The Journal of Supercomputing (2020) 76:9211–9245
https://doi.org/10.1007/s11227-020-03197-y

Equilibrium: an elasticity controller for parallel tree search in the cloud

Stefan Kehrer[1] · Wolfgang Blochinger[1]

© The Author(s) 2020, corrected publication 2021

Abstract

Elasticity is considered to be the most beneficial characteristic of cloud environments, which distinguishes the cloud from clusters and grids. Whereas elasticity has become mainstream for web-based, interactive applications, it is still a major research challenge how to leverage elasticity for applications from the high-performance computing (HPC) domain, which heavily rely on efficient parallel processing techniques. In this work, we specifically address the challenges of elasticity for parallel tree search applications. Well-known meta-algorithms based on this parallel processing technique include branch-and-bound and backtracking search. We show that their characteristics render static resource provisioning inappropriate and the capability of elastic scaling desirable. Moreover, we discuss how to construct an elasticity controller that reasons about the scaling behavior of a parallel system at runtime and dynamically adapts the number of processing units according to user-defined cost and efficiency thresholds. We evaluate a prototypical elasticity controller based on our findings by employing several benchmarks for parallel tree search and discuss the applicability of the proposed approach. Our experimental results show that, by means of elastic scaling, the performance can be controlled according to user-defined thresholds, which cannot be achieved with static resource provisioning.

Keywords Cloud computing · High-performance computing · Task Parallelism · Elasticity of parallel computations

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

Wolfgang Blochinger
wolfgang.blochinger@reutlingen-university.de

[1] Parallel and Distributed Computing Group, Reutlingen University, Reutlingen, Germany

Published online: 20 February 2020

1 Introduction

The cloud evolved into an attractive execution environment for high-performance computing (HPC) with benefits such as on-demand access to compute resources, pay-per-use, and elasticity [20, 23, 55, 59, 80, 81]. Former performance issues inherent to standard cloud environments such as heterogeneous processing speeds as well as low network throughput and high network latency (resulting from virtualization and resource pooling) have been addressed by novel concepts to make cloud environments HPC-aware [23, 32, 53, 84]. As of today, many cloud providers, including Amazon Web Services (AWS)[1] and Microsoft Azure,[2] offer HPC-aware cloud environments, i.e., cloud environments optimized for HPC [2, 85]. Recently, a single cloud-based parallel system built on top of AWS has been listed in the TOP500 list[3] at rank 136.

Whereas HPC-aware cloud environments enable the migration of existing parallel applications without modifications [45], elasticity, which is often considered to be the most beneficial cloud-specific property [1, 22], gives rise to a fundamentally new concept: The ability to control the number of processing units employed by an application at runtime. However, as of today, it is still an open research question, which parallel applications can benefit from elasticity. Further, it is not well understood according to which principles elasticity control mechanisms should be built, which also largely depends on the class of parallel applications considered. Existing research in this field mainly targets trivial parallel applications with simple communication and coordination patterns [61, 66, 69, 70].

In this work, we discuss the opportunities and challenges related to elasticity for parallel tree search applications. Well-known meta-algorithms based on this parallel processing technique include branch-and-bound and backtracking search. Their computation and communication patterns are input-dependent, unstructured, and evolving during the computation and thus their scaling behavior cannot be determined upfront [24, 77]. Traditionally, these applications have been operated in on-site compute clusters, where a fixed number of processing units has to be defined upon job submission [51]. However, selecting the number of processing units upfront is a difficult task for parallel tree search applications: As their scaling behavior is unknown and hard to predict, the performance in terms of efficiency is only known after the execution has been completed. Selecting too many processing units leads to tremendous waste of money and energy and might also prevent other jobs from being executed while only little improvement in speedup is gained. On the other hand, selecting too few processing units leads to very long execution time or the application might even run into a time limit enforced by the environment [59], which results in termination of the application.

In stark contrast to on-site compute clusters, elasticity in cloud environments allows to explicitly control efficiency and the monetary costs of a computation by

[1] https://aws.amazon.com.

[2] https://azure.microsoft.com.

[3] TOP500 list (June 2019): https://www.top500.org/lists/2019/06.

adapting the number of processing units at runtime according to measured runtime metrics. We discuss this novel ability in detail and introduce an elasticity controller that dynamically adapts the number of processing units according to user-defined cost and efficiency thresholds. Our contributions are as follows:

- We describe the opportunities and challenges related to elasticity for parallel tree search applications based on an in-depth analysis of their inherent characteristics.
- We discuss how to construct an elasticity controller for parallel tree search applications that dynamically adapts the number of processing units according to user-defined cost and efficiency thresholds.
- We provide an extensive evaluation of a prototypical elastic parallel system architecture based on the presented concepts and report on experiments in an OpenStack-based private cloud environment.

Our work is structured as follows. In Sect. 2, we describe the challenges related to parallel processing in the cloud, the characteristics of parallel tree search applications as well as the predominantly used parallel execution model, elasticity in cloud environments, and elastic parallel systems. In Sect. 3, we formulate the problem statement addressed in this work. In Sect. 4, we discuss how parallel tree search applications can benefit from elasticity as well as the related challenges. Based on our findings, we present *Equilibrium*—an elasticity controller that enables efficiency-based and cost-based elasticity control for parallel tree search applications in Sect. 5. In Sect. 6, we describe the system architecture, which is employed in Sect. 7 to evaluate our approach. Moreover, we discuss the applicability of the presented concepts in Sect. 8. Related work is analyzed in Sect. 9. Finally, Sect. 10 concludes this work.

2 Fundamentals

First, we discuss the specifics of parallel processing in cloud environments. Subsequently, we describe the characteristics of parallel tree search applications, i.e., applications that employ parallel tree search as the underlying parallel processing technique, as well as the related task pool execution model. Then, we summarize different approaches to elasticity control. Finally, we describe the characteristics of elastic parallel systems.

2.1 Parallel processing in the cloud

Since hitting the power wall, parallel processing has been considered the ultimate tool to speed up the computation of ever growing problems in science and industry [8]. To solve those problems in parallel, traditionally, large clusters of compute nodes, operated in on-site data centers, have been employed. However, the consumption of compute resources changes drastically with the emerging cloud

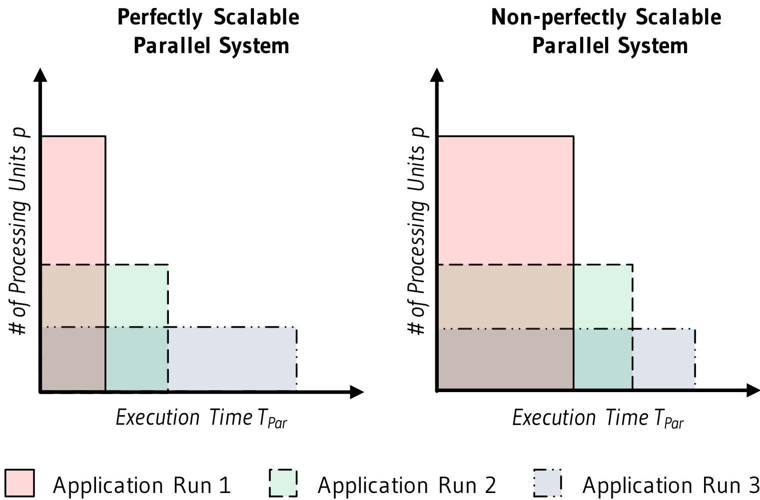

Fig. 1 Whereas the monetary costs of a perfectly scalable parallel system (left) are independent of the number of processing units employed, for parallel systems that are not perfectly scalable (right), the monetary costs increase with the number of processing units [42]. The monetary costs of the computation are expressed as the sizes of the areas shown. Note that the sizes of the areas shown for the non-perfectly scalable parallel system increase with an increasing number of processing units

computing paradigm: The cloud provides metered resources on-demand, which have to be paid on a per-use basis. Consequently, the monetary costs of computations have to be explicitly considered per application run.

Figure 1 compares three different application runs with different numbers of processing units for an (ideal) perfectly scalable and a (realistic) non-perfectly scalable parallel system. The areas shown for each application run 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 system increase with an increasing number of processing units. This can be explained by the overhead that increases with the number of processing units [42]. In cloud environments, because one pays processing units per time unit, both using more processing units and using processing units for a longer period of time increase the monetary costs of a parallel computation. How much overhead occurs for which number of processing units depends on the specific scaling behavior of the parallel system considered.

As a result, whereas elasticity enables users to control the number of processing units at runtime by means of an elasticity controller, a *cost/efficiency-time trade-off* has to be considered for all but the ideal (perfectly scalable) case [42]: Whereas adding more processing units effectively reduces the execution time, a higher number of processing units also leads to higher monetary costs due to

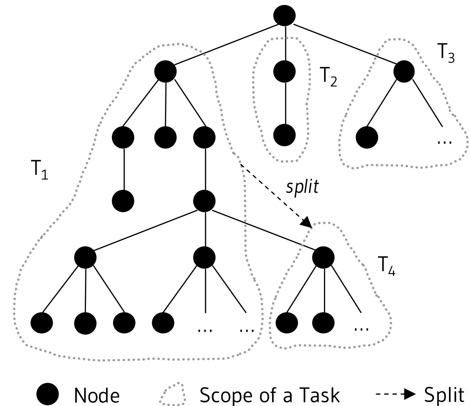

Fig. 2 For parallelization, the search tree is cut into tasks, each capturing a subproblem of the initial problem. Because the search tree is dynamically constructed and its size and shape are not known a priori, new tasks have to be dynamically created by splitting an unexplored subtree from the search tree of an existing task

a lower efficiency. This leads to two conflicting optimization goals: (1) Reduce monetary costs and maximize efficiency versus (2) shorten the execution time.

2.2 Parallel tree search, algorithms and applications

In this work, we specifically investigate parallel tree search applications. Tree search (also called tree traversal) refers to a processing technique for visiting nodes in a tree structure. Many advanced algorithms including branch-and-bound and backtracking search rely on tree search as their underlying processing technique and typically enumerate a large state space that is unknown a priori and unpredictable by nature. Branch-and-bound and backtracking search are typically employed for solving enumeration, decision, and optimization problems—including boolean satisfiability, constraint satisfaction, and graph search problems—with numerous applications in artificial intelligence [39], biochemistry [71], electronic design automation [76], finite geometry [5], model checking [9], automotive product configuration [75], financial portfolio optimization [15], production planning and scheduling [67], as well as fleet and vehicle scheduling [63].

Note that the search tree is not fully materialized in memory but dynamically constructed at runtime as the computation evolves. The structure of the search tree is highly influenced by branching and pruning operations (branch-and-bound) or the backtracking mechanism (backtracking search), respectively. As a result, these applications exhibit a high degree of irregularity.

