpenWhisk cluster is operated on two Ubuntu 16.04 virtual machines (VM)

[11] https://github.com/odinsbane/least-squares-in-java.

[12] https://math.nist.gov/javanumerics/jama.

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. The hardware underlying the OpenStack-based cloud environment consists of identical servers, each equipped with two Intel Xeon E5-2650v2 CPUs and 128 GB RAM. The virtual network connecting tenant VMs is operated on a 10 Gbit/s physical ethernet network. Our experiments were performed during regular multi-tenant operation.

7.1 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 ($T_{seq} = 89.28$ seconds) with different degrees of parallelism. The application simply computes the numerical value of a definite integral of a quadratic polynomial. Figure 8 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. Note that, for an increasing degree of parallelism, the time difference between both setups becomes larger.

7.2 Parallel performance

We measured the parallel execution time for four instances of the numerical integration application (with different sequential execution times) with respect to different degrees of parallelism. The sequential execution time has been measured by deploying the serverless skeleton instance as a single FaaS function as described in Sect. 4.3 and depicted in Fig. 6. We measured the parallel execution time for four instances of the numerical integration application with a sequential runtime T_{seq} of (1) 1.02, (2) 9.78, (3) 89.28, and (4) 879.27 seconds with respect to different degrees of parallelism. Figure 9 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. 10. The application instances depicted have a sequential runtime T_{seq} of (1) 158.68, (2) 516.05, (3) 895.08, and (4) 2071.59 seconds. All parallel performance measurements were executed with the Redis backend service. To ensure parallel execution, we executed less worker FaaS functions in parallel then vCPUs available.

7.3 Proactive elasticity control

Our proactive elasticity controller groups tasks generated by the predecessor function according to the predicted number of processing units $\hat{p}$ (cf. Sect. 6.3 and Fig. 7). Consequently, the success of this approach heavily relies on the accuracy of the underlying prediction model. In the following, the model described in Sect. 6.2 is evaluated with respect to its accuracy. Therefore, the model parameters are estimated from monitoring data generated during the performance measurements of the hyperparameter optimization application (cf. Sect. 7.2). The prediction model (cf. Eq. 11) is fitted to a set of data records, with each record containing the input size and the measured execution time. With respect to the hyperparameter optimization application, the input size is defined as the number of hyperparameter configurations that have to be evaluated. The execution time is given in seconds.

Note that the number of model parameters is automatically selected by the model generation function by defining n and m (as discussed in Sects. 6.2 and 6.3). In this regard, several instances of the prediction model $\mathcal{P}_i, i \in \{1, 2, ..., 7\}$ with different sizes of the parameter vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are discussed in the following:

$$\mathcal{P}_i = \begin{cases} n = 0 \text{ and } m = 0 & \text{for} \quad i = 1 \\ n = 0 \text{ and } m = 1 & \text{for} \quad i = 2 \\ n = 1 \text{ and } m = 0 & \text{for} \quad i = 3 \\ n = 1 \text{ and } m = 1 & \text{for} \quad i = 4 \\ n = 1 \text{ and } m = 2 & \text{for} \quad i = 5 \\ n = 2 \text{ and } m = 1 & \text{for} \quad i = 6 \\ n = 2 \text{ and } m = 2 & \text{for} \quad i = 7 \end{cases} \tag{12}$$

To evaluate the accuracy of a prediction model instance, the measured number of processing units p is compared with the predicted number of processing units $\hat{p}$. Therefore, the model generation function randomly splits the data records to create a training data set, which contains 75% of the data records used to fit the model instances, and a test data set, which contains 25% of the data records used to assess the prediction accuracy. For the hyperparameter optimization application, 32 data records have been created by means of performance measurements. Thus, 24 data records are employed to train the model instances and 8 data records are employed to evaluate the accuracy. The calculated out-of-sample metrics mean-squared-error *MSE* and root-mean-squared-error *RMSE* are given in Table 1. In particular, the prediction model instances $\mathcal{P}_5$, $\mathcal{P}_6$, and $\mathcal{P}_7$ show a good accuracy with respect to the test data set. Prediction model instance $\mathcal{P}_5$ is selected by the model generation function because the accuracy of the model (in terms of the *RMSE*) is better than the accuracy of $\mathcal{P}_6$ and

Fig. 8 Measured execution time of numerical integration application instance based on MinIO/Redis backend service

