An Elasticity Description Language for Task-parallel Cloud Applications

Jens Haussmann[1][a], Wolfgang Blochinger[1][b] and Wolfgang Kuechlin[2][c]
[1]*Parallel and Distributed Computing Group, Reutlingen University, Germany*
[2]*Symbolic Computation Group, University of Tuebingen, Germany*

Keywords: Cloud Computing, High Performance Computing, Parallel Application, Elasticity.

Abstract: In recent years, the cloud has become an attractive execution environment for parallel applications, which introduces novel opportunities for versatile optimizations. Particularly promising in this context is the elasticity characteristic of cloud environments. While elasticity is well established for client-server applications, it is a fundamentally new concept for parallel applications. However, existing elasticity mechanisms for client-server applications can be applied to parallel applications only to a limited extent. Efficient exploitation of elasticity for parallel applications requires novel mechanisms that take into account the particular runtime characteristics and resource requirements of this application type. To tackle this issue, we propose an *elasticity description language*. This language facilitates users to define elasticity policies, which specify the elasticity behavior at both cloud infrastructure level and application level. Elasticity at the application level is supported by an adequate programming and execution model, as well as abstractions that comply with the dynamic availability of resources. We present the underlying concepts and mechanisms, as well as the architecture and a prototypical implementation. Furthermore, we illustrate the capabilities of our approach through real-world scenarios.

1 INTRODUCTION

Cloud computing evolved into a mature computing paradigm that has turned out to be a promising technological path heading the utility computing vision. Employing cloud environments for executing parallel applications is particularly promising since properties like pay-per-use, on-demand access, and elasticity open up new opportunities. For instance, one can explicitly control and optimize monetary costs on the level of individual parallel application runs. However, existing parallel applications have almost never been designed for execution in cloud environments. It is still not completely known in which ways and to what extent they can profit from the opportunities that have been emerged from cloud computing. This issue motivated an increasing activity in a wide-ranging research domain concerning the exploitation of cloud characteristics for parallel applications.

In principle, several classes of parallel applications can exploit cloud characteristics with no or few modifications, like some HPC applications, such as e.g., in the following scenario: By employing a simple ”copy and paste” approach, users can substitute their HPC infrastructure by virtual hardware, harnessing the on-demand self-service and pay-per-use characteristics of the cloud model. This approach provides several benefits: First, the user only pays for actually used resources. This characteristic is well suited for institutions that otherwise would have to deal with underutilized resources or a restricted budget that prevents an investment for on-site clusters. Second, the *on-demand self-service* characteristic of cloud offerings allows additional execution scenarios. For example, jobs submitted to grids or clusters are typically handled by a scheduling system, stored in a queue and executed later when resources become available. In contrast, unlimited and immediately available resources in cloud environments allow the execution of all jobs in parallel, avoiding waiting times.

In this work, we focus on a cloud characteristic of fundamental significance for parallel applications: Elasticity. For parallel applications elasticity is a new concept, to which our research contributes towards its understanding and utilization. Elasticity is commonly defined as the degree to which a system can adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time, the available resources match the current demand as closely as possible. To

[a] https://orcid.org/0000-0003-0986-7594
[b] https://orcid.org/0000-0002-5946-7225
[c] https://orcid.org/0000-0002-5469-5544

date, there exists a substantial body of research that addresses various aspects of elasticity in cloud computing. However, previous studies mostly focused on elasticity regarding traditional client-server cloud applications, like web-, mail-, and database servers. Despite its promising potential, implications and opportunities of elasticity on parallel applications have not been studied in depth. To benefit from elasticity, it is not sufficient to focus solely on the elasticity of the infrastructure. Instead, parallel applications are also required to consider the application's internal structure and runtime behavior. To this end, it is mandatory to support elasticity at the application level by providing adequate programming and execution models as well as abstractions that take into account the dynamic availability of resources.

The main contributions of this paper are:

- We describe an approach of elasticity mechanisms that control the logical parallelism of the application and the physical parallelism of the cloud infrastructure.
- We show how to efficiently leverage elasticity for task-parallel applications in cloud environments.
- We develop an elasticity description language for parallel applications in the cloud with a corresponding programming model and report on the experimental evaluation.

The remainder of the paper is organized as follows: In Section 2, we motivate our study by discussing in more detail the specific problem we are addressing. Section 3 describes in detail our approach for comprehensive elasticity management of parallel cloud applications. Later, in Section 4, we discuss several use-cases to substantiate the capabilities of our proposed elasticity mechanisms. In Section 5, we report on our experimental evaluation. Next, Section 6 gives an overview of related work. Section 7 concludes the paper and outlines directions for future research.