The most common approach to make use of parallel processing is *exploratory parallelism*. Therefore, different branches (subtrees) of the search tree are explored in parallel by a set of (potentially distributed) processing units. By following a task-parallel approach, subtrees are split from the search tree. Each task represents the traversal of the subtree rooted at a specific node of the tree. This approach is also visualized in Fig. 2. However, statically assigning tasks to processing units leads to a load imbalance because the search tree is dynamically constructed and the size

and shape of each subtree is highly influenced by pruning/backtracking mechanisms, which depend on the input. Consequently, new tasks have to be dynamically created by splitting an unexplored subtree from the search tree of an existing task thus leading to *dynamic task parallelism*. Additionally, distributing the workload evenly across a set of distributed processing units is a challenging task and requires dynamic load balancing to avoid idling processing units. The irregularity introduced by pruning/backtracking mechanisms is highly input problem-specific and thus the problem size (defined by the shape and size of the search tree) and task sizes (defined by the shape and size of a specific subtree) are hard to predict upfront. Consequently, their computation and communication patterns are input-dependent, unstructured, and evolving during the computation [24, 77]. The irregular nature of these applications also constitutes the major source of parallel overhead and dramatically affects their parallel performance and scaling behavior.

2.3 Task pool execution model

To deal with the characteristics of parallel tree search applications, the so-called *task pool model* is typically used to manage tasks. A task pool is a data structure that can be used to store dynamically generated tasks and to fetch these tasks later for processing. Moreover, a task pool can be accessed by the load balancing mechanism to fetch tasks that should be processed by another processing unit.

In general, a task pool might be implemented following a centralized or a distributed approach. A centralized task pool is maintained by a single processing unit and accessible by all other processing units. On the other hand, a distributed implementation leads to multiple task pools, local to each processing unit, forming a distributed task pool. Whereas tasks have to be transferred over the network by following a centralized approach, a distributed task pool enables local accesses and thus minimizes communication overhead.

In this work, we specifically address the *distributed task pool* model that enables processing units to store generated tasks locally. Nevertheless, load balancing is required to avoid processor idling. This can be accomplished by either sending tasks to processing units (work pushing) or by fetching tasks from processing units (work stealing) [83]. As the transfer of tasks leads to additional overhead, we favor work stealing as communication is only required in this case if a processing unit runs idle.

2.4 Elasticity in cloud environments

Elasticity is often considered to be the most important cloud-specific property because it enables applications to react to workload changes by dynamically increasing/decreasing the number of processing units [1, 22]. As of today, elasticity is mainly employed in the context of interactive (multi-tier) applications that process (independent) user requests [33, 36, 54]. The entity that controls the number of processing units, e.g., in form of virtual machines (VM), is called *elasticity controller*.

In the context of interactive applications, it provisions more compute resources as the arrival rate of user requests increases and decommissions them as soon as the arrival rate decreases. Most often, elasticity control is executed in an automated manner and decides on scaling actions either by means of reactive or proactive mechanisms [22]. Whereas reactive elasticity control employs user-defined thresholds on monitored metrics to control the number of processing units, proactive elasticity control uses forecasting to adapt a system according to its predicted behavior in the future. Reactive or proactive mechanisms can also be combined to construct a hybrid approach [10]. Detailed analyses of existing research on elasticity as well as classifications of elasticity mechanisms are presented in [1, 22, 38, 52].

2.5 Elastic parallel systems

In this section, we define the term *elastic parallel system* and describe the fundamental differences to a *parallel system* based on [42].

A parallel system is traditionally defined as a combination of a parallel algorithm (parallel application, programming model/middleware) and a parallel architecture (hardware) [27]. Parallel systems are evaluated as a whole by means of performance metrics such as parallel execution time T_{par}, speedup S, and parallel efficiency E, which are measured with a specific input I and under the assumption of a static number of processing units p.

Elastic parallel systems, on the other hand, cannot be evaluated under the assumption of a static number of processing units. We define an elastic parallel system as a parallel system accompanied by an elasticity controller that adapts the number of processing units at runtime. Consequently, for elastic parallel systems, the number of processing units is a function of time $p(t)$ that is implicitly defined by the elasticity controller.

We define **elastic speedup** $S_{elastic}$ and **elastic efficiency** $E_{elastic}$ analogously to speedup S and parallel efficiency E. However, whereas speedup and parallel efficiency are functions of a fixed number of processing units p, elastic speedup and elastic efficiency are functions of $p(t)$:

$$S_{elastic}(I, p(t)) = \frac{T_{seq}(I)}{T_{par}(I, p(t))} \tag{1}$$

$$E_{elastic}(I, p(t)) = \frac{S_{elastic}(I, p(t))}{\bar{p}} = \frac{T_{seq}(I)}{T_{par}(I, p(t)) \cdot \bar{p}}, \tag{2}$$

where $\bar{p}$ is the time-averaged number of processing units contributing to the computation.

$S_{elastic}$ can be calculated by measuring T_{seq} and T_{par}. $E_{elastic}$ can be calculated according to Eq. 2 based on $S_{elastic}$ and $\bar{p}$. Note that, for a constant function $p(t) = p = \bar{p}$, elastic speedup $S_{elastic}$ and elastic efficiency $E_{elastic}$ are identical to speedup S and parallel efficiency E.

Because elastic parallel systems are typically operated in cloud environments where compute resources have to be paid, also the *monetary costs* have to be considered. We define the monetary costs to operate an elastic parallel system C_{par} as follows:

$$C_{par}(I, p(t)) = T_{par}(I, p(t)) \cdot \bar{p} \cdot c_{\pi}, \tag{3}$$

where c_{π} is the price for one processing unit per time unit. Note that c_{π} is a constant that depends on the cloud offering selected. Technically, each processing unit can be considered as a virtual machine (VM) with one vCPU and application-specific resources (memory, disk, etc.).

3 Problem statement and motivation

Parallel tree search applications and corresponding meta-algorithms such as branch-and-bound and backtracking search are often employed in an industrial setting, where they are used, e.g., to optimize the makespan in production, the composition of a financial portfolio, or the routes in a logistics network. In this context, parallel processing has been used to speed up the computation by many orders of magnitude. At the same time, the money spent for computations is a scarce resource and thus has to be explicitly considered for economic reasons. As a result, the cost/efficiency-time trade-off has to be considered, i.e., one should only pay for more compute resources if the scaling behavior of the corresponding parallel system allows to exploit these resources with a considerable level of efficiency. Otherwise, the monetary costs for additional resources cannot be transformed into an adequate speedup improvement.

However, parallel tree search applications are highly irregular. Thus, their execution time and scaling behavior are hard to predict. This also means that one is not able to predict the parallel performance and monetary costs for solving a specific problem with a specific number of processing units. As a result, statically selecting the number of processing units, which is required to submit a job to a traditional compute cluster, can only be based on guesses, which are prone to produce bad results. Additionally, compute clusters can only be accessed via job schedulers that manage submitted jobs in (most often long) waiting queues and thus do not provide on-demand access to compute resources. This renders static resource provisioning in on-site compute clusters impractical.

In cloud environments, however, parallel tree search applications can benefit from on-demand access to compute resources and elasticity by adapting the number of processing units at runtime. With such an approach, predicting the execution time and scaling behavior is not necessarily required. Instead, one can rely on an elasticity controller that dynamically adapts the number of processing units according to measured runtime metrics. However, as of today, it is not clear how to select appropriate runtime metrics to consider the cost/efficiency-time trade-off inherent to parallel systems (cf. Sect. 2.1). Moreover, there is a lack of concepts

how to construct an elasticity controller, i.e., according to which principles processing units have to be added to or removed from the parallel system at runtime.

In this work, we address this novel opportunity of elastic scaling as well as the related challenges for parallel tree search applications. We discuss how to construct an elasticity controller that dynamically adapts the number of processing units according to user-defined cost and efficiency thresholds.

4 Opportunities and challenges of elasticity control for parallel tree search

In this section, we analyze and discuss how elasticity can be beneficially employed in the context of parallel tree search applications. Because the execution time of these applications is unknown and hard to predict, the cost/efficiency-time trade-off can only be addressed by controlling (1) the efficiency of a parallel computation and (2) the associated monetary costs.

4.1 Efficiency-based elasticity control

Traditionally, a static number of processing units has been employed for parallel computations. As a result, an important question to be answered was “What is the required input size for a given core count such that we maintain a constant, given efficiency?” [74]. However, in the cloud, there is no given core count (number of processing units). Instead, on-demand access to compute resources and elasticity introduce the ability to freely select the number of processing units, which can be even adapted at runtime. Thus, we rather have to answer the question: What is the required number of processing units for a given input size such that we maintain a constant, given efficiency? Note that this question is often far more practical compared to the aforementioned one because one typically employs parallel processing to speed up the computation of a particular (real-world) problem instead of adapting the size of the processed problem according to a given number of processing units. This especially holds in an industrial setting.

However, as discussed in Sect. 2.2, the scaling behavior of parallel tree search applications depends on the processed problem and is hard to predict upfront. Whereas this renders static resource provisioning impractical, we argue that, by means of elastic scaling, one can continuously monitor a parallel system and adapt the number of processing units to meet a user-defined target efficiency. Because the scaling behavior of parallel tree search applications is hard to predict, the design space of elasticity control mechanisms is restricted to reactive approaches that adapt the number of processing units based on the current state of the system.

Unfortunately, elastic efficiency cannot be monitored at runtime because it is, by definition, only known after the parallel computation has been completed. It can thus only be used for ex-post performance evaluation of elastic parallel systems, which requires that the sequential execution time T_{seq} and the parallel execution time

T_{par} are known (cf. Eq. 2). As a result, the fundamental challenge that has to be addressed is to *find an appropriate runtime metric that approximates the elastic efficiency*, which can be employed for elasticity control. With such a runtime metric, an elasticity controller is able to continuously compare the measured value and the user-defined target efficiency, while bringing the measured value close to the target efficiency by adapting the number of processing units. We address the challenge of defining an appropriate runtime metric that approximates the elastic efficiency and constructing a corresponding elasticity controller in Sect. 5.

4.2 Cost-based elasticity control

First, we investigate on the monetary costs of executing a sequential tree search application in a cloud environment. For a sequential application the execution time is unknown a priori due to the algorithmic characteristics. If one processes an input problem described by I with a sequential application in the cloud, the corresponding monetary costs can be defined as:

$$C_{seq}(I) = T_{seq}(I) \cdot c_\pi = W(I) \cdot c_\pi, \tag{4}$$

where T_{seq} is the sequential execution time, W is the size of a problem described by I, and c_π is the price for one processing unit per time unit. Note that the problem size W is defined as the number of (basic) computational steps required to solve a problem with the best sequential algorithm [27]. Under the assumption that it takes unit time to perform a single computational step, the problem size is equivalent to the sequential execution time T_{seq} [27].

Because the monetary costs C_{seq} depend on T_{seq}, the total monetary costs C_{seq} to process a problem described by I sequentially are also unknown a priori and hard to predict. As a result, one cannot reason about the monetary costs to process a specific problem in absolute terms. However, there is a linear correlation between C_{seq} and W: Given a pay-per-use billing model, the monetary costs to process a problem described by I grow linearly with the problem size W. Consequently, whereas one cannot reason about the costs in absolute terms, the costs per problem size remain constant for sequential tree search applications. In the best case, this should also hold for parallel tree search applications. In the following, we discuss how to meet a user-defined target costs per problem size ratio for parallel tree search applications by means of an elasticity controller. Note that this is infeasible with static resource provisioning and requires elastic scaling at runtime.