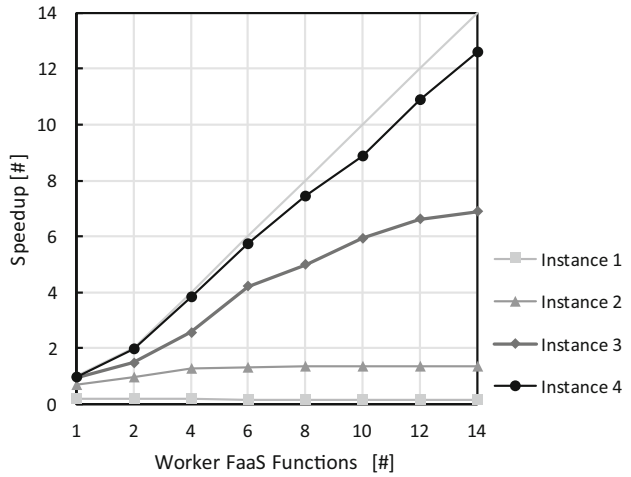

Fig. 9 Measured speedups of the numerical integration application with Redis backend service

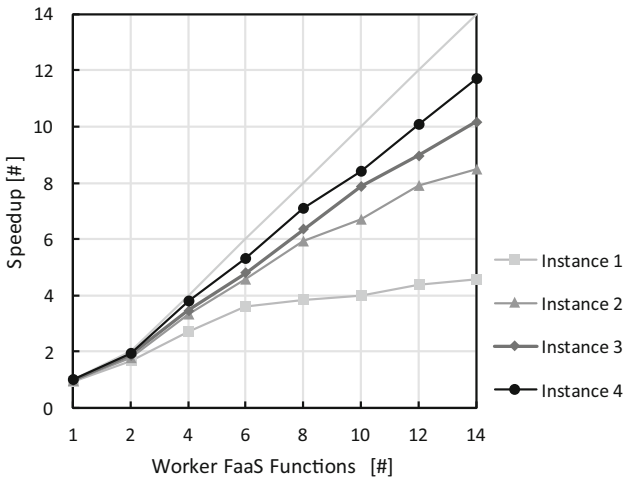

Fig. 10 Measured speedups of the hyperparameter optimization application with Redis backend service

cannot be increased by adding another parameter (cf. Table 1).

Table 1 Out-of-sample metrics calculated for the generated prediction model instances

	Number of parameters	*MSE*	*RMSE*
$\mathcal{P}_1$	2	14.2667	3.7771
$\mathcal{P}_2$	3	5.3883	2.3213
$\mathcal{P}_3$	3	0.4281	0.6543
$\mathcal{P}_4$	4	0.2650	0.5148
$\mathcal{P}_5$	5	0.0258	0.1605
$\mathcal{P}_6$	5	0.0499	0.2235
$\mathcal{P}_7$	6	0.0415	0.2036

Table 2 compares the predicted number of processing units $\hat{p}$ and the measured number of processing units p with respect to the test data set for model instances $\mathcal{P}_1$–$\mathcal{P}_7$. The parameters of model instance $\mathcal{P}_1$ are simply averaged across all data records leading to a poor accuracy. On the other hand, the accuracy of predictions increases for an increasing number of parameters. Model instance $\mathcal{P}_5$ shows the best accuracy with respect to the test data set. The accuracy of model instance $\mathcal{P}_5$ is also visualized in Fig. 11, which compares the predicted number of processing units $\hat{p}$ and the measured number of processing units p for the test data set according to the corresponding parallel execution time or execution time limit, respectively. Note that, because the prediction model does not provide integer values, the predicted number of processing units has to be rounded by the controller function. For model instance $\mathcal{P}_5$, the rounded predicted number of processing units is identical to the measured number of processing units for all data records of the test data set (cf. Table 2), which confirms the utility of the presented approach.

To show that the model also enables predictions with a sufficient level of accuracy even when trained with a few data records, we fitted model instance $\mathcal{P}_5$ to only 8 data records and evaluated the model instance with the remaining 24 data records. Here, the prediction accuracy in terms of the *RMSE* is 0.4359 and the rounded predicted number of processing units is identical to the measured number of processing units in 20/24 cases. In 2/20 cases, the prediction model proposes one processing unit more than actually required. Note that in such a case the user-defined execution time limit can still be met. For the other 2/20 cases, the prediction model proposes one processing unit less than actually required. The measured and predicted numbers of processing units are also visualized in Fig. 12.

In order to show that the model enables predictions with a sufficient level of accuracy even in case of an input size that has not been considered in the training, we also fitted model instance $\mathcal{P}_5$ to 24 data records and evaluated the

Table 2 Comparison of the measured and predicted numbers of processing units for prediction model instances $\mathcal{P}_1$–$\mathcal{P}_7$, which have been fitted to 24 data records