2 PROBLEM STATEMENT

Among all characteristics that paved the way for the success of cloud computing, elasticity is widely considered to be the most fundamental of all. It is a novel characteristic in resource management that distinguishes the cloud from other computing paradigms. Elasticity offers the ability to allocate computing resources dynamically, according to the current demand. This introduces a wide range of new opportunities to optimize individual executions of an application with changing resource requirements.

Generally, elasticity can be profitable for applications with dynamic resource demand during execution. However, while many applications possess this property, previous work has mostly been limited to client-server applications. Typically, these are client-server applications such as web servers, e-mail servers, and database servers whose workload is defined by external and independent user requests. The rate at which these requests arrive constitutes the application's resource demand. Since the arrival rate can change over time, static resource capacity can lead to over-/under-provision of resources. In such situations, these applications profit from the elasticity of cloud infrastructures by dynamically matching resource allocation to the current resource demand.

For parallel applications, however, elasticity is a new concept to which our research contributes towards understanding and utilization. Unlike client-server applications, the workload of parallel applications is defined by a set of input parameters, including a description of the problem to be processed. The left part of Figure 1 summarizes the most important factors that necessitate leveraging elasticity mechanisms for parallel applications. Parallel applications also differ from client-server applications in certain other aspects. Typically, they are executed on dedicated HPC clusters consisting of a static number of processing elements. In addition, they are often dependent on a particular type of hardware and utilize optimized runtime systems, programming models, middleware, and software libraries (Blochinger et al., 1999; Blochinger et al., 1998). However, these instruments do not sufficiently support the elasticity mechanisms present in cloud environments. Such limitations prevent the exploitation of elasticity without a major redesign of the existing parallel application.

In order to exploit elasticity, mechanisms are needed at both the infrastructure and the application level. Cloud providers offer APIs at the infrastructure level to control the number of processing elements, which we refer to as physical parallelism. To leverage these processing elements, the application must provide a sufficient number of tasks, which we call logical parallelism. For efficient parallel computations, the degree of logical parallelism must match the degree of physical parallelism. Controlling both levels is a complex task that can be considerably simplified with elasticity policies, Elasticity policies express the requirements on both parallelism levels and their enforcement in a descriptive way.

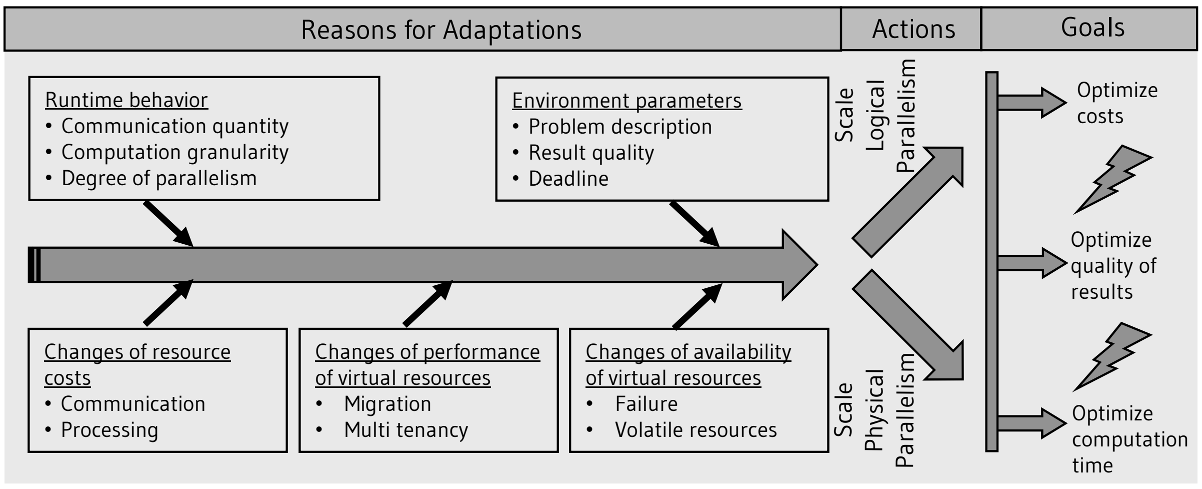

Figure 1: Elasticity of parallel applications.