Parallel tree search applications inherit all the characteristics of sequential tree search. To complicate matters further, parallel processing adds a second challenging dimension: The computation and communication patterns of parallel tree search applications are highly irregular and do not allow any form of prediction with respect to the structure of executions and their scaling behavior. Because the size of a processed problem as well as the corresponding scaling behavior is unknown a priori, we cannot predict the number of processing units required and the resulting monetary costs of the computation.

The fundamental difference between sequential and parallel tree search is that parallel processing leads to overhead in form of idle time, communication, and excess computation. Because one has to explicitly consider the monetary costs of parallel computations in cloud environments and processing units are paid per time unit, in fact, one pays not only for efficiently employed compute resources but also for the overhead that occurs (e.g., in form of idle time and excess computation).

A fundamental problem of constructing a corresponding elasticity controller is that one does not know neither the actual problem size nor the total monetary costs at runtime. To deal with this problem, it is shown that meeting a user-defined target efficiency actually implies a specific costs per problem size ratio.

The monetary costs for parallel processing in the cloud can be described based on Eq. 3. The costs per problem size ratio can be formalized as follows:

$$\frac{C_{par}(I,p(t))}{W(I)} = \frac{C_{par}(I,p(t))}{T_{seq}(I)} = \frac{T_{par}(I,p(t)) \cdot \bar{p} \cdot c_\pi}{T_{seq}(I)} \tag{5}$$

From Eqs. 2 and 5 follows:

$$\frac{C_{par}(I,p(t))}{W(I)} = \frac{c_\pi}{E_{elastic}(I,p(t))} \tag{6}$$

Because c_π is a constant, meeting a user-defined target efficiency $E_{elastic}$ across different problems (application runs) means that the monetary costs for parallel computations increase linearly with the problem size of each problem. For instance, a problem of double the size leads to doubled costs. Note that, under the assumption that we are able to meet a user-defined target efficiency, this also holds for problems with an unknown size. As a result, the monetary costs are bound relative to the problem size even if the actual problem size is not known a priori.

Whereas traditionally the number of processing units p has been considered to be fixed across different problems and application runs, with this approach the costs per problem size can be fixed across different problems and application runs. Also note that the costs per problem size can be configured per application run. Thus, one can also consider different priorities of problems to be solved. For instance, a problem that is more important (has to be solved faster) can be run with a higher costs per problem size ratio.

Technically, a user-defined target cost per problem size ratio can be considered by translating it to the target efficiency required (cf. Eq. 6). Cost-based elasticity control can thus be reduced to efficiency-based elasticity control by meeting the corresponding target efficiency required for the selected cost per problem size ratio.

4.3 Key findings and results

Efficiency-based and cost-based elasticity control basically correspond to two perspectives on the same optimization goal related to the cost/efficiency-time trade-off. Because the execution time of parallel tree search applications is unknown and hard to predict, one can only optimize for faster execution by selecting a lower target

efficiency/higher costs per problem size ratio. Note that compared to traditional approaches this is a huge step forward because one can effectively control the efficiency of parallel computations as well as the associated monetary costs, which have to be explicitly considered in cloud environments. We are thus able to avoid situations in which one spends money for inefficiently used compute resources.

From an energy perspective this also leads to an important benefit: Because cloud users have to consider the monetary costs of their computation, they automatically optimize the number of compute resources employed by only using more resources when it is actually required to gain an improved speedup. As a result, compute resources that spend most of their time in communication and excess computation without gaining any considerable speedups are avoided, which also leads to lower power consumption from a data center operator perspective.

As we have seen, efficiency-based elasticity control with a user-defined target efficiency and cost-based elasticity control with a user-defined target cost per problem size ratio can be considered equivalent. As a result, the key issue of enabling efficiency/cost-based elasticity control for parallel tree search applications is the ability to meet a user-defined target efficiency based on runtime metrics (to evaluate the cost/efficiency-time trade-off) and elastic scaling (to dynamically adapt the number of processing units).

5 Equilibrium: an elasticity controller for parallel tree search

For both efficiency-based and cost-based elasticity control, finding an appropriate runtime metric that approximates the elastic efficiency is required. In this section, we show how to approximate the elastic efficiency at runtime and how to build an elasticity controller based on this knowledge.

The core idea of our elasticity controller is to approximate the elastic efficiency at runtime by monitoring the parallel system and to adapt the number of processing units thus that the measured (approximated) efficiency corresponds to the user-defined target efficiency. If the measured approximated efficiency is higher than the user-defined target efficiency, the elasticity controller is able to provision more processing units. If the measured approximated efficiency is lower than the user-defined target efficiency, the elasticity controller has to decommission existing processing units.

To approximate the elastic efficiency at runtime, we consider the effects of parallel overhead. Parallel overhead is inversely correlated to efficiency, i.e., a higher overhead corresponds to a lower efficiency. The three sources of overhead are idle time, communication, and excess computation. With respect to the task pool execution model (cf. Sect. 2.3), communication overhead is largely related to the transferal of tasks for load balancing purposes (task stealing). Further, load imbalance also leads to idle time (when a processing unit waits for tasks to be received). Excess computation includes all computations that are not performed by the sequential application, e.g., task management.

All three sources of overhead finally affect the percentage of time a processing unit allocates the CPU to do useful work. To enable measurements at runtime, we define the *workload efficiency* in line with [42] as follows.

Definition 1 (*Workload Efficiency*) The workload efficiency *WE* is the percentage of time in which all processing units execute essential (basic) computational steps, i.e., computational steps that are also executed by a corresponding sequential implementation, within a defined time interval.

When the selected time interval is T_{par}, the workload efficiency approximates the elastic efficiency and is called the total workload efficiency WE_{total} in the following. However, to enable elasticity control, shorter time intervals must be selected to monitor the workload efficiency at runtime. Technically, one is able to instrument an implementation of the task pool model to calculate the workload efficiency by comparing the CPU time of worker threads, which execute tasks, to the wall-clock time. The selected time interval to measure the workload efficiency defines the monitoring interval. The implementation of the proposed approach is described in more detail for our prototype in Sect. 6.

Whereas we employ the workload efficiency to approximate the elastic efficiency of parallel tree search applications, existing work considering other application classes proposes the use of the CPU utilization metric based on which scaling actions can be made [66, 68–70]. However, note that CPU utilization does not distinguish between essential and non-essential computations, with the latter stemming from overhead in form of excess computation. As a result, the CPU utilization metric cannot be used to approximate the elastic efficiency in our case.

Based on gathered monitoring data, the elasticity controller decides on how the parallel system should be scaled horizontally, i.e., it adds or removes processing units to/from the computation. In the following, we discuss our scaling strategy.

A well-known problem of elasticity controllers is oscillating effects [11], i.e., compute resources are continuously provisioned and decommissioned leading to high overhead and, as a consequence, to low cost efficiency. We deal with this problem by stopping to add processing units after the target efficiency level has been reached. By doing so, we make use of the fact that the overhead of parallel tree search applications does not decrease as execution time increases. This can be explained as follows: With an increasing execution time (1) more and more subtrees of the search tree have already been evaluated; thus, it is harder to generate large tasks and (2) pruning is typically more effective (e.g., due to better bounds); thus, more subtrees can be pruned and do not have to be evaluated explicitly, which makes the generation of large tasks even harder. Both effects lead to an increasing overhead in form of communication (e.g., transfer of tasks), idle time (e.g., waiting for tasks to be received), and excess computation (e.g., generating and managing more tasks).

Algorithm 1 Scaling Strategy

```
Cloud Management, instance cm.
Group Membership Management, instance gmm.
Timer, instance t.
 1: initCounter ← 3;
 2: lastOp ← NONE; lastOpExecuted ← null;
 3: E_upper ← null; E_lower ← null;                          ▷ threshold values
 4: sg_out ← 5; sg_in ← 1;                                   ▷ scaling granularity [#]
 5: sl_out ← 20; sl_in ← 10;                                 ▷ scaling latency [s]
 6: upon event <Init | E_target, interval> do
 7:     E_upper ← E_target + 1%;
 8:     E_lower ← E_target − 1%;
 9:     trigger <t, Schedule | interval>;
10: upon event <t, Elapsed> do
11:     d ← getTimeDiff(getWallClockTime(), lastOpExecuted);
12:     if lastOp ≠ NONE then
13:         if (lastOp = IN ∧ d < sl_in) ∨ (lastOp = OUT ∧ d < sl_out) then
14:             return;                                      ▷ prevent scaling operations
15:         end if
16:     end if
17:     Scale(gmm.getCurrentNumberOfProcessingUnits());
18: procedure Scale(p)
19:     WE ← [];                                             ▷ workload efficiency measurements
20:     for i = 0 to p − 1 step +1 do
21:         WE[i] ← monitorWE(i);                            ▷ monitoring
22:     end for
23:     WE_avg ← avg(WE);
24:     if WE_avg ≤ E_upper ∧ initCounter > 0 then
25:         initCounter ← initCounter − 1;                   ▷ stop scale-out
26:     end if
27:     if WE_avg > E_upper ∧ initCounter > 0 then
28:         cm.startProcessingUnits(sg_out);                 ▷ scale-out
29:         lastOp ← OUT; initCounter ← 3;
30:         lastOpExecuted ← getWallClockTime();
31:     end if
32:     if WE_avg < E_lower ∧ initCounter = 0 then
33:         cm.stopProcessingUnits(sg_in)                    ▷ scale-in
34:         lastOp ← IN;
35:         lastOpExecuted ← getWallClockTime();
36:     end if
37: end procedure
```

To make use of this knowledge, our scaling strategy can be described as follows. The corresponding algorithm is shown in Algorithm 1. The notation is partially based on the asynchronous event-based composition model introduced in [18]. We start the computation with one processing unit.[4] At runtime, the elasticity controller scales out by adding a configurable number of processing units, which we call the

[4] Note that the scaling strategy can also be adapted to start with a user-defined number of processing units. However, employing more processing units at the beginning of the computation can lead to additional overhead for applications with an unknown scaling behavior, e.g., if more processing units are initially provisioned than actually required.

scale-out granularity sg_{out}, to the computation. Each scale-out operation is followed by a configurable threshold that depends on the time required to provision the processing units, which we call the *scale-out latency* sl_{out}. Thereafter, the workload efficiency is continuously measured and more scale-out operations might be triggered. At some point in time, we reach the target efficiency level, i.e., the number of processing units at which the measured workload efficiency is equal to (or lower than) the user-defined target efficiency. At this point, the initialization phase is completed. This means that after the target efficiency level is reached, no more processing units are added for this application run. Technically, we consider the initialization phase as completed when the target efficiency level is reached or fallen below in three consequential monitoring intervals. After the completion of the initialization phase has been detected, we scale in (by decommissioning processing units) as follows: Whenever the workload efficiency falls below the target efficiency level, a configurable number of processing units, which we call the *scale-in granularity* sg_{in}, is removed from the computation. Each scale-in operation is followed by a configurable threshold that depends on the time required to decommission the processing units, which we call the *scale-in latency* sl_{in}. Technically, we work with two thresholds: An upper threshold E_{upper} that has to be exceeded to trigger scale-out operations and a lower threshold E_{lower} that has to be fallen below to trigger scale-in operations. The elasticity controller automatically creates these thresholds from the user-defined target efficiency E_{target}, which is required as input.