Data record index	Measured number of processing units p [#]	Predicted number of processing units $\hat{p}$ [#]						
		$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$	$\mathcal{P}_6$	$\mathcal{P}_7$
1	2	8.8584	6.7740	1.8910	1.8341	1.8380	1.8612	1.8562
2	12	10.2918	9.4830	11.6340	11.7220	11.9082	11.9067	11.9062
3	2	6.1059	3.8573	2.0766	2.2030	2.1102	1.9516	1.9788
4	4	7.7733	6.1827	4.1892	4.3801	4.1722	3.9234	3.9657
5	12	9.5345	10.4397	13.4336	13.2111	12.3068	12.3925	12.3626
6	2	2.6898	1.2286	1.8970	2.0553	1.9822	2.1125	2.0888
7	6	7.2890	6.4044	5.8920	6.1268	5.8886	5.7331	5.7569
8	14	9.0132	12.3949	15.0781	14.5881	14.1530	14.3549	14.3009

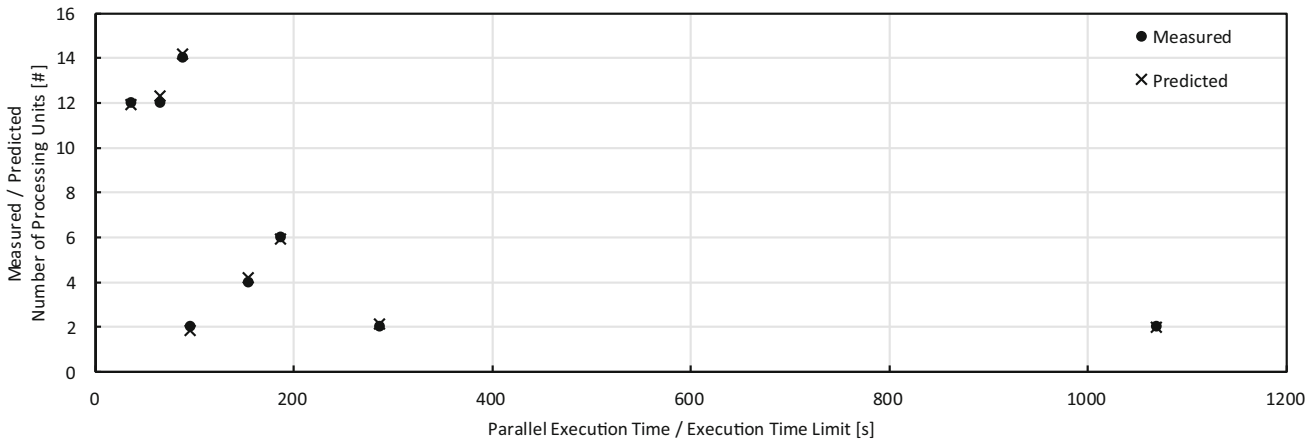

Fig. 11 Comparison of the measured and predicted numbers of processing units for prediction model instance $\mathcal{P}_5$, which has been fitted to 24 data records

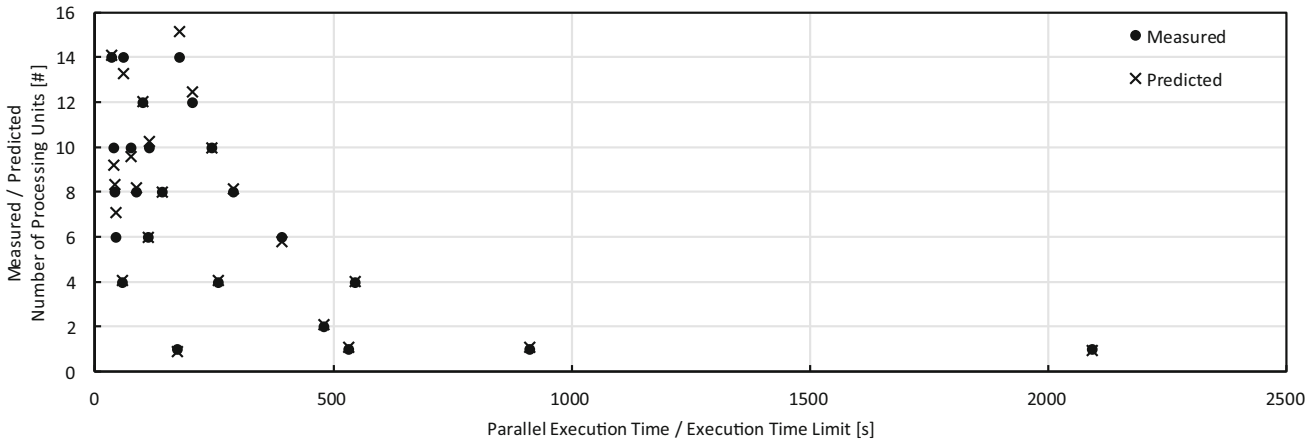

Fig. 12 Comparison of the measured and predicted numbers of processing units for prediction model instance $\mathcal{P}_5$, which has been fitted to only 8 data records

model instance with the 8 data records related to a previously unseen input size. Therefore, we split the data records thus that all measurements of application instance 3 (cf. Fig. 10) are contained in the test data set and trained the model based on the remaining data records. The resulting prediction accuracy in terms of the *RMSE* is