3 ENABLING ELASTICITY FOR TASK-PARALLEL COMPUTATIONS

In this section, we present our solution that enables efficient exploitation of elasticity for task-parallel applications. Particularly, we focus on the discussion of the essential components and their interaction. One important aspect is the programming model, which has to offer mechanisms for taking into account elasticity at the program structure level, along with an appropriate model of the parallel execution. Additionally, we present the *elasticity description language* (EDL) that enables users to define the elasticity behavior of the parallel system.

3.1 Programming and Execution Model

In this work, we employ the task pool execution model for task management and load balancing. A task pool consists of a set of processing elements, each of which maintains a queue for storing tasks. Task decomposition and processing is performed by each processing element individually. Generated tasks are stored in the local queue, and any processing element may fetch them for processing through work-stealing. The resulting dynamic mapping of tasks to processing elements decouples problem decomposition and processing.

Programming models for parallel applications can be classified according to their abstraction level in terms of parallelism. According to (Skillicorn and Talia, 1998), there are six different abstraction levels, ranging from models that completely abstract parallelism (level 1) to models in which parallelism is completely explicit (level 6). In this work, we employ a modified version of the fork-join programming model, being at abstraction level 2. This model is also used by Cilk (Blumofe et al., 1995) and OpenMP (Galante and Bona, 2014). At this abstraction level, the parallelism in the program is marked explicitly, while task generation, mapping, and communication is implicit. Thus, developers are aware that the application is running in parallel, while they only need to express that part of the code that might run in parallel. The fork-join statements allow developers to specify potential parallelism at the program level, which can be transformed into *logical parallelism* in the form of tasks. In the basic fork-join model, the degree of logical parallelism increases whenever a fork statement is encountered. Commonly, there is a fixed relationship between calling the fork statement and instantiating new tasks, with each fork typically instantiating a single task. In the following, we will refer to this relationship as *fork policy*. An accurate formulation of the fork policy is essential, as creating new tasks results in overhead in the form of excess computation and communication. A fixed fork policy is well suited for executions with a static number of processing elements, since the policy for fork statements and thus the granularity of the tasks can be specified prior to execution. However, cloud infrastructures feature a dynamic resource capacity and therefore require a dynamic policy for efficient parallel execution, and will be discussed in the next section. The parallelism levels which have to be aligned are therefore: 1) Parallelism at the program level, specified with the fork-join programming model. 2) Logical parallelism through the tasks created by the task pool. 3) Physical parallelism through the processing elements of the cloud infrastructure.

3.2 Elasticity Description Language

From a cloud user's perspective, elasticity requirements of a cloud system are often specified in a template file written in a markup language such as *YAML*. This specification contains a set of conditions that the system must fulfill together with appropriated scaling operations to be carried out in the case of violation. A fundamental advantage of this method lies in its straightforward usability. The specification file is read, interpreted, and automatically enforced by services from the cloud provider (e.g., *heat*), shielding the user from details of the implementation. Hence, elasticity is enabled in a transparent manner, i.e., users do not have to deal with burdens like modifications of the applications' source code, monitoring tools, and APIs for interaction with the cloud infrastructure.

Although there exist several languages for the specification of elasticity requirements, most of them focus on elasticity for client-server applications with elasticity mechanisms operating at the infrastructure level. Hitherto, there are only a few languages capable of defining elasticity requirements on multiple levels of cloud systems, like *SYBL* (Copil et al., 2015) The *SYBL* language defines several constructs that enable the specification of multi-level elasticity requirements for cloud services. While several implementations of these constructs have been developed to date, none of them has yet addressed the elasticity specification at the parallel programming model and infrastructure level.

We remedy this limitation and propose our *elasticity description language (EDL)* that provides elasticity support for fork-join applications. The language facilitates detailed elasticity specifications of the parallel system with logical and physical parallelism being the control levers. A template file written in the EDL is called an *elasticity policy*. This policy describes the scope of states within which it should be operating and the elastic mechanisms through which this is achieved. The structure of our language is primarily based on the languages of *YAML*, *YAQL*, and *HOT*. To control elasticity beyond the hardware level, we use the same level of abstraction for our components as in *SYBL*. We provide three core components for the construction of elasticity policies, which are described in more detail below: *Monitor*, *constraint*, and *strategy*. The basic structure of a policy is shown in Listing 1. Both, constraints and strategies have configurable properties and are grouped into two separate lists within a policy. In contrast to constraints

```
constraints:
  - [Constraint 1]
  - [Constraint 2]
     ...
  - [Constraint n]
↪
```

```
↪
strategies:
  - [Strategy 1]
  - [Strategy 2]
     ...
  - [Strategy n]
```

Listing 1: Basic structure of an elasticity policy.

and strategies, monitors are implicitly available and do not need to be explicitly defined in a policy before use. We will now move on to a more detailed discussion of the use and concepts of the introduced components from our EDL.