6 Elastic parallel system architecture

In this section, we describe the elastic parallel system architecture used to evaluate the proposed elasticity controller (cf. Fig. 3). The system architecture includes (1) an OpenStack-based private cloud at the infrastructure layer, (2) a cloud-aware runtime system based on the distributed task pool execution model, on top of which elastic parallel applications can be built, and (3) an implemented prototype of our elasticity controller. In the following, we explain all components and their relationships in detail.

OpenStack[5] is a widely used open-source platform for cloud computing that offers Infrastructure-as-a-Service (IaaS) to customers. OpenStack provides a unified software layer on top of potentially diverse hardware resources such as processing, storage, and networking resources. On-demand self-service is enabled by a web-based user interface, command-line tools, and RESTful web services. Note that because we only require the capability to provision and decommission processing units (in form of VMs) on demand, also any other cloud environment could be used.

Parallel applications have to be constructed according to cloud-specific design principles to benefit from elasticity [43, 61]. To develop and operate parallel tree search applications, we employ TASKWORK [46]—a Java-based cloud-aware runtime system that enhances the distributed task pool execution model to support

[5] https://www.openstack.org.

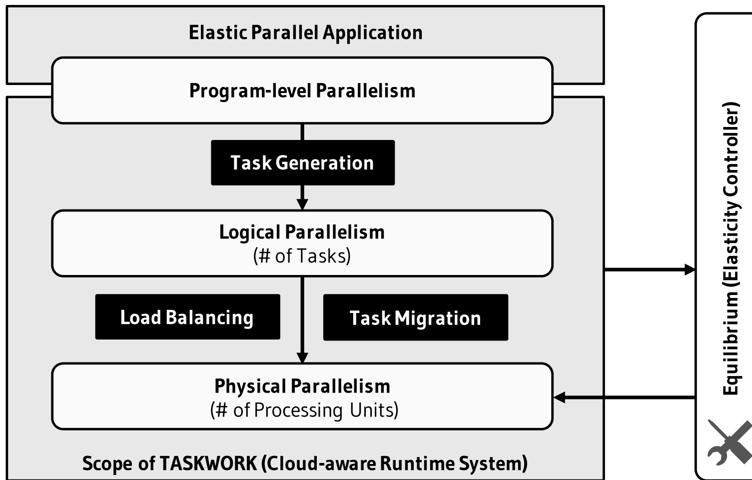

Fig. 3 The elasticity controller monitors the parallel system and adapts the physical parallelism at runtime. The runtime system transparently adapts the logical parallelism by generating tasks whenever required, handles load balancing and task migration to map the logical parallelism to the physical parallelism, and thus provides an elastically scalable parallel system

elastic scaling. It also provides a development framework to implement elastic parallel applications. Based on the framework, application developers only mark potential parallelism in their programs while TASKWORK automatically manages the dynamic adaptation. Adding new processing units to a parallel computation means that the physical parallelism changes at runtime. To effectively exploit the available physical parallelism, the degree of logical parallelism of the application has to fit the physical parallelism given by the number of processing units to achieve maximum efficiency. In this context, the degree of logical parallelism can be defined as the number of tasks. Consequently, two things are required to exploit newly added processing units: (1) the generation of new tasks and (2) load balancing (to transfer these tasks to the new processing units). As described in Sect. 2.3, the distributed task pool model supports dynamic load balancing (e.g., in form of task stealing) by design. On the other hand, the physical parallelism can also be adapted by removing processing units. In this case, task migration, i.e., the transferal of tasks to other processing units, is required to release processing units that have been selected for decommissioning. In summary, the runtime system automatically adapts the logical parallelism by generating tasks whenever required (dynamic task generation), handles load balancing and task migration to map the logical parallelism to the physical parallelism, and thus provides an elastically scalable parallel system. The applications employed for our evaluation, which have been implemented on top of the described runtime system, are presented in Sect. 7.

The elasticity controller monitors the parallel system and adapts the number of processing units (i.e., the physical parallelism) according to the principles discussed in Sect. 5. Technically, we implemented the monitoring of required metrics based on code-level instrumentation. Corresponding mechanisms are provided

by the management interface of the Java Virtual Machine (JVM) thread system in form of the `ThreadMXBean`, which allows the measurement of a thread's CPU time. Because TASKWORK employs one so-called *worker thread* per processing unit, the CPU time of each worker thread is measured to determine the workload efficiency (as defined in Sect. 5). Therefore, the measured CPU time of a worker thread is compared to the wall-clock time in order to calculate the percentage of time in which a processing unit executes essential (basic) computational steps. Collection and aggregation of metrics are performed by distributed monitoring agents and a global aggregator that supplies the elasticity controller with monitoring data. The monitoring interval can be specified according to application-specific requirements. It is worth mentioning that code-level instrumentation has been integrated at the runtime system level and thus has not to be dealt with by application developers, which fosters the usability of our approach. Scaling operations, i.e., provisioning and decommissioning of VMs, are performed via OpenStack's self-service API.

7 Experimental evaluation

To evaluate the proposed elasticity control mechanism, we report on several experiments that consider different parallel tree search applications. First, we describe our evaluation method in detail. Second, we describe the parallel tree search applications employed. Finally, we report on our measurements and discuss the results obtained.

Setup Processing units are operated on CentOS 7 virtual machines (VM) with 1 vCPU clocked at 2.6 GHz, 2 GB RAM, and 40 GB disk. All VMs are deployed in our OpenStack-based cloud environment. The underlying hardware 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.

7.1 Evaluation method

Our goal is to show that the proposed elasticity controller is able to meet a user-defined target efficiency for different input problems, for which the resulting scaling behavior of the parallel system is unknown a priori. Therefore, we measure the performance of several example problems with different degrees of irregularity (leading to a different scaling behavior) and discuss the results obtained. The applications used are described in Sect. 7.2. For each application run, the parallel execution time T_{par}, the time-averaged number of processing units employed $\bar{p}$, and the total workload efficiency WE_{total} are measured. Moreover, $S_{elastic}$ and $E_{elastic}$ are determined. Based on the values, we calculated three important metrics to evaluate the elasticity controller: (1) The percentage error between the determined elastic efficiency $E_{elastic}$ and the target elastic efficiency E_{target}, which quantifies the ability of the elasticity controller to meet the target elastic efficiency:

$$\delta_{overall} = \frac{|E_{elastic} - E_{target}|}{E_{target}} \cdot 100 \tag{7}$$

(2) The percentage error between the measured workload efficiency WE_{total} and the target elastic efficiency E_{target}, which quantifies the quality of the scaling strategy:

$$\delta_{scale} = \frac{|WE_{total} - E_{target}|}{E_{target}} \cdot 100 \tag{8}$$

(3) The percentage error between the measured workload efficiency WE_{total} and the determined elastic efficiency $E_{elastic}$, which quantifies the approximation of the elastic efficiency with the workload efficiency by means of our code instrumentation and monitoring approach:

$$\delta_{approx} = \frac{|WE_{total} - E_{elastic}|}{E_{elastic}} \cdot 100 \tag{9}$$

To evaluate the reliability of our implemented elasticity controller, we also compare different application runs for the same input problem in terms of the scaling actions executed. Finally, we compare the performance of the elastic parallel system with the performance of the same parallel system employing a static number of processing units to assess the overhead related to the dynamic adaptation of processing units.

To deal with the platform-specific provisioning overhead of compute resources, we measure the performance of the elastic parallel system with respect to two scenarios: In the first scenario, VMs are already running and only have to be added to the computation by deploying and starting the runtime system described in Sect. 6. In the second scenario, the VMs have to be started before the runtime system can be deployed. It thus includes the VM provisioning overhead. Note that the first scenario enables the systematic evaluation of the presented elasticity controller independent of platform-specific effects. For instance, different technologies can be used in this context such as VMs or containers, which largely affect the provisioning time.

Finally, we discuss the cost implications and show (based on the experimental results) that meeting a specific target efficiency, as a consequence, leads to the same costs per problem size ratio for all processed problem instances.

7.2 Parallel tree search applications

Evaluating parallel tree search applications is a hard task because for many applications work anomalies occur [26, 49, 50]. This means that the amount of work significantly differs between sequential and parallel processing as well as across parallel application runs. This effect results from a dynamically changing shape of the search tree due to pruning (cf. Sect. 2.2) combined with simultaneous knowledge sharing (such as bounds [25] or lemmas [72]) at runtime. As a result, the expanded search tree differs significantly in its size and shape, which renders a systematic evaluation (by measuring multiple parallel runs) infeasible. To deal with this issue, we employ

Table 1 Unbalanced tree search (UTS) instances

Problem instance	Random seed r	Expected value b	Depth d	Tree size [# of nodes]
UTS_1	19	4	17	67688164184
UTS_2	19	4	18	270751679750
UTS_3	29	5	16	195676745034

two benchmark applications to systematically evaluate our elasticity controller. We describe the employed benchmark applications in the following.

Unbalanced Tree Search (UTS) Unbalanced Tree Search [56] is a commonly employed benchmark to evaluate task pool architectures for parallel tree search (for examples see [5, 17, 58]). It can be used to construct synthetic irregular workloads that do not suffer from work anomalies and thus support a systematic evaluation. UTS allows the construction of workloads with different tree shapes and sizes as well as imbalances by means of a small set of parameters. Each node in the tree is represented by a 20-byte descriptor that is used as random variable. Based on a node's descriptor and the selected tree type, the number of children is determined at runtime. Each child node's descriptor is generated by an SHA-1 hash function using the parent descriptor and a child index as input. Consequently, the generation process is reproducible due to the determinism of the underlying hash function. For our measurements, we employ several instances of the geometric tree type, which mimics iterative deepening depth-first search, a commonly applied technique to deal with intractable search spaces, and has also been extensively used in related work [5, 17, 58]. The 20-byte descriptor of the root node is initialized with a random seed r. The geometric tree type's branching factor follows a geometric distribution with an expected value b. An additional parameter d specifies the maximum depth, beyond which the tree is not expanded further. Table 1 shows the UTS instances employed for our measurements.

Generic State Space Search Application (GSSSA) GSSSA [34] has been introduced to address the problem that one cannot control the degree of irregularity of UTS, which has a direct influence on an application's scaling behavior. To deal with this problem, GSSSA explicitly models the tree search workload as a regular and an irregular fraction, which together define the degree of irregularity. Therefore, the root node has two children: One for the regular workload fraction, which performs w_r random SHA-1 hash calculations, and one for the irregular workload fraction, which performs w_i random SHA-1 hash calculations. Two child nodes are generated by splitting the workload fraction of each parent node. For the regular workload fraction, each child node receives half of the parent's workload fraction. For the irregular fraction, the parent's workload fraction is distributed across the child nodes according to a specific balancing factor b. Finally, a granularity parameter g defines the smallest workload fraction allowed (which cannot be split further), i.e., the number of random SHA-1 hash calculations processed as a single atomic operation. The distribution of regular and irregular workload fractions across all processing units is ensured by means of parallel execution (with randomized task stealing).