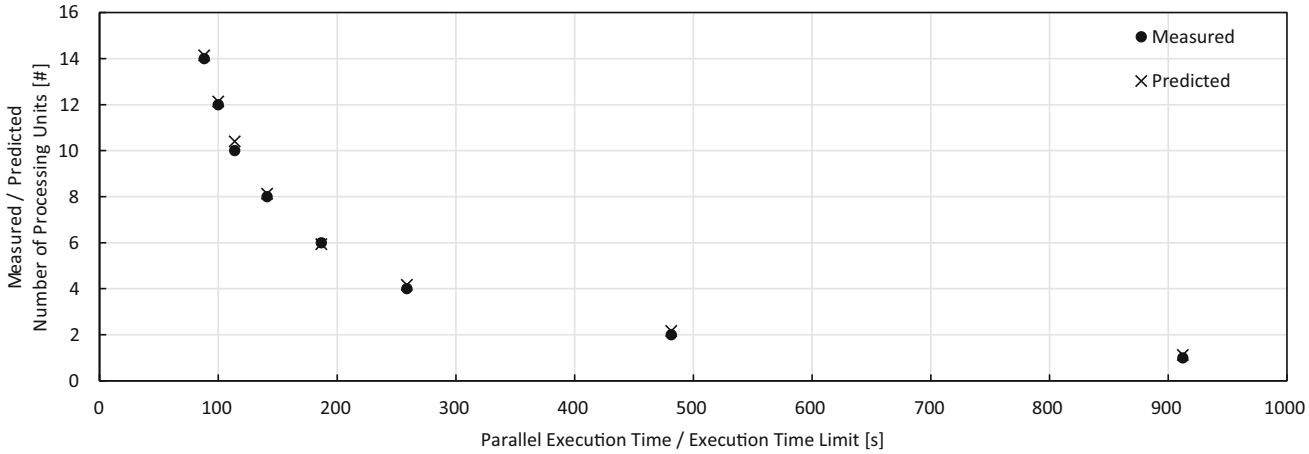

Fig. 13 Comparison of the measured and predicted numbers of processing units for prediction model instance $\mathcal{P}_5$, which has been fitted to 24 data records and evaluated with 8 data records related to a previously unseen input size

0.1924, which shows that predictions with a sufficient level of accuracy are also possible for a previously unseen input size, which has not been considered to train the model. The measured and predicted numbers of processing units are also visualized in Fig. 13.

8 Findings and discussion

In this section, we discuss the assumptions underlying our proactive elasticity control mechanism specifically designed for serverless task farming as well as its applicability to other classes of parallel applications.

We employ a *black-box* approach based on regression techniques to automatically derive a model of the scaling behavior of serverless task farming applications. This model is used by the elasticity controller to predict the required number of processing units (worker FaaS functions) per application run according to a user-defined execution time limit. Our regression model is based on Amdahl's law [6], which explains the non-linear scaling behavior of parallel applications by modeling the execution time of a sequential program part t_s and the execution time of a parallelizable program part t_p. As discussed in Sect. 6.2, Amdahl's model is well-suited for serverless task farming applications, for which the parallelizable program part t_p can be considered as the execution time of the worker FaaS functions and the execution time of the sequential program part t_s can be considered as the execution time of all other functions, which are executed sequentially. A major benefit of our simple regression model is that it does not require knowledge about the internals of the application or additional modeling by users while providing predictions with a sufficient level of accuracy even when trained only with a few data records. However, note that this simple model does not explicitly consider additional sources of overhead, which might, in general, also increase when a higher number of processing units is employed for parallel computations: For a very high number of processing units, the parallel execution time might be increased again due to this additional overhead. Our regression model implicitly assumes that parallel overhead stemming from communication is negligible for the application class considered. This is a reasonable assumption for serverless task farming applications because communication (for task distribution and result aggregation) is implemented based on shared backend services, which have been shown to scale almost linearly with the number of FaaS functions [28]. Moreover, communication across FaaS functions is infrequent and typically only small data volumes (such as task descriptions and results) have to be transferred. Nevertheless, whereas our experimental results show that this simple model enables accurate predictions for task farming applications, other models might be required for other classes of parallel applications, e.g., to explicitly consider communication overhead. We plan to extend our approach to consider other classes of parallel applications in the future. In this context, also additional skeleton types and corresponding execution models are required to benefit from serverless computing platforms.