A monitor is applied to check the current state of a particular component within the parallel system. Traditionally, monitors gather information related to the employed cloud resources at the infrastructure level like CPU-utilization, network bandwidth, or disk I/O. However, since our EDL addresses both physical and logical parallelism, application-level monitoring is also required for efficient elasticity management. Monitors at the infrastructure level, such as processor utilization, are not further elaborated here as the cloud provider usually provides them. In the following, we discuss in more detail the information gathered by the monitors at the application level. At the application level, there are several categories within the task pool execution model where monitor implementations enable insights of significant relevance:

- *Load Balancing:* No. of task stealing attempts, no. of received tasks, response time, consumed CPU-time, etc.
- *Local Task Queue:* Size, no. of accesses, details of most recent access (time, type, initiator), etc.
- *Task Processing:* No. of completed tasks, consumed CPU-time, status, up-time, no. of basic operations, etc.
- *Current Task:* Elapsed processing time, no. of child tasks, occupied memory, etc.
- *Task Decomposition:* No. of decompositions (tasks created), no. of failed decompositions, consumed CPU-time, etc.
- *Overall Task Pool:* No. of active workers, arrival rate of tasks, up-time, etc.

As part of the prototypical runtime system that leverages our EDL (cf. Section 5), we provide a set of monitors that are implicitly accessible without a prior declaration.

Constraints define the conditions that the system must fulfill and trigger the execution of strategies

```
constraints:
 - name: <string>
   properties:
    value_L: <number|monitor name>
    #left hand value of
     ↪  comparison
    value_R: <number|monitor name>
    #right hand value of
     ↪  comparison
    comparison_operator:
    <le|ge|eq|lt|gt|ne>
    #operator to compare
     ↪  right/left
    time_constraint: <number>
    #constraint validity in sec
```

Listing 2: Constraints section within an elasticity policy.

when they are not fulfilled. The structure of a constraint is shown in Listing 2. Based on the three properties (`value_L`, `value_R`, `comparison_operator`), conditional expressions can be specified. An expression defines threshold values for a particular aspect of the parallel system within it is supposed to operate. Furthermore, the constraints reflect individual trade-offs that exist between several conflicting optimization goals at runtime, obliging the user to define a suitable solution. For parallel computations in cloud environments, for example, these conflicting goals are usually fast processing and low monetary costs. The properties `value_L` and `value_R` can be assigned with constant numeric values or real-time data retrieved from monitors. Finally, the `time_constraint` property defines the operation time in seconds, i.e., the period for which the constraint is active and gets evaluated.

Strategies are used to specify the steps that need to be taken if a constraint gets violated. Listing 3 shows the basic structure of a strategy. The `condition` property defines the constraint whose violation triggers the execution of the strategy's actions. The employed set of actions is in accordance with the previously discussed levels of parallelism. Hence, there exist actions for controlling the physical parallelism (e.g., adapting the number of processors) and for controlling the logical parallelism (e.g., adapting the fork policy for task decomposition).

4 USE-CASES

In this section, we present two exemplary use-cases to substantiate the capabilities of our proposed elasticity mechanisms.

```
strategies:
 - name: <string>
   properties:
    condition: <constraint>
    #Triggering constraint
    actions: <list of actions>
    #Actions carried out
```

Listing 3: Strategies section within an elasticity policy.

In general, parallel applications that rely on dynamic problem decomposition (e.g. Boolean satisfiability or discrete optimization) share a set of common runtime characteristics. For example, a reasonable assumption is that the ongoing decomposition during computation continuously reduces the granularity of newly generated tasks. The task granularity is defined by the amount of essential computations in ratio to the amount of communication overhead. As parallel computation progresses, the overhead for decomposition and load balancing increases, which in turn continually reduces parallel efficiency. Ultimately, decomposition of tasks reaches a granularity where the overhead outweighs the gains achieved by parallel execution. Particularly at the end of the computation, when remaining work becomes scarce and processors begin to compete, decomposition generates fine-grained tasks, and parallel execution becomes inefficient. Since computing resources can no longer be utilized efficiently, users would pay for parallel overhead rather than for essential computations.