Table 2 Generic state space search application (GSSSA) instances

Problem instance	Regular fraction w_r	Irregular fraction w_i	Balancing factor b	Granularity g
$GSSSA_{IW}$	5000000000	15000000000	0.001	1000000
$GSSSA_{IS}$	1000000000	19000000000	0.0002	1000000
$GSSSA_{C}$	0	20000000000	0	1000000

Table 3 Elasticity measurements for unbalanced tree search (UTS) instances without VM provisioning

Target efficiency:	UTS_1 ($r = 19; b = 4; d = 17$)			UTS_2 ($r = 19; b = 4; d = 18$)			UTS_3 ($r = 29; b = 5; d = 16$)		
95.0%	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3
T_{par} (s)	783.06	804.08	818.85	3068.90	3049.49	3024.99	2001.77	1789.80	1766.99
$\bar{p}$ (#)	10.50	10.29	10.00	10.70	10.86	10.83	9.66	10.61	10.78
$S_{elastic}$ (#)	10.09	9.83	9.65	10.08	10.15	10.23	8.81	9.85	9.98
$E_{elastic}$ (%)	96.10	95.46	96.49	94.20	93.44	94.50	91.13	92.84	92.59
WE_{total} (%)	94.52	94.48	94.81	95.40	95.32	95.42	95.14	95.13	94.67
$AVG(E_{elastic})$ (%)	96.02			94.05			92.19		
$AVG(WE_{total})$ (%)	94.60			95.38			94.98		
$\delta_{overall}$ (%)	1.07			1.00			2.96		
δ_{scale} (%)	0.42			0.40			0.02		
δ_{approx} (%)	1.47			1.42			3.03		

Table 2 shows the GSSSA instances taken from [34], which we employed for our measurements.

7.3 Experiments and results

To show that meeting a specific workload efficiency effectively approximates the elastic efficiency, we determine the performance of three application runs for each UTS instance depicted in Table 1. For our experimental evaluation, we configured the elasticity controller with a target efficiency of 95.0%. The results of our measurements (in line with the evaluation method described in Sect. 7.1) are shown in Table 3 and discussed in the following.

As depicted in Table 3, our elasticity controller is able to meet the target elastic efficiency of all three UTS instances by dynamically adapting the number of processing units, with only small percentage errors ($\delta_{overall} < 3\%$ in all cases). Moreover, we can see that the time-averaged number of processing units employed by the elastic parallel system $\bar{p}$ is similar for all three UTS instances. This implies that their scaling behavior is also similar.

To show that the elasticity controller is also able to control the elastic efficiency for input problems that lead to a different scaling behavior of the parallel system,

Table 4 Elasticity measurements for generic state space search application (GSSSA) instances without VM provisioning

Target efficiency	$GSSSA_C$			$GSSSA_{IW}$			$GSSSA_{IS}$		
80.0%	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3
T_{par} (s)	752.85	747.86	772.60	488.29	498.79	472.17	647.47	636.16	649.81
$\bar{p}$ (#)	17.20	17.61	17.31	27.57	25.96	28.08	20.63	21.14	20.54
$S_{elastic}$ (#)	13.51	13.60	13.16	21.28	20.83	22.01	15.18	15.45	15.12
$E_{elastic}$ (%)	78.55	77.22	76.03	77.18	80.23	78.36	73.59	73.07	73.62
WE_{total} (%)	79.24	79.32	79.06	79.84	79.64	79.85	79.67	79.29	79.56
$AVG(E_{elastic})$ (%)	77.27			78.59			73.42		
$AVG(WE_{total})$ (%)	79.20			79.78			79.51		
$\delta_{overall}$ (%)	3.42			1.76			8.22		
δ_{scale} (%)	0.99			0.28			0.62		
δ_{approx} (%)	2.51			1.51			8.28		

we employ the GSSSA instances shown in Table 2. As described in Sect. 7.2, GSSSA allows to explicitly control the degree of irregularity of a problem instance, which has a direct influence on the scaling behavior of the parallel system. We discuss the results of our elasticity measurements in the following.

Table 4 shows the elasticity measurements for the three GSSSA instances shown in Table 2. The structure of the table is identical to the one of Table 3. Note that, due to the limited scaling behavior of the GSSSA instances, the elasticity controller was not able to add more than one processing unit while meeting a target efficiency of 95.0%. To enable the evaluation, we thus configured the elasticity controller with a target efficiency of 80.0% for processing the GSSSA instances. As we can see in Table 4, the elasticity controller adapted the number of processing units according to the target efficiency, with only small percentage errors for $GSSSA_{IW}$ and $GSSSA_C$. For $GSSSA_{IS}$, however, the percentage error between the determined elastic efficiency $E_{elastic}$ and the target elastic efficiency E_{target} is slightly higher.

Also note that the time-averaged number of processing units employed by the elastic parallel system $\bar{p}$ is different for all three GSSSA instances. For $GSSSA_{IW}$, ~ 27 processing units have been employed on average. For $GSSSA_C$, ~ 17 processing units have been employed on average. For $GSSSA_{IS}$, ~ 21 processing units have been employed on average. This implies that their scaling behavior is also different, with the scaling behavior of $GSSSA_{IW}$ being better than the one of $GSSSA_{IS}$ and the scaling behavior of $GSSSA_{IS}$ being better than the one of $GSSSA_C$. To show that this is correct, we analyzed the scaling behavior of all three GSSSA instances by measuring the performance in terms of speedup and efficiency for different settings (with a static number of processing units per setting). The results of these scalability measurements are depicted in Fig. 4.

Reliability of elasticity control To evaluate the reliability of our elasticity controller, we compare three different application runs for the same input problem with respect to the scaling actions executed. For this purpose, we selected the $GSSSA_{IS}$

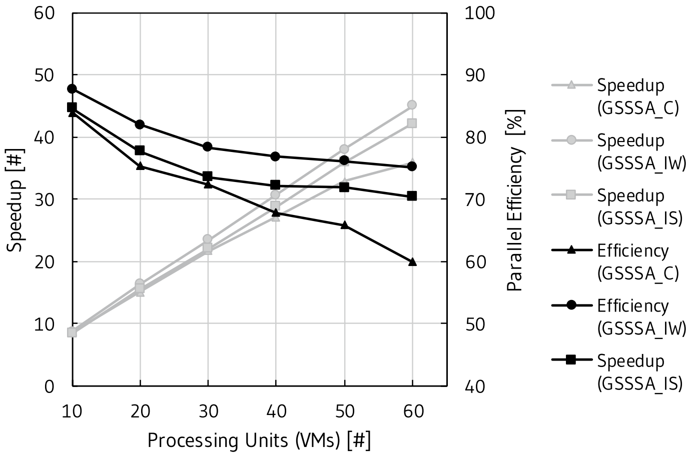

Fig. 4 Scaling behavior of GSSSA instances

instance because it requires the highest number of dynamic adaptations. Figure 5 shows the number of processing units employed as a function of time. For all three application runs, the elasticity controller shows a similar behavior. As shown in the figure, at the end of the computation processing units are decommissioned by the elasticity controller. This is related to overhead in form of load balancing (transferal of tasks) and idle time (when a processing unit waits for tasks), which grows as the execution time increases.

Dynamic adaptation overhead The dynamic adaptation of the number of processing units results in additional runtime overhead, which we assess by comparing the performance measurements of our elasticity experiments for the GSSSA instances (cf. Table 4) with the performance measured for a static setting with a number of processing units close to $\bar{p}$. The results obtained are depicted in Fig. 6 and discussed in the following. Whereas we achieved an elastic efficiency of 78.59% on average

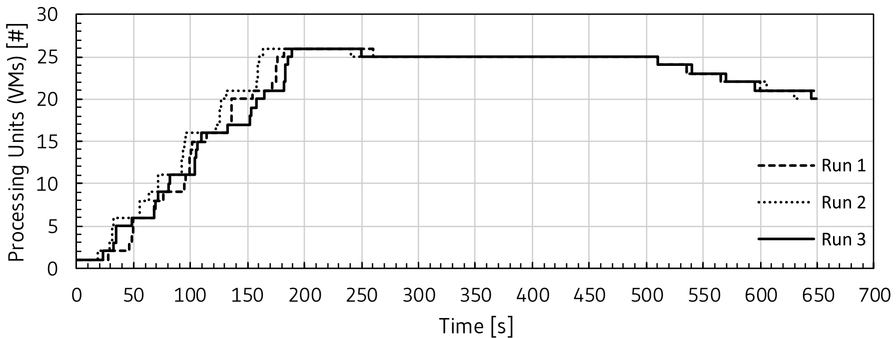

Fig. 5 Comparison of the three application runs with the GSSSA_{IS} problem instance shown in Table 4 with respect to the processing units employed over time. The figure visualizes $p(t)$ for each application run

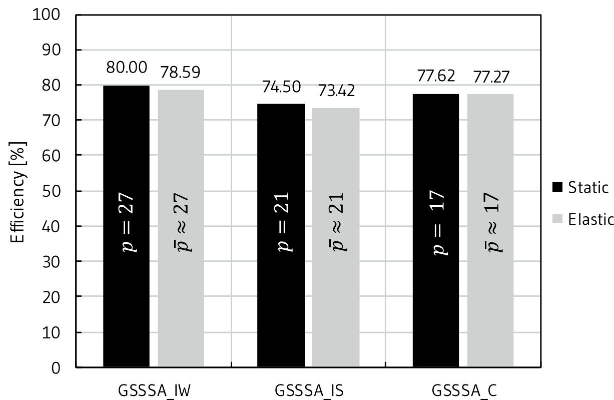

Fig. 6 Comparison of the performance measured in the elasticity experiments (cf. Table 4) and the performance of the same parallel system employing a static number of processing units

for $\mathrm{GSSSA_{IW}}$ with ~ 27 processing units in the elastic setting, we measured a parallel efficiency of 80.00% for a static setting with $p = 27$. For $\mathrm{GSSSA_{IS}}$, we achieved an elastic efficiency of 73.42% on average with ~ 21 processing units in the elastic setting and measured a parallel efficiency of 74.50% for a static setting with $p = 21$. For $\mathrm{GSSSA_{C}}$, we achieved an elastic efficiency of 77.27% on average with ~ 17 processing units in the elastic setting and measured a parallel efficiency of 77.62% for a static setting with $p = 17$. The dynamic adaptation required to control the elastic efficiency thus only imposes minor overhead. This also emphasizes that parallel tree search applications are ideal candidates for cloud adoption because they are less sensitive to dynamic adaptations when compared to data-parallel, tightly coupled applications. Note that controlling the efficiency of parallel tree search applications is infeasible with static resource provisioning. This means that static resource provisioning cannot be employed because we do not know the required number of processing units a priori (cf. Sect. 3).