Because proactive elasticity control mechanisms in general heavily rely on prediction models, their applicability is limited to applications that allow accurate predictions of the number of processing units based on input data. This also means that not all parallel applications can benefit from proactive elasticity control. Examples are parallel applications based on branch-and-bound and backtracking search that target combinatorial search problems. Because these applications exhibit a high degree

of irregularity, their execution time and scaling behavior are hard to predict, which consequently also limits the applicability of proactive elasticity control mechanisms. We addressed the challenges related to elasticity control for these applications in our recent research by designing reactive elasticity control mechanisms, which are presented in [21, 36]. Whereas proactive elasticity control mechanisms rely on prediction models to select the number of processing units, reactive elasticity control mechanisms dynamically adapt the number of processing units based on measured runtime metrics. With such an approach predicting the required number of processing units is not necessarily required. However, note that reactive mechanisms typically lead to additional overhead and thus proactive elasticity control should be preferred for applications for which prediction models provide a sufficient level of accuracy.

Moreover, the proactive elasticity control mechanism has several programming and system level implications. According to Amdahl's law, the parallelizable program part t_p can benefit from each added processing unit. However, in a realistic scenario, the physical parallelism (i.e., the number of processing units) that can be employed efficiently is limited by the logical parallelism (i.e., the number of tasks) provided by the application. In our case, the number of tasks generated by the predecessor function (cf. Sect. 4.1) limits the maximum number of worker FaaS functions that can be employed efficiently. We assume that at least one task can be assigned to each worker FaaS function and that all tasks have the same size. This is a reasonable assumption for task farming applications because most often a large number of similar tasks is generated. Nevertheless, developers should consider this aspect by making use of overdecomposition, i.e., generate fine-grained tasks, whenever possible. These fine-grained tasks are then automatically grouped by the controller function (as discussed in Sect. 6.3). Additionally, if the number of tasks cannot be distributed evenly across processing units, this might lead to additional side effects on the measured scaling behavior. Note that a higher number of fine-grained tasks automatically ensures an (almost) even distribution across worker FaaS functions.

9 Related work

We have identified different fields of related work: Existing work considering (1) serverless computing for parallel applications, (2) skeleton frameworks and management approaches, and (3) mechanisms for application management, performance modeling, and prediction.

Serverless computing Serverless computing platforms promise integrated auto-scaling and transparent resource management, but are mainly employed to operate interactive and event-driven applications. More recently, serverless computing platforms have become of interest for parallel processing. The authors of [28] state that many large-scale parallel applications are able to exploit serverless cloud offerings with high bandwidth and high latency object storage as a substitute for distributed memory. Specifically, the authors show that the read / write bandwidth of Amazon S3 scales linearly with the number of FaaS functions getting on average 40 MB/s read operations and 30 MB/s write operations per FaaS function. The authors present a prototype called PyWren that enables developers to make use of AWS Lambda for parallel execution of locally developed code segments. The authors of [50] adapted PyWren to be used with IBM Cloud Functions. The resulting framework has been additionally optimized for MapReduce jobs. The authors of [51] describe how to execute linear algebra algorithms on AWS Lambda. In [54], serverless computing platforms are evaluated for big data processing use cases based on a matrix multiplication application. None of the aforementioned approaches investigates on proactive elasticity control mechanisms for task farming applications.

Skeleton frameworks and management Algorithmic skeletons [14, 19] 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. Consequently, one can say that each skeleton comprises a built-in parallel behavior [17]. 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 [17, 39]. Over the years, many frameworks and libraries have been developed for a variety of programming languages and parallel execution environments [2, 5, 8, 18]. 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. A well-known example is Muesli [39], which is a C++ template library that supports parallel execution on top of MPI, OpenMP, and CUDA. Existing work also shows how to enhance the concept of algorithmic skeletons with automated management solutions. The authors of [4] discuss how to integrate autonomic management of non-functional concerns into algorithmic skeletons. An abstract control loop is described that allows system programmers to compose a given set of

monitoring and actuation actions into management rules. Moreover, the adaptation of a skeleton instance at runtime is discussed. The behavioral skeleton concept [3] is utilized to combine skeletons as higher-level programming abstractions with autonomic managers technically based on a rule engine. The authors of [16] employ the behavioral skeleton concept in the context of the farm skeleton. Their approach considers a WorkpoolService for which the service time is optimized based on specified rules. All these approaches also propose automated management solutions for skeleton-based applications, but do not target serverless computing platforms and also do not provide proactive elasticity control mechanisms. They are rather based on rule-based techniques.