Whether this is the case can be assessed by monitoring the mean time required for the execution of individual tasks. Falling below a certain threshold, efficiency can be optimized by decreasing the number of processors. In this way, the degree of physical parallelism is adapted to the degree of logical parallelism. The specification of the threshold enables users to opt for a trade-off between cost and speedup. Listing 4 shows the elasticity policy containing the specification for this use-case.

The policy can be created with minimal effort, as it requires only a few lines of code. In particular, we can formulate the previously stated elasticity requirements by employing only two elements: One constraint and one strategy. Constraint *c_granularity* (cf. Line 2) defines the granularity limit of generated tasks that must be maintained during execution. The first three properties of the constraint are constituents of a logical expression, specifying that the mean execution time of tasks must be at least 1000 milliseconds. The current mean task execution time is obtained by the monitor *m_taskProcessingTime*. Finally, the last property specifies that the evaluation of the constraint

```
1  constraints:
2   - name: c_granularity
3     properties:
4      value_L:  m_taskProcessingTime
5      value_R: 1000
6      comparison_operator: ge
7      time_constraint: infinity
8
9  strategies:
10  - name: s_scaleInPhysicalParallelism
11    properties:
12     conditions: c_granularity
13     actions:
14      physicalParallelism: 0.75
```

Listing 4: Elasticity specification of first use-case.

```
1  constraints:
2   - name: c_lowIdleOverhead
3     properties:
4      value_L:  m_idle
5      value_R: 0.25
6      comparison_operator: lt
7      time_constraint: infinity
8
9  strategies:
10  - name: s_scaleOutLogicalParallelism
11    properties:
12     conditions: c_lowIdleOverhead
13     actions:
14      logicalParallelism: 1.25
```

Listing 5: Elasticity specification of second use-case.

is not limited to a given point in time or period. Complementary to the constraint *c_granularity*, strategy *s_scaleInPhysicalParallelism* (cf. Line 10) is responsible for its enforcement. According to its properties, violations of *c_granularity* lead to a decrease in physical parallelism, i.e., fewer processors will be utilized during parallel execution. Each violation triggers a scale-in action, reducing the number of processors to 75% of the currently utilized number.

In the following use case we will deal with another source of parallel overhead in the context of dynamic problem decomposition, namely processor idling. Minimizing processor idling is of particular importance in cloud environments due to the pay-per-use billing model. Since compute resources are billed per time unit, without regard to their actual utilization, idle resources can be an excessive cost driver. In parallel applications, a major source of idling overhead is the lack of logical parallelism, i.e., the number of available tasks is too low. However, the EDL can be employed to control the problem decomposition process at runtime and thus the number of task instantiations. By monitoring the idling time of processing units in the task pool, one can assess whether there is a lack of logical parallelism. If the idle time falls below a certain threshold, the ratio of called and performed fork statements is adjusted so that the calls result more often in task instantiations. Listing 5 shows the elasticity policy for this use-case.

5 EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of the proposed elasticity description language. First, the methodology and setup of the experimental platform are described in Section 5.1. In Section 5.2, we then report on a detailed analysis of the correlation between the elasticity requirements specified in a policy and the runtime behavior of a parallel computation.

5.1 Methodology and Platform

To put our work into practice, we have implemented a prototype of a runtime system that manages the elasticity of a parallel system based on EDL policies. This prototype employs the distributed task pool execution model, with each worker being hosted on a separate virtual machine equipped with 1 vCPU and 2 GB RAM. The experiments were conducted on our private cloud running OpenStack during regular multi-tenant operation. The hardware underlying this cloud consists of a cluster of identically configured servers, each equipped with two Intel Xeon E5-2650v2 CPUs and 128 GB RAM.