Elastic scaling with provisioning overhead Up to this point, we neglected the overhead of VM provisioning and decommissioning in our elasticity measurements. For the measurements shown in Tables 3 and 4, we employed running VMs as processing units that are only added to/removed from the computation by starting the runtime system on the respective VM. This approach enables us to evaluate the elasticity controller independently of the cloud environment, i.e., without effects from the underlying infrastructure (such as heterogeneous VM provisioning latencies). In addition, we also executed the elasticity experiments including the VM provisioning overhead to show that our elasticity controller also works in this context. Table 5 shows the corresponding measurements for the three GSSSA instances shown in Table 2. The results obtained are similar to the ones shown in Table 4 and discussed before. Specifically, the scaling strategy seems not to be affected by the VM provisioning overhead. The percentage error between the measured workload efficiency WE_{total} and the determined elastic efficiency $E_{elastic}$ stemming from instrumentation

Table 5 Elasticity measurements for generic state space search application (GSSSA) instances with VM provisioning

Target efficiency	$GSSSA_C$			$GSSSA_{IW}$			$GSSSA_{IS}$		
80.0%	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3
T_{par} (s)	888.58	884.87	831.49	631.97	627.33	624.77	705.17	708.29	677.14
$\bar{p}$ (#)	14.18	15.59	15.98	20.72	21.01	20.80	18.87	18.62	19.86
$S_{elastic}$ (#)	11.44	11.49	12.23	16.44	16.56	16.63	13.94	13.88	14.51
$E_{elastic}$ (%)	80.69	73.72	76.52	79.34	78.82	79.97	73.85	74.52	73.08
WE_{total} (%)	81.61	80.05	80.29	82.32	82.51	83.06	80.65	82.63	80.07
$AVG(E_{elastic})$ (%)	76.97			79.37			73.81		
$AVG(WE_{total})$ (%)	80.65			82.63			81.12		
$\delta_{overall}$ (%)	3.78			0.78			7.73		
δ_{scale} (%)	0.81			3.29			1.40		
δ_{approx} (%)	4.77			4.10			9.89		

and monitoring is only slightly higher. For the sake of brevity, we do not report on the results observed for the UTS instances, which are very similar.

Cost implications As we have shown in Sect. 4, efficiency-based elasticity control with a target efficiency and cost-based elasticity control with a target cost per problem size ratio can be considered equivalent. In the following, we show that meeting a target efficiency, as a consequence, results in the same costs per problem size ratio for all processed problem instances. Therefore, we calculated the costs per problem size for all UTS and GSSSA instances based on Eq. 5 and our measurements. We assume a price for one processing unit per time unit $c_\pi = \$0.000017/s$. This value was taken from Google Compute Engine,[6] where, at the time of writing, a VM with 1 vCPU costs $\$0.0612/h = \$0.000017/s$[7] (region *europe-west3*).

The costs per problem size ratios are shown in Fig. 7 for all UTS instances and Fig. 8 for all GSSSA instances. As we can see, the costs per problem size for all UTS instances are close to each other. The same holds for the GSSSA instances. However, the costs per problem size of the GSSSA instances are on average higher than the costs per problem size of the UTS instances. This results from the different target efficiencies used in the experiments: A lower target efficiency leads to a higher costs per problem size ratio as discussed in Sect. 4. The horizontal line shown in both figures represents the smallest achievable costs per problem size ratio, which is equal to c_π and corresponds to the costs per problem size to process these problems sequentially.

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

[7] Google Compute Engine provides a per-second billing model.

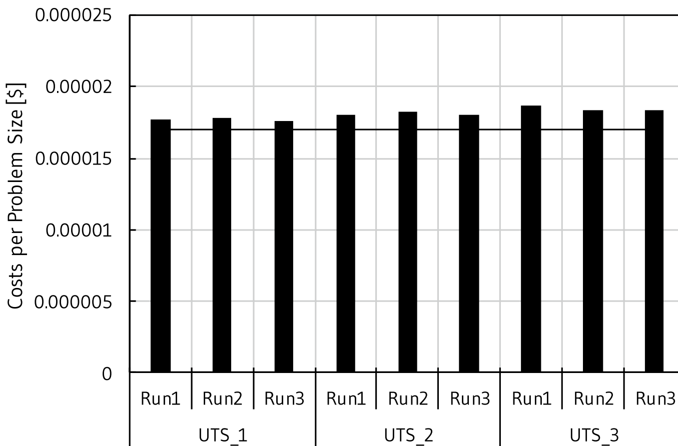

Fig. 7 Comparison of the costs per problem size of all UTS instances shown in Table 1, which are calculated based on Eq. 5 and our measurements

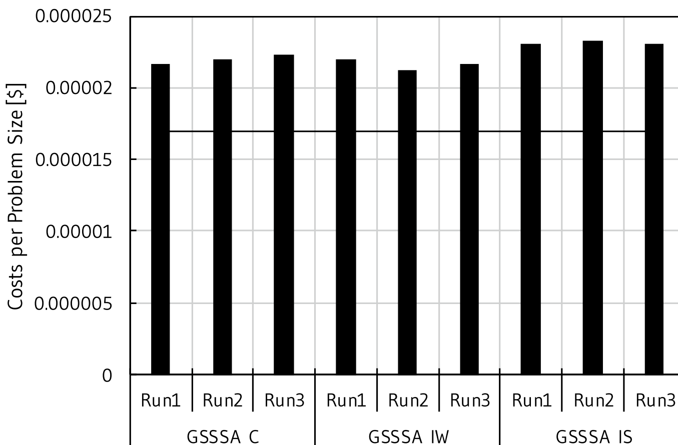

Fig. 8 Comparison of the costs per problem size of all GSSSA instances shown in Table 2, which are calculated based on Eq. 5 and our measurements

8 Discussion

Our elasticity controller basically shows how to handle the fundamental cost/efficiency-time trade-off (cf. Sect. 2.1) by adapting the number of processing units according to user-defined specifications. As we have seen (cf. Sect. 4), typical parallel tree search applications limit the mechanisms that can be employed by an elasticity controller to handle the cost/efficiency-time trade-off due to their algorithmic characteristics. Basically, this results from the use of highly problem-specific heuristics to prune the search space. Because their execution time and scaling behavior are hard to predict, in general, hard limits on the execution time cannot be considered.

Similarly, cost-based elasticity control cannot be based on a fixed monetary budget; one can only control the costs per problem size ratio. Nevertheless, it is possible to enforce both an absolute limit on the execution time and a fixed monetary budget in application scenarios, where parallel tree search applications are used to solve optimization problems, while also using the concepts presented in this work to ensure that compute resources are only employed when they can be exploited with a considerable level of efficiency. This can be accomplished by additionally considering the quality of results.

For enforcing a hard limit on the execution time, one is able to define the specific time limit as a configuration parameter of the elasticity controller, which simply terminates the computation and decommissions the compute resources employed when the given time limit is exceeded. This can be easily implemented by monitoring the wall-clock time and comparing it to the given time limit. When the time limit is exceeded, the currently known best solution of the optimization problem is returned to the user,[8] which is able to proceed with this result. Even though this result might not be the global optimum, one is thus able to proceed without delay thus trading solution quality for shorter execution time. To enforce a fixed monetary budget, a similar strategy can be used. In this case, however, the elasticity controller has to be configured with the monetary budget and terminates the computation when the money already spent for compute resources exceeds the fixed budget.

Trading the quality of results for execution time or monetary costs is specifically valuable in an industrial setting, where users are sensitive to both time and costs depending on the context. However, note that such an approach is only applicable for algorithms that are designed to produce a sequence of (approximate) intermediate results thus that, given a fixed monetary budget or execution time limit, a user is still able to proceed with a usable result after terminating the computation [29]. Parallel tree search applications that are designed to solve an optimization problem continuously refine the result by following a systematic search process and thus allow for these trade-offs. In both cases described above, the concepts presented in this work can be used in addition to these absolute limits to avoid situations, in which money is spent for inefficiently exploited compute resources.

9 Related work

We have identified different fields of related work: (1) Existing approaches to make parallel applications cloud-aware, (2) concepts related to elasticity control for parallel systems, (3) the use of malleability as it has been considered in cluster computing, and (4) related work with respect to parallel tree search applications. Related work that addresses HPC in cloud environments is also discussed in more detail in [42, 45, 55].

[8] With the user being either a human or another system/service.

9.1 Cloud-aware parallel applications

In the past, researchers mainly investigated on how to make cloud environments HPC-aware [53, 85]. However, we can also see a growing interest to make parallel applications cloud-aware [30–32, 43, 61].

The authors of [32] compare the performance of different parallel applications executed in a cloud environment. Several strategies to make parallel applications cloud-aware are presented. One way is to overlap computation and communication by using asynchronous execution models rather than SPMD-based synchronous communication models. Moreover, overdecomposition and controlling the size of tasks/objects can be used to deal with heterogeneous network performance and to avoid idling processing units. Basically, to select the optimal task size, one has to balance various sources of overhead. The runtime system that we employed (cf. Sect. 6) is based on the distributed task pool execution model and implements load balancing with a work stealing approach. It thus overlaps computation and communication by design. Additionally, it controls the number of tasks by generating tasks only when required instead of overdecomposition.

The authors of [31] recognized that heterogeneous processing speeds in cloud environments specifically affect tightly coupled parallel applications. They make use of a dynamic load balancer to deal with load imbalance across vCPUs. In this work, also overdecomposition is used, whereas our runtime system actively controls the logical parallelism of an application to minimize task management overhead. However, it largely depends on the application class considered which approach yields better results.

The Work Queue framework [16] is used in [61] to implement parallel applications for cloud environments. The Work Queue framework is designed for scientific computing and based on a master/worker architecture. The number of workers can be dynamically adapted at runtime to enable elastic scaling. The authors discuss in detail how to convert a parallel application for replica exchange molecular dynamics (REMD) to an elastic application. Several other applications are discussed by the authors in [62].

The authors of [19] discuss opportunities and challenges related to cloud computing with a specific focus on applications from the bioinformatics domain. Moreover, elasticHPC, a software package to ease the use of cloud resources for bioinformatics applications, is presented.

The authors of [48] investigate on elastic parallel processing with serverless computing platforms. They describe a novel approach to parallel cloud programming with so-called *serverless skeletons*, which capture common parallelism patterns and provide abstract implementations of these patterns for serverless computing platforms to ease development effort. A prototypical development and runtime framework is described and evaluated with exemplary parallel applications.

9.2 Elasticity control for parallel systems

In [69], a reactive elasticity controller for iterative-parallel applications is described. The presented approach can be used to transform existing MPMD[9]-based MPI-2 applications with a master/worker architecture into elastic parallel applications. As a result, elasticity can be employed by adapting the number of processing units at the beginning of each iteration based on defined thresholds. Technically, the dynamic adaptation is based on MPI-2, which features dynamic process management [28]. The described elasticity controller uses upper and lower CPU utilization thresholds to decide on the number of processing units required. As discussed in Sect. 5, by employing the CPU utilization metric we cannot distinguish between essential and non-essential computations and thus the cost/efficiency-time trade-off cannot be considered.

The authors of [65] leverage elasticity for MPI-based applications by stopping (along with check pointing) and relaunching the application with a new resource configuration. The authors address MPI-based iterative-parallel applications. The described elasticity controller is designed to optimize the desired execution time, which is estimated based on the number of iterations and the average execution time of an iteration. The underlying assumption is that the amount of work per iteration is constant. Scaling decisions are made by comparing the measured average iteration time with the required iteration time to complete within the user-defined execution time: If the average iteration time is below the required iteration time, processing units are added. Otherwise, processing units are removed. However, this model is not applicable to parallel tree search applications because they cannot be modeled as iterative MPI-based applications. Furthermore, stopping and relaunching the application imposes large overheads.