Application management, performance modeling, and prediction A comprehensive survey and classification of workload forecasting methods is presented in [44]. Several approaches show how to consider a user-defined execution time limit (or deadline) in the context of managing workflows [30, 46, 52] and parallel applications [49]. An elasticity controller for iterative parallel applications is presented in [15], which detects workload patterns by comparing the last two average load values calculated based on monitored time series data and simple exponential smoothing. Related work considering performance prediction in the context of parallel applications employs different techniques such as linear regression, support vector machines (SVMs), decision trees, and artificial neural networks [45]. The authors of [41] use neural networks to predict the task execution time in the context of operational cost minimization for hybrid clouds. The authors of [10, 56] use prediction models to extrapolate the performance of an application that solves a problem larger than the problems used for the measurements (to generate training data). The authors of [55] generate an application model from log data, which is then used to predict an application's performance for different execution environments. In [25], performance and costs of parallel applications in cloud environments are predicted per application run based on a given application specification. The presented approach automatically selects the optimal resource configuration among a set of defined combinations. The authors of [42] propose prediction models to find the best resource configuration for a specific application, which can be offered by cloud providers. Their approach is implemented based on random forests, which automatically select the predictors during model construction. Cloud users can use the prediction model to choose a resource configuration for their application. In contrast to this approach, in this work, prediction models are employed to construct a proactive elasticity control mechanism that automatically selects the optimal number of processing units according to user-defined goals. The authors of [27] present a cluster scheduling policy that considers the scaling behavior of applications to increase the efficiency. Therefore, they model the scaling behavior based on Amdahl's law [6] with the goal to maximize the sum of the speedups of all jobs. Whereas this approach enables optimization from a cluster operator perspective considering all jobs, we focus on application-specific optimization in cloud environments from a cloud customer perspective. The authors of [22] also model the scaling behavior based on Amdahl's law in the context of a novel cost model for quantifying the monetary costs of executing parallel applications with volatile cloud resources. In this work, serverless computing platforms are targeted.

10 Conclusion

In this work, we discuss a novel approach that enables elastic parallel processing without considering parallelism or resource management issues. Based on the well-known concept of algorithmic skeletons, parallel applications, which require coordination, communication, and synchronization, can benefit from serverless computing platforms. The prototypical development and runtime framework shows how to apply the presented concepts to implement self-tuning serverless task farming. A proactive elasticity controller handles the cost/efficiency-time trade-off in an automated manner by predicting the required number of processing units to meet a user-defined execution time limit after which the result of the computation needs to be present while minimizing the associated monetary costs. The underlying prediction model is obtained and refined in an automated manner by employing supervised learning to infer the scaling behavior from labeled performance measurement data of previous application runs.

Whereas the experimental evaluation shows very promising results for task farming applications, 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 could be offered by cloud providers, which store data in close physical proximity to FaaS functions (e.g., on the same rack). Moreover, explicitly modeling the communication overhead might be required for other classes of parallel applications to ensure accurate predictions. Finally, current serverless computing platforms do not support the use of specialized hardware accelerators. However, it is expected that serverless computing platforms will support these in the future. For

instance, hyperparameter optimization would substantially benefit from GPU-enabled training of large artificial neural networks.