To demonstrate broad applicability and improve the reliability of our results, we require an application whose parallel execution shows differing runtime characteristics depending on the input data. This particularly holds for the class of state space search problems, which we employ for evaluation purposes. Algorithms that deal with this class of problems implicitly construct a search tree in which each node represents a state, edges constrain state properties, and paths represent results. As the computation proceeds, these algorithms explore the state space tree by expanding tree nodes, aiming to find profitable paths. For parallel search algorithms, where the exploration of individual subtrees takes place simultaneously, the different and unpredictable sizes of the subtrees can lead to an imbalance of the load, which in turn significantly affects scalability. Due to this dynamic process, the shape and size of the tree highly depend on the input data and cannot be estimated in advance by a (semi-)static analysis. Since this behavior corresponds to our requirements for the runtime characteristics, state space search problems are well suited for our evaluation. However, standard implementations of representative state space search problems, such as Boolean satisfiability or discrete optimization exhibit randomization, resulting in highly varying parallel runtimes for the same input. Furthermore, we cannot directly control the runtime characteristics, which renders a systematic investigation of elasticity control mechanisms unfeasible. Therefore, we employ for evaluation purposes the Generic State Space Search Application (GSSSA) presented in (Haussmann et al., 2018). This application exhibits all relevant characteristics of state space search applications, particularly w.r.t. parallel execution, and allows to explicitly control runtime characteristics through a small set of parameters. The total workload is represented by random SHA-1 hash calculations and defined by the parameters w_r and w_i, each representing a fraction of the search tree of different structure. The parameter w_r defines the regular fraction where expansions lead to two subtrees of equal workload. On the other hand, w_i defines the irregular fraction where the balancing parameter b specifies the partitioning of the workload between the expanded subtrees. Finally, the parameter g specifies the smallest permissible workload of expansions. For our evaluation we applied the parameters w_r = 2,500,000,000, w_i = 7,500,000,000, $b = 0.001$, and $g = 1$.

5.2 Experimental Analysis of Use-case

In the following, we investigate whether the elasticity requirements specified in a policy are adequately fulfilled during execution. There are several aspects that determine the efficiency of our approach, such as the responsiveness of corrective elasticity actions taken and their impact on the runtime behaviour. In this evaluation we will employ the elasticity policy of the first use-case, presented in Section 4 (cf. Listing 4). In particular, we analyze the parallel computations for four different parameterizations of this policy, which are shown in Table 1. In the following we refer to the parameterizations by their IDs (#1-#4). Additionally, we will examine the same parallel computation utilizing a constant number of processors. This reference values enables us to draw a comparison between computations with and without employing elasticity policies. We performed each computation on a cluster with a capacity of 32 virtual machines.

In the following, we evaluate the performance and cost aspects of parallel computations employing elasticity policies. Of particular interest for our evaluation are the parallel runtime T_{par}, speedup S, efficiency E, and the costs stemming from the total consumption of processing time C. Due to the pay-per-use billing model of the cloud, the consumed processing time directly translates into monetary costs (thus we can assume w.l.o.g. costs of 1\$ per processing time unit). Without using an elasticity policy, computation generally would be performed utilizing a constant number of processors that is selected by the user. As comparative data for our evaluation, we employ two scenarios: A sequential computation (i.e. number of processors $p = 1$) and a parallel computation utilizing a number of processors $p = 32$, which will be denoted as #*seq* and #*par*, respectively. This, in turn, allows to determine the overhead that results from the performed elasticity operations. All computations were carried out five times and the results shown in Figure 2 are based on the arithmetic mean.

Table 1: Policy parameterizations used for evaluation.

	Parameterization			
	#1	#2	#3	#4
constraints				
name	c_granularity			
properties				
value_L [ms]	m_taskTime			
value_R [ms]	500	100	50	10
operator	gt			
strategies				
name	s_scaleInPhysicalParallelism			
properties				
conditions	not: c_granularity			
actions	physicalParallelism: 0.75			

The sequential computation #seq has a runtime of T_{seq} = 5,163 sec, and thus costs of C = 5,163 \$. On the other hand, the parallel computation #par that utilizes a constant number of processors $p = 32$, has a parallel runtime of T_{par} = 295 sec, resulting in costs of C = 9,424 \$, a speedup of $S = 17.53$ and an efficiency of $E = 0.55$. Concerning the parallel runtime T_{par} of computations that employ elasticity policies, one can observe that T_{par} decreases for smaller limits of the task granularity. This effect occurs since the number of processors is reduced at earlier stages of the computation when using large granularity limits. Ordered by decreasing granularity limit, the parallel runtime T_{par} is 3,333 sec, 1,205 sec, 620 sec, and 296 sec. The corresponding speedups are 1.55, 4.29, 8.32, 17.44 and the efficiencies are 0.92, 0.87, 0.83, and 0.65 accordingly. Compared to the parallel computation #par that utilizes a constant number of processors $p = 32$, policy #4 has the best cost-speedup trade-off as the runtime is only increased by 0.3%, but the costs are reduced by 15.5%. Depending on