The authors of [60] address applications based on the so-called *split-map-merge* paradigm, which includes bag-of-tasks, bulk synchronous parallel, and Map-Reduce applications. They present an application model for these applications that enables a model-based prediction of the optimal number of processing units—optimal with respect to the cost-time product, which the authors employ as objective function to evaluate the cost/efficiency-time trade-off. Information on the execution environment (e.g., processing speed, network bandwidth) is obtained by measuring sample workloads. However, model-based prediction cannot be used in the context of parallel tree search applications due to the algorithmic characteristics of these applications.

In [34], an elasticity control mechanism based on minimization of the monetary costs is presented. The authors also address parallel search applications and discuss the two conflicting objectives of fast processing and low monetary costs finally leading to a multi-objective optimization problem and Pareto optimal solutions, which avoids automated decision-making with respect to the number of processing units. To deal with this problem, the authors employ the concept of opportunity costs to convert the underlying objective functions into a single aggregated objective

[9] Multiple Program Multiple Data.

function, thus allowing cost-based elasticity control. The presented cost model is also employed in [35]. Note that one has to adopt the concept of opportunity costs to make use of the presented optimization technique. We present an alternative approach to adapt the number of processing units according to a user-defined target efficiency or costs per problem size ratio.

The authors of [64] discuss the trade-off between the monetary costs of a parallel computation and the quality of the results obtained. Examples of applications for which the quality of results can be traded for the monetary costs of the computation include video and image processing as well as scientific simulations. A measurement-driven analytical modeling approach called CELIA is described, which allows to determine the cost-time optimal cloud configurations given a time deadline and a cost budget. Their approach also employs predictions based on an application model that is parameterized using previous measurement results. Again, note that such an approach cannot be used in the context of parallel tree search applications because one cannot derive information on the resource requirements of a new problem from a previously measured one. The authors state that they focus only on highly parallelizable problems and thus do not model communication overhead. As a result, non-perfectly scalable applications and the cost/efficiency-time trade-off that arises in this context are not considered.

9.3 Malleability in cluster computing

The idea of adapting the number of processing units at runtime is not a fundamentally new concept. Such mechanisms have already been discussed and researched in the field of cluster computing. In this context, *malleability* is considered to be the ability of a scheduled job to deal with a changing number of processing units at runtime [21]. However, this concept differs from elasticity with respect to a fundamental aspect: The instance that controls the number of processing units at runtime. In cluster computing, the job scheduler has the ability to control the number of processing units according to cluster-wide (global) optimization goals. Different job scheduling policies have been proposed in this context [78, 79]. In cloud computing, on the other hand, the cloud customer controls the number of processing units by means of an application-specific elasticity controller. Note that these approaches are fundamentally different in that the cloud provider (infrastructure operator) and the cloud customer (application owner) are two different parties with conflicting optimization goals: The cloud provider optimizes for resource utilization, whereas the cloud customer has to consider efficiency because he also pays for inefficiently exploited compute resources.

The authors of [37] 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 [3] with the goal to maximize the sum of the speedups of all jobs. However, the authors also state that this model largely depends on the quality of speedup estimations. As described earlier, such a model cannot be applied to parallel tree search applications and thus their scaling behavior is hard to

predict. Moreover, 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.

9.4 Parallel tree search

In this work, we specifically address parallel tree search applications. These applications have extensively been considered by related work, discussing their execution with respect to different environments including clusters [6, 7, 12, 13] and grids [4]. In this context, the characteristics of parallel tree search applications have been studied in detail and the task pool model is commonly used as a starting point for environment-specific optimizations. In [57], a skeleton for branch-and-bound applications is presented, which supports parallel execution based on MPI. The authors of [17] present a distributed task pool implementation based on the parallel programming language X10, which follows the Partitioned Global Address Space (PGAS) programming model. The authors of [72] discuss the challenges related to parallel tree search in the context of a distributed parallel satisfiability solver. The authors of [7] discuss the problem of replicable parallel performance of branch-and-bound applications and propose a skeleton that preserves the search order heuristic by distributing work in an ordered manner. COHESION is a microkernel-based platform for desktop grid computing [14, 73] that has been designed for irregularly structured task-parallel problems. It addresses the challenges of desktop grids such as limited connectivity and control. A specific work stealing algorithm that selects victims based on the measured network link latency is presented in [82]. Processing units reachable with a lower network latency are preferred for stealing operations. In this work, we address cloud environments and show how elasticity can be beneficially employed in the context of parallel tree search. More specifically, we present an approach to control the number of processing units at runtime according to user-defined cost and efficiency thresholds.

10 Conclusion and future work

In this work, we discuss the opportunities related to elasticity for parallel tree search applications, which specifically suffer from static resource provisioning leading to huge waste of money and energy because their scaling behavior is hard to predict. We discuss how to monitor the scaling behavior of the corresponding parallel system at runtime and present a reactive elasticity controller that is able to dynamically adapt the number of processing units according to a user-defined target efficiency/costs per problem size ratio. We recognized that a detailed understanding of an application's scaling behavior is a fundamental building block upon which elasticity control mechanisms have to be constructed. Whereas we specifically address the distributed task pool model, our concepts can also be employed for centralized task pool architectures because the major sources of overhead are the same. We plan to evaluate this more thoroughly in the future. We also investigate on container

virtualization for deployment automation [40, 41, 44, 47] and to speed up the provisioning time required for scaling operations. Moreover, we plan to investigate which other application classes can benefit from elasticity control and how to measure the scaling behavior of parallel systems based on other parallel execution models. For instance, reactive elasticity control might also be employed for applications with a known scaling behavior, but where the characteristics of the execution environment change at runtime (e.g., varying network latency and network bandwidth, processing speed). Additionally, other application classes with different characteristics might allow the construction of proactive elasticity control mechanisms thus enabling adaptations based on predictions.

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

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Al-Dhuraibi Y, Paraiso F, Djarallah N, Merle P (2018) Elasticity in cloud computing: state of the art and research challenges. IEEE Trans Serv Comput 11(2):430–447
2. Aljamal R, El-Mousa A, Jubair F (2018) A comparative review of high-performance computing major cloud service providers. In: 2018 9th International Conference on Information and Communication Systems (ICICS), pp 181–186
3. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the 18–20 April 1967, Spring Joint Computer Conference, ACM, New York, NY, USA, AFIPS'67 (Spring), pp 483–485
4. Anstreicher K, Brixius N, Goux JP, Linderoth J (2002) Solving large quadratic assignment problems on computational grids. Math Program 91(3):563–588
5. Archibald B (2018) Algorithmic skeletons for exact combinatorial search at scale. Ph.D. thesis, University of Glasgow
6. Archibald B, Maier P, Stewart R, Trinder P, De Beule J (2017) Towards generic scalable parallel combinatorial search. In: Proceedings of the international workshop on parallel symbolic computation, ACM, New York, NY, USA, PASCO 2017, pp 6:1–6:10
7. Archibald B, Maier P, McCreesh C, Stewart R, Trinder P (2018) Replicable parallel branch and bound search. J Parallel Distrib Comput 113:92–114
8. Asanovic K, Bodik R, Demmel J, Keaveny T, Keutzer K, Kubiatowicz J, Morgan N, Patterson D, Sen K, Wawrzynek J, Wessel D, Yelick K (2009) A view of the parallel computing landscape. Commun ACM 52(10):56–67
9. Barnat J, Brim L, Ceska M, Rockai P (2010) Divine: parallel distributed model checker. In: 2010 ninth international workshop on parallel and distributed methods in verification, and second international workshop on high performance computational systems biology, pp 4–7

10. Bauer A, Herbst N, Spinner S, Ali-Eldin A, Kounev S (2019) Chameleon: a hybrid, proactive auto-scaling mechanism on a level-playing field. IEEE Trans Parallel Distrib Syst 30(4):800–813
11. Bersani MM, Bianculli D, Dustdar S, Gambi A, Ghezzi C, Krstić S (2014) Towards the formalization of properties of cloud-based elastic systems. In: Proceedings of the 6th international workshop on principles of engineering service-oriented and cloud systems, ACM, New York, NY, USA, PESOS 2014, pp 38–47
12. Blochinger W, Michlin W, Weber A (1998) The distributed object-oriented threads system dots. In: Ferreira A, Rolim J, Simon H, Teng SH (eds) Solving irregularly structured problems in parallel. Springer, Heidelberg, pp 206–217
13. Blochinger W, Küchlin W, Ludwig C, Weber A (1999) An object-oriented platform for distributed high-performance symbolic computation. Math Comput Simul 49:161–178
14. Blochinger W, Dangelmayr C, Schulz S (2006) Aspect-oriented parallel discrete optimization on the cohesion desktop grid platform. In: Sixth IEEE international symposium on cluster computing and the grid, 2006. CCGRID 06, vol 1, pp 49–56
15. Bonami P, Lejeune MA (2009) An exact solution approach for portfolio optimization problems under stochastic and integer constraints. Oper Res 57(3):650–670
16. Bui P, Rajan D, Abdul-Wahid B, Izaguirre J, Thain D (2011) Work queue+python: a framework for scalable scientific ensemble applications. In: Workshop on python for high-performance and scientific computing
17. Bungart M, Fohry C (2017) A malleable and fault-tolerant task pool framework for x10. In: IEEE International Conference on Cluster Computing (CLUSTER). IEEE, pp 749–757
18. Cachin C, Guerraoui R, Rodrigues L (2011) Introduction to reliable and secure distributed programming, second edn. Springer, Berlin
19. El-Kalioby M, Abouelhoda M, Krüger J, Giegerich R, Sczyrba A, Wall DP, Tonellato P (2012) Personalized cloud-based bioinformatics services for research and education: use cases and the elastichpc package. BMC Bioinform 13(17):S22
20. Emeras J, Varrette S, Plugaru V, Bouvry P (2019) Amazon elastic compute cloud (ec2) versus in-house hpc platform: a cost analysis. IEEE Trans Cloud Comput 7(2):456–468
21. Feitelson DG, Rudolph L (1996) Toward convergence in job schedulers for parallel supercomputers. In: Feitelson DG, Rudolph L (eds) Job scheduling strategies for parallel processing. Springer, Berlin, pp 1–26
22. Galante G, d Bona LCE (2012) A survey on cloud computing elasticity. In: 2012 IEEE Fifth International Conference on Utility and Cloud Computing, pp 263–270
23. Galante G, Erpen De Bona LC, Mury AR, Schulze B, da Rosa Righi R (2016) An analysis of public clouds elasticity in the execution of scientific applications: a survey. J Grid Comput 14(2):193–216
24. Gautier T, Roch JL, Villard G (1995) Regular versus irregular problems and algorithms. In: Ferreira A, Rolim J (eds) Parallel algorithms for irregularly structured problems. Springer, Berlin, pp 1–25
25. Gendron B, Crainic TG (1994) Parallel branch-and-branch algorithms: survey and synthesis. Oper Res 42(6):1042–1066
26. Grama A, Kumar V (1999) State of the art in parallel search techniques for discrete optimization problems. IEEE Trans Knowl Data Eng 11(1):28–35
27. Grama A, Gupta A, Karypis G, Kumar V (2003) Introduction to parallel computing, 2nd edn. Pearson Education, London
28. Gropp W, Thakur R, Lusk E (1999) Using MPI-2: advanced features of the message passing interface. MIT Press, Cambridge
29. Guo Y, Ghanem M, Han R (2012) Does the cloud need new algorithms? An introduction to elastic algorithms. In: 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings, pp 66–73
30. Gupta A, Kale LV, Gioachin F, March V, Suen CH, Lee BS, Faraboschi P, Kaufmann R, Milojicic D (2013) The who, what, why, and how of high performance computing in the cloud. In: IEEE 5th International Conference on Cloud Computing Technology and Science, vol 1, pp 306–314
31. Gupta A, Sarood O, Kale LV, Milojicic D (2013) Improving hpc application performance in cloud through dynamic load balancing. In: 13th IEEE/ACM international symposium on cluster, cloud, and grid computing, pp 402–409
32. Gupta A, Faraboschi P, Gioachin F, Kale LV, Kaufmann R, Lee B, March V, Milojicic D, Suen CH (2016) Evaluating and improving the performance and scheduling of hpc applications in cloud. IEEE Trans Cloud Comput 4(3):307–321