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

References

1. Aggarwal, C.C.: Data Mining: The Textbook. Springer, New York (2015)
2. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in java. Future Gener. Comput. Syst. **19**(5), 611–626 (2003)
3. Aldinucci, M., Campa, S., Danelutto, M., Vanneschi, M., Kilpatrick, P., Dazzi, P., Laforenza, D., Tonellotto, N.: Behavioural skeletons in GCM: autonomic management of grid components. In: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008). IEEE, pp 54–63 (2008)
4. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Co-design of distributed systems using skeleton and autonomic management abstractions. In: César, E., Alexander, M., Streit, A., Träff, J.L., Cérin, C., Knüpfer, A., Kranzlmüller, D., Jha, S. (eds.) Euro-Par 2008 Workshops—Parallel Processing, pp. 403–414. Springer, Heidelberg (2009)
5. Alexandre, F., Marques, R., Paulino, H.: On the support of task-parallel algorithmic skeletons for multi-GPU computing. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, ACM, New York, NY, USA, SAC '14, pp. 880–885 (2014)
6. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the Spring Joint Computer Conference, ACM, New York, NY, USA, AFIPS '67 (Spring), pp. 483–485 (1967)
7. Atkinson, K.E.: An Introduction to Numerical Analysis, 2nd edn. Wiley, New York (1989)
8. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P3 l: a structured high-level parallel language, and its structured support. Concurrency **7**(3), 225–255 (1995)
9. Barcelona-Pons, D., Sánchez-Artigas, M., París, G., Sutra, P., García-López, P.: On the faas track: building stateful distributed applications with serverless architectures. In: Proceedings of the 20th International Middleware Conference, ACM, New York, NY, USA, Middleware '19, pp. 41–54 (2019)
10. Barnes, B.J., Rountree, B., Lowenthal, D.K., Reeves, J., de Supinski, B., Schulz, M.: A regression-based approach to scalability prediction. In: Proceedings of the 22nd Annual International Conference on Supercomputing, ACM, ICS '08, pp. 368–377 (2008)
11. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. J. Mach. Learn. Res. **13**, 281–305 (2012)
12. Bergstra, J., Yamins, D., Cox, D.D.: 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—vol. 28, ICML'13, pp. I–115–I–123 (2013)
13. Berk, R.A.: Statistical Learning from a Regression Perspective, 2nd edn. Springer, New York (2016)
14. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1991)
15. da Rosa, R.R., Rodrigues, V.F., Rostirolla, G., da Costa, C.A., Roloff, E., Navaux, P.O.A.: A lightweight plug-and-play elasticity service for self-organizing resource provisioning on parallel applications. Future Gener. Comput. Syst. **78**, 176–190 (2018)
16. Danelutto, M., Zoppi, G.: Behavioural skeletons meeting services. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) Computational Science—ICCS 2008, pp. 146–153. Springer (2008)
17. Darlington, J., Field, A.J., Harrison, P.G., Kelly, P.H.J., Sharp, D.W.N., Wu, Q., While, R.L.: Parallel programming using skeleton functions. In: Bode, A., Reeve, M., Wolf, G. (eds.) PARLE '93 Parallel Architectures and Languages Europe, pp. 146–160. Springer, Heidelberg (1993)
18. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software **40**(12), 1135–1160 (2010)
19. Gorlatch, S., Cole, M.: Parallel skeletons. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1417–1422. Springe, Boston (2011)
20. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Pearson Education, London (2003)
21. Haussmann, J., Blochinger, W., Kuechlin, W.: Cost-efficient parallel processing of irregularly structured problems in cloud computing environments. Clust. Comput. **22**(3), 887–909 (2019a)
22. Haussmann, J., Blochinger, W., Kuechlin, W.: Cost-optimized parallel computations using volatile cloud resources. In: Djemame, K., Altmann, J., Bañares, J.Á., Agmon Ben-Yehuda, O., Naldi, M. (eds.) Economics of Grids, Clouds, Systems, and Services, pp. 45–53. Springer, Cham (2019b)
23. Haussmann. J., Blochinger, W., Kuechlin, W.: An elasticity description language for task-parallel cloud applications. In: Proceedings of the 10th International Conference on Cloud Computing and Services Science, SciTePress, pp. 473–481 (2020)
24. Hellerstein, J.M., Faleiro, J.M., Gonzalez, J., Schleier-Smith, J., Sreekanti, V., Tumanov, A., Wu, C.: 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)
25. Huang, H., Wang, L., Tak, B.C., Wang, L, Tang C.: CAP3: a cloud auto-provisioning framework for parallel processing using on-demand and spot instances. In: 2013 IEEE Sixth International Conference on Cloud Computing. IEEE, pp. 228–235 (2013)
26. Humble, J., Farley, D.: Continuous delivery: reliable software releases through build, test, and deployment automation. Addison-Wesley, Boston (2010)
27. Hungershöfer, J., Streit, A., Wierum, J.M.: Efficient resource management for malleable applications. Tech. Rep. TR-003-01, Paderborn Center for Parallel Computing (2001)
28. Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., Recht, B.: Occupy the cloud: distributed computing for the 99%. In: Proceedings of the 2017 Symposium on Cloud Computing, ACM, New York, NY, USA, pp. 445–451 (2017)
29. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J.E., Popa, R.A., Stoica, I., Patterson, D.A.: (2019) Cloud programming simplified: a berkeley view on serverless computing
30. Kalyan Chakravarthi, K., Shyamala, L., Vaidehi, V.: Budget aware scheduling algorithm for workflow applications in IaaS clouds. Clust. Comput. (2020)
31. Kehrer, S., Blochinger, W.: Elastic parallel systems for high performance cloud computing: state-of-the-art and future directions. Parallel Process. Lett. **29**(02), 1950006-1 (2019a)
32. Kehrer, S., Blochinger, W.: Migrating parallel applications to the cloud: assessing cloud readiness based on parallel design