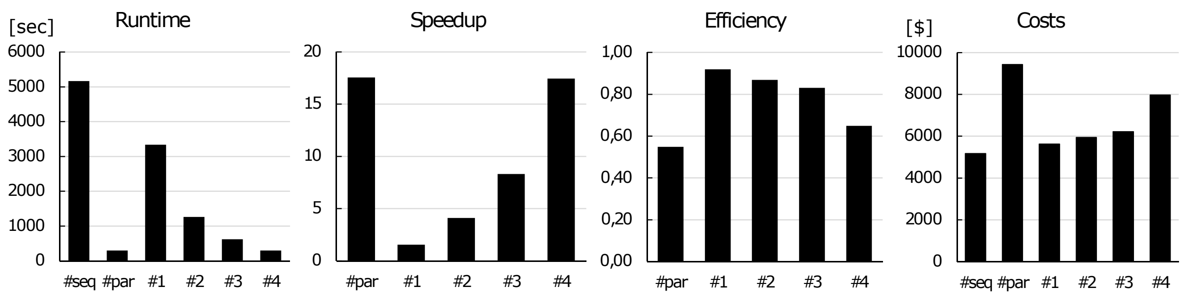

Figure 2: Results of parallel computations with and without employing elasticity policies.

whether speedup or cost-savings are considered to be more relevant, the other policies are preferable. Compared to the most cost-efficient computation, which in this case is the sequential computation #seq, policy #1 decreases the runtime by 35.4%, but increases the costs only by 8.8%.

6 RELATED WORK

As of today, it has been recognized that the utilization of cloud resources is by no means limited to client-server applications but also includes parallel applications. Major cloud providers, like Amazon and Microsoft, already offer several products that are specifically designed for parallel applications (Aljamal et al., 2018). Unlike standard virtual resources, which often possess heterogeneous processing speeds and high-latency network-links (Jackson and Ramakrishnan, 2010), these are performance-optimized virtual resources with a high-speed interconnect such as InfiniBand (Zhang et al., 2017) (Mauch and Kunze, 2013).

From the perspective of cloud consumers, there exists also a growing trend to migrate parallel applications into the cloud and render them cloud-aware (Gupta and Faraboschi, 2016).Recent research has indicated that several classes of parallel applications are well suited for migration, while some others can only be migrated with limitations (Kehrer and Blochinger, 2019c). How this migration takes place depends on the requirements of the respective application. The survey conducted in (Kehrer and Blochinger, 2019a) has found that there exist three prevalent cloud migration strategies of varying complexity in existing research: Copy and paste, cloud-aware refactoring, and cloud-aware refactoring, including elasticity control. For parallel applications, on the one hand, these strategies aim to mitigate the adverse effects of characteristics inherent in cloud environments. For example, heterogeneous capacities of virtual resources such as processors and networks, degrade the performance of parallel applications, especially when synchronous communication is required. On the other hand, more sophisticated strategies aim to enable parallel applications to take advantage of cloud characteristics such as on-demand resource access, elasticity, and pay-per-use billing (Netto et al., 2018), (Rajan and Thain, 2017), (Haussmann et al., 2019).

Elasticity, in particular, is a cloud characteristic that holds great potential for parallel applications. Previous research such as (Galante et al., 2016) and (Kehrer and Blochinger, 2019b) highlighted the need for elasticity support at the application level and emphasized the relevance of novel frameworks for constructing elastic parallel cloud applications. Our work validates these results and provides a better insight into the underlying elasticity mechanisms for parallel cloud applications.

Rule-based control of cloud systems has become an increasingly important topic in the domain of cloud computing. On the one hand, most cloud providers have such offers directly available to consumers on their platforms, such as *OpenStack* or *AWS*, which leverage the formats *HOT* and *CFN*. Meanwhile, an increasing number of studies on this issue indicates a growing interest in research. In (Copil et al., 2015), the authors introduce a language that enables the specification of multi-level elasticity requirements for cloud services. These elasticity requirements are high-level demands, formulated by the user, concerning a cloud service. Furthermore, they proposed a model that considers elasticity as a multidimensional space in which the system can oscillate. An architecture for controlling the behavior of cloud services is presented in (Jennings and Stadler, 2014). By leveraging the introduced abstractions and rules engine, applications can exploit virtual resources from different cloud providers simultaneously to control application behavior. In contrast to our work, their approach focus on elasticity for client-server applications, while elasticity mechanisms mainly act at the infrastructure level. Our work focuses on parallel applications and enables elastic task parallelism in cloud environments, by applying elasticity on both the logical and the physical level of a parallel system.

7 CONCLUSION

In this paper, we introduced an elasticity description language that enables the specification and utilization of elasticity policies at both the cloud infrastructure and application level. Concepts and mechanisms of this language are specifically designed for parallel applications that rely on the fork-join programming model. Our work deals with aspects of elasticity that go beyond the traditional context, which usually considers virtual infrastructures solely. The use cases presented and the evaluation performed have proven the viability of our work and demonstrate that elasticity is an issue beyond the infrastructure level.