33. Han R, Ghanem MM, Guo L, Guo Y, Osmond M (2014) Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. Future Gener Comput Syst 32:82–98
34. Haussmann J, Blochinger W, Kuechlin W (2019) Cost-efficient parallel processing of irregularly structured problems in cloud computing environments. Clust Comput 22(3):887–909
35. Haussmann J, Blochinger W, Kuechlin W (2019) Cost-optimized parallel computations using volatile cloud resources. In: Djemame K, Altmann J, Bañares JÁ, Agmon Ben-Yehuda O, Naldi M (eds) Economics of grids, clouds, systems, and services. Springer, Cham, pp 45–53
36. Herbst NR, Kounev S, Reussner R (2013) Elasticity in cloud computing: what it is, and what it is not. In: Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), USENIX, San Jose, CA, pp 23–27
37. Hungershöfer J, Streit A, Wierum JM (2001) Efficient resource management for malleable applications. Tech. Rep. TR-003-01, Paderborn Center for Parallel Computing
38. Jennings B, Stadler R (2015) Resource management in clouds: survey and research challenges. J Netw Syst Manag 23(3):567–619
39. Kautz H, Selman B (1992) Planning as satisfiability. In: Proceedings of the 10th European Conference on Artificial Intelligence, ECAI'92. Wiley, New York, pp 359–363
40. Kehrer S, Blochinger W (2018) Autogenic: automated generation of self-configuring microservices. In: Proceedings of the 8th International Conference on Cloud Computing and Services Science, SciTePress, pp 35–46
41. Kehrer S, Blochinger W (2018) Tosca-based container orchestration on mesos. Comput Sci Res Dev 33(3):305–316
42. Kehrer S, Blochinger W (2019) Elastic parallel systems for high performance cloud computing: state-of-the-art and future directions. Parallel Process Lett 29(02):1950006-1–1950006-20
43. Kehrer S, Blochinger W (2019) Migrating parallel applications to the cloud: assessing cloud readiness based on parallel design decisions. SICS Softw Intensive Cyber Phys Syst 34(2):73–84
44. Kehrer S, Blochinger W (2019) Cloud computing and services science. In: Muñoz VM, Ferguson D, Helfert M, Pahl C (eds) Model-based generation of self-adaptive cloud services. Springer, Berlin, pp 40–63
45. Kehrer S, Blochinger W (2019d) 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
46. Kehrer S, Blochinger W (2019e) 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
47. Kehrer S, Riebandt F, Blochinger W (2019) Container-based module isolation for cloud services. In: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE). IEEE, pp 177–186
48. Kehrer S, Scheffold J, Blochinger W (2019) Serverless skeletons for elastic parallel processing. In: 2019 IEEE 5th International Conference on Big Data Intelligence and Computing (DATACOM). IEEE, pp 185–192
49. Lai TH, Sahni S (1984) Anomalies in parallel branch-and-bound algorithms. Commun ACM 27(6):594–602
50. Li G, Wah BW (1986) Coping with anomalies in parallel branch-and-bound algorithms. IEEE Trans Comput C–35(6):568–573
51. Liu F, Weissman JB (2015) Elastic job bundling: an adaptive resource request strategy for large-scale parallel applications. In: SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–12
52. Lorido-Botran T, Miguel-Alonso J, Lozano JA (2014) A review of auto-scaling techniques for elastic applications in cloud environments. J Grid Comput 12(4):559–592
53. Mauch V, Kunze M, Hillenbrand M (2013) High performance cloud computing. Future Gener Comput Syst 29(6):1408–1416
54. Moldovan D, Copil G, Truong H, Dustdar S (2013) Mela: monitoring and analyzing elasticity of cloud services. In: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, pp 80–87
55. Netto MAS, Calheiros RN, Rodrigues ER, Cunha RLF, Buyya R (2018) Hpc cloud for scientific and business applications: taxonomy, vision, and research challenges. ACM Comput Surv (CSUR) 51(1):81–829

56. Olivier S, Huan J, Liu J, Prins J, Dinan J, Sadayappan P, Tseng CW (2007) Uts: an unbalanced tree search benchmark. In: Almási G, Caşcaval C, Wu P (eds) Languages and compilers for parallel computing. Springer, Berlin, Heidelberg, pp 235–250
57. Poldner M, Kuchen H (2008) Algorithmic skeletons for branch and bound. In: Filipe J, Shishkov B, Helfert M (eds) Software and data technologies. Springer, Berlin, pp 204–219
58. Posner J, Fohry C (2018) Hybrid work stealing of locality-flexible and cancelable tasks for the apgas library. J Supercomput 74(4):1435–1448
59. Prabhakaran A, Lakshmi L (2018) Cost-benefit analysis of public clouds for offloading in-house hpc jobs. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp 57–64
60. Rajan D, Thain D (2017) Designing self-tuning split-map-merge applications for high cost-efficiency in the cloud. IEEE Trans Cloud Comput 5(2):303–316
61. Rajan D, Canino A, Izaguirre JA, Thain D (2011) Converting a high performance application to an elastic cloud application. In: IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, pp 383–390
62. Rajan D, Thrasher A, Abdul-Wahid B, Izaguirre JA, Emrich S, Thain D (2013) Case studies in designing elastic applications. In: 2013 13th IEEE/ACM international symposium on cluster, cloud, and grid computing, pp 466–473
63. Ralphs T (2003) Parallel branch and cut for capacitated vehicle routing. Parallel Comput 29(5):607–629
64. Rathnayake S, Loghin D, Teo YM (2017) Celia: cost-time performance of elastic applications on cloud. In: 46th International Conference on Parallel Processing (ICPP), pp 342–351
65. Raveendran A, Bicer T, Agrawal G (2011) A framework for elastic execution of existing mpi programs. In: 2011 IEEE international symposium on parallel and distributed processing workshops and Ph.D Forum, pp 940–947
66. Rodrigues VF, da Rosa Righi R, da Costa CA, Singh D, Munoz VM, Chang V (2018) Towards combining reactive and proactive cloud elasticity on running hpc applications. In: Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security: IoTBDS, SciTePress, pp 261–268
67. Ronconi DP (2005) A branch-and-bound algorithm to minimize the makespan in a flowshop with blocking. Ann Oper Res 138(1):53–65
68. da Rosa Righi R, Rodrigues VF, da Costa CA, Kreutz D, Heiss HU (2015) Towards cloud-based asynchronous elasticity for iterative HPC applications. J Phys Conf Ser 649:012006
69. da Rosa Righi R, Rodrigues VF, da Costa CA, Galante G, de Bona LCE, Ferreto T (2016) Autoelastic: automatic resource elasticity for high performance applications in the cloud. IEEE Trans Cloud Comput 4(1):6–19
70. da Rosa Righi R, Rodrigues VF, Rostirolla G, da Costa CA, Roloff E, Navaux POA (2018) A lightweight plug-and-play elasticity service for self-organizing resource provisioning on parallel applications. Future Gener Comput Syst 78:176–190
71. Schmidt MC, Samatova NF, Thomas K, Park BH (2009) A scalable, parallel algorithm for maximal clique enumeration. J Parallel Distrib Comput 69(4):417–428
72. Schulz S, Blochinger W (2010) Parallel sat solving on peer-to-peer desktop grids. J Grid Comput 8(3):443–471
73. Schulz S, Blochinger W, Held M, Dangelmayr C (2008) Cohesion a microkernel based desktop grid platform for irregular task-parallel applications. Future Gener Comput Syst 24(5):354–370
74. Shudler S, Calotoiu A, Hoefler T, Wolf F (2017) Isoefficiency in practice: configuring and understanding the performance of task-based applications. In: Proceedings of the 22nd ACM SIGPLAN symposium on principles and practice of parallel programming, ACM, New York, NY, USA, PPoPP'17, pp 131–143
75. Sinz C, Kaiser A, Küchlin W (2003) Formal methods for the validation of automotive product configuration data. Ai Edam 17(1):75–97
76. Stephan P, Brayton RK, Sangiovanni-Vincentelli AL (1996) Combinational test generation using satisfiability. IEEE Trans Comput Aided Des Integr Circuits Syst 15(9):1167–1176
77. Sun Y, Wang CL (2003) Solving irregularly structured problems based on distributed object model. Parallel Comput 29(11–12):1539–1562
78. Utrera G, Corbalan J, Labarta J (2004) Implementing malleability on mpi jobs. In: Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004, pp 215–224

79. Vadhiyar SS, Dongarra JJ (2003) Srs: a framework for developing malleable and migratable parallel applications for distributed systems. Parallel Process Lett 13(02):291–312
80. Varghese B, Buyya R (2018) Next generation cloud computing: new trends and research directions. Future Gener Comput Syst 79:849–861
81. Vecchiola C, Pandey S, Buyya R (2009) High-performance cloud computing: a view of scientific applications. In: 10th international symposium on pervasive systems, algorithms, and networks (ISPAN). IEEE, pp 4–16
82. Vu TT, Derbel B (2014) Link-heterogeneous work stealing. In: 2014 14th IEEE/ACM international symposium on cluster, cloud and grid computing, pp 354–363
83. Yang J, He Q (2018) Scheduling parallel computations by work stealing: a survey. Int J Parallel Program 46(2):173–197
84. Yang X, Wallom D, Waddington S, Wang J, Shaon A, Matthews B, Wilson M, Guo Y, Guo L, Blower JD, Vasilakos AV, Liu K, Kershaw P (2014) Cloud computing in e-science: research challenges and opportunities. J Supercomput 70(1):408–464
85. Zhang J, Lu X, Panda DKD (2017) Designing locality and numa aware mpi runtime for nested virtualization based hpc cloud with sr-iov enabled infiniband. In: Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ACM, New York, NY, USA, VEE'17, pp 187–200

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.