decisions. SICS Softw.-Intensive Cyber-Phys. Syst. **34**(2), 73–84 (2019b)
33. Kehrer, S., Blochinger, W.: 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, pp. 57–69 (2019c)
34. Kehrer, S., Blochinger, W.: 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, pp. 198–209 (2019d)
35. Kehrer, S., Blochinger, W.: Development and operation of elastic parallel tree search applications using taskwork. In: Ferguson, D., Méndez Muñoz, V., Pahl, C., Helfert, M. (eds.) Cloud Comput. Serv. Sci., pp. 42–65. Springer International Publishing, Cham (2020a)
36. Kehrer, S., Blochinger, W.: Equilibrium: an elasticity controller for parallel tree search in the cloud. J. Supercomput. (2020b)
37. Kehrer, S., Riebandt, F., Blochinger, W.: Container-based module isolation for cloud services. In: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 177–186 (2019a)
38. Kehrer, S., Scheffold, J., Blochinger, W.: Serverless skeletons for elastic parallel processing. In: 2019 IEEE 5th International Conference on Big Data Intelligence and Computing (DATACOM). IEEE, pp. 185–192 (2019b)
39. Kuchen, H.: Parallel programming with algorithmic skeletons. In: Bergener, K., Räckers, M., Stein, A. (eds.) The Art of Structuring: Bridging the Gap Between Information Systems Research and Practice, pp. 527–536. Springer International Publishing, Cham (2019)
40. Levenberg, K.: A method for the solution of certain non-linear problems in least squares. Q. Appl. Math. **2**(2), 164–168 (1944)
41. Li, C., Tang, J., Luo, Y.: Towards operational cost minimization for cloud bursting with deadline constraints in hybrid clouds. Clust. Comput. **21**(4), 2013–2029 (2018)
42. Mariani, G., Anghel, A., Jongerius, R., Dittmann, G.: Predicting cloud performance for HPC applications before deployment. Future Gener. Comput. Syst. **87**, 618–628 (2018)
43. Marquardt, D.W.: An algorithm for least-squares estimation of nonlinear parameters. SIAM J. Appl. Math. **11**(2), 431–441 (1963)
44. Masdari, M., Khoshnevis, A.: A survey and classification of the workload forecasting methods in cloud computing. Clust. Comput. (2019)
45. Matsunaga, A., Fortes, J.A.B.: On the use of machine learning to predict the time and resources consumed by applications. In: 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 495–504 (2010)
46. Mortazavi-Dehkordi, M., Zamanifar, K.: Efficient deadline-aware scheduling for the analysis of big data streams in public cloud. Clust. Comput. **23**(1), 241–263 (2020)
47. Poldner, M., Kuchen, H.: On implementing the farm skeleton. Parallel Process. Lett. **18**(01), 117–131 (2008)
48. Rajan, D., Thain, D.: Designing self-tuning split-map-merge applications for high cost-efficiency in the cloud. IEEE Trans. Cloud Comput. **5**(2), 303–316 (2017)
49. Raveendran, A., Bicer, T., Agrawal, G.: A framework for elastic execution of existing MPI programs. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pp. 940–947 (2011)
50. Sampé, J., Vernik, G., Sánchez-Artigas, M., García-López, P.: Serverless data analytics in the ibm cloud. In: Proceedings of the 19th International Middleware Conference Industry, ACM, Middleware '18, pp. 1–8 (2018)
51. Shankar, V., Krauth, K., Pu, Q., Jonas, E., Venkataraman, S., Stoica, I., Recht, B., Ragan-Kelley, J.: Numpywren: serverless linear algebra. CoRR abs/1810.09679 (2018)
52. Sun, T., Xiao, C., Xu, X.: A scheduling algorithm using sub-deadline for workflow applications under budget and deadline constrained. Clust. Comput. **22**(3), 5987–5996 (2019)
53. van Eyk, E., Toader, L., Talluri, S., Versluis, L., Ută, A., Iosup, A.: Serverless is more: from paas to present cloud computing. IEEE Internet Comput. **22**(5), 8–17 (2018)
54. Werner, S., Kuhlenkamp, J., Klems, M., Müller, J., Tai, S.: Serverless big data processing using matrix multiplication as example. In: 2018 IEEE International Conference on Big Data, pp. 358–365 (2018)
55. Wong, A., Rexachs, D., Luque, E.: Parallel application signature for performance analysis and prediction. IEEE Trans. Parallel Distrib. Syst. **26**(7), 2009–2019 (2015)
56. Wu, X., Mueller, F.: Scalaextrap: trace-based communication extrapolation for spmd programs. SIGPLAN Not. **46**(8), 113–122 (2011)

Stefan Kehrer is a Ph.D. student in the Parallel and Distributed Computing Group at Reutlingen University and does research in the fields of Cloud Computing, Parallel Computing, and Distributed Systems. He is a member of the Doctoral Program Services Computing and affiliated with the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart. He aims to develop novel concepts and methods for the design, development, and management of cloud-aware parallel applications and systems.

Dominik Zietlow is pursuing his Ph.D. in the Autonomous Learning Group at the Max-Planck-Institute for Intelligent Systems Tuebingen with a focus on deep representation learning. He is a scholar of the International Max Planck Research School for Intelligent Systems (IMPRS-IS).

Jochen Scheffold received his M.Sc. degree in Business Informatics from Reutlingen University in 2019. He investigates on how parallel applications can benefit from serverless computing platforms and is interested in novel approaches to parallel cloud programming.

Wolfgang Blochinger is a professor of Computer Science at Reutlingen University, Germany. He received his Ph.D. in 2002 and his Habilitation in 2008, both from the University of Tuebingen, Germany. He leads the Parallel and Distributed Computing Group at Reutlingen University. His research interests include high-performance systems, parallel and distributed application design, as well as Grid and Cloud Computing.