REFERENCES

Aljamal, R., El-Mousa, A., and Jubair, F. (2018). A comparative review of high-performance computing major cloud service providers. In *Proc. of the 9th Int. Conf. on Information and Communication Systems*, pages 181–186.

Blochinger, W., Küchlin, W., Ludwig, C., and Weber, A. (1999). An object-oriented platform for distributed high-performance symbolic computation. *Mathematics and Computers in Simulation*, 49(3):161–178.

Blochinger, W., Weber, A., and Küchlin, W. (1998). The Distributed Object-Oriented Threads System DOTS. In *Proc. of the 5th Int. Symp. on Solving Irregularly Structured Problems in Parallel*, pages 206–217.

Blumofe, R. D., Leiserson, C. E., and Joerg, C. F. (1995). Cilk: An Efficient Multithreaded Runtime System. In *Proc. of the 5th ACM SIGPLAN Symposium*, volume 30, pages 207–216.

Copil, G., Moldovan, D., and Dustdar, S. (2015). On Controlling Elasticity of Cloud Applications in CELAR. In *Emerging Research in Cloud Distributed Computing Systems*, pages 222–252.

Galante, G. and Bona, L. C. (2014). Supporting elasticity in OpenMP applications. In *Proc. of the 22nd Int. Conf. on Parallel, Distributed, and Network-Based Processing*, pages 188–195.

Galante, G., Erpen De Bona, L. C., Mury, A. R., Schulze, B., and da Rosa Righi, R. (2016). An Analysis of Public Clouds Elasticity in the Execution of Scientific Applications: a Survey. *Journal of Grid Computing*, 14(2):193–216.

Gupta, A. and Faraboschi, P. (2016). Evaluating and Improving the Performance and Scheduling of HPC Applications in Cloud. *IEEE Transactions on Cloud Computing*, 4(3):307–321.

Haussmann, J., Blochinger, W., and Kuechlin, W. (2018). Cost-efficient Parallel Processing of Irregularly Structured Problems in Cloud Computing Environments. *Cluster Computing*, 22(3):887–909.

Haussmann, J., Blochinger, W., and Kuechlin, W. (2019). Cost-optimized Parallel Computations using Volatile Cloud Resources. In *Proc. of the 16th Int. Conf. on the Economics of Grids, Clouds, Systems, and Services*, pages 45–53.

Jackson, K. and Ramakrishnan, L. (2010). Performance Analysis of High Performance Computing Applications on the AWS Cloud. In *Proc. of the 2nd Int. Conf. on Cloud Computing Technology and Science*, pages 159–168.

Jennings, B. and Stadler, R. (2014). Resource Management in Clouds. *Journal of Network and Systems Management*, 23(3):567–619.

Kehrer, S. and Blochinger, W. (2019a). A Survey on Cloud Migration Strategies for High Performance Computing. In *Proc. of the 13th Advanced Summer School on Service-Oriented Computing*. IBM Research Division.

Kehrer, S. and Blochinger, W. (2019b). Elastic Parallel Systems for High Performance Cloud Computing: State-of-the-Art and Future Directions. *Parallel Processing Letters*, 29(2).

Kehrer, S. and Blochinger, W. (2019c). Migrating parallel applications to the cloud: assessing cloud readiness based on parallel design decisions. *Software-Intensive Cyber-Physical Systems*, 34(2-3):73–84.

Mauch, V. and Kunze, M. (2013). High performance cloud computing. *Future Generation Computer Systems*, 29(6):1408–1416.

Netto, M. A., Calheiros, R. N., Rodrigues, E. R., Cunha, R. L., and Buyya, R. (2018). HPC cloud for scientific and business applications. *ACM Computing Surveys*, 51(1).

Rajan, D. and Thain, D. (2017). Designing Self-Tuning Split-Map-Merge Applications for High Cost-Efficiency in the Cloud. *IEEE Transactions on Cloud Computing*, 5(2):303–316.

Skillicorn, D. B. and Talia, D. (1998). Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169.

Zhang, J., Lu, X., and Panda, D. K. (2017). Designing locality and NUMA aware MPI runtime for nested virtualization based HPC cloud with SR-IOV enabled InfiniBand. In *Proc. of the 13th ACM SIGPLAN/SIGOPS*, pages 187–200.