Development and Operation of Elastic Parallel Tree Search Applications Using TASKWORK

Stefan Kehrer(✉) and Wolfgang Blochinger

Parallel and Distributed Computing Group, Reutlingen University,
Alteburgstrasse 150, 72762 Reutlingen, Germany
{stefan.kehrer,wolfgang.blochinger}@reutlingen-university.de

Abstract. Cloud resources can be dynamically provisioned according to application-specific requirements and are payed on a per-use basis. This gives rise to a new concept for parallel processing: Elastic parallel computations. However, it is still an open research question to which extent parallel applications can benefit from elastic scaling, which requires resource adaptation at runtime and corresponding coordination mechanisms. In this work, we analyze how to address these system-level challenges in the context of developing and operating elastic parallel tree search applications. Based on our findings, we discuss the design and implementation of TASKWORK, a cloud-aware runtime system specifically designed for elastic parallel tree search, which enables the implementation of elastic applications by means of higher-level development frameworks. We show how to implement an elastic parallel branch-and-bound application based on an exemplary development framework and report on our experimental evaluation that also considers several benchmarks for parallel tree search.

Keywords: Cloud computing · Parallel computing · Task parallelism · Elasticity · Branch-and-bound

1 Introduction

Many cloud providers, including Amazon Web Services (AWS)[1] and Microsoft Azure[2], introduced new cloud offerings optimized for High Performance Computing (HPC) workloads. Whereas traditional HPC clusters are based on static resource assignment and job scheduling, cloud environments provide attractive benefits for parallel applications such as on-demand access to compute resources, pay-per-use, and elasticity [12,31]. Specifically, elasticity, i.e., the ability to provision and decommission compute resources at runtime, introduces a new concept: Fine-grained cost control per application run by means of elastic parallel computations [11,12,19,24,36]. This fundamentally new concept in parallel computing

[1] https://aws.amazon.com.
[2] https://azure.microsoft.com.

© Springer Nature Switzerland AG 2020
D. Ferguson et al. (Eds.): CLOSER 2019, CCIS 1218, pp. 42–65, 2020.
https://doi.org/10.1007/978-3-030-49432-2_3

leads to new opportunities and challenges thus stimulating new research efforts and approaches. For instance, processing time and/or the quality of results can be related to costs, allowing versatile optimizations at runtime [19,24,36].

During the last years, there has been a growing interest to make parallel applications cloud-aware [11,15,17,27,37]. In particular, applications have to cope with the effects of virtualization and resource pooling causing fluctuations in processing times [17]. Existing research also studies how to employ elasticity for applications with simple communication and coordination patterns (e.g., iterative-parallel workloads) [11,37]. In these cases, problems are decomposed into a set of independent tasks, which can be farmed out for distributed computation. However, it is still an open research question to which extent other parallel application classes can benefit from cloud-specific properties, how to leverage elasticity in these cases, and how to ensure cloud-aware coordination of distributed compute resources.

In this work, we discuss how to tackle these challenges for parallel tree search applications. These applications are less sensitive to heterogeneous processing speeds when compared to data-parallel, tightly-coupled applications [15,16], but comprise unstructured interaction patterns and complex coordination requirements. Prominent meta-algorithms based on the parallel tree search processing technique include branch-and-bound and backtracking search with many applications in biochemistry, electronic design automation, financial portfolio optimization, production planning and scheduling, as well as fleet and vehicle scheduling. We discuss the challenges that have to be addressed to make these applications cloud-aware and present TASKWORK - a cloud-aware runtime system that provides a comprehensive foundation for implementing and operating elastic parallel tree search applications in cloud environments. In particular, we make the following contributions: (1) We discuss how to construct a cloud-aware runtime system for parallel tree search applications. (2) We describe the design and implementation of TASKWORK, an integrated runtime system based on our findings and solve corresponding coordination problems based on Apache ZooKeeper[3]. (3) We present a development framework for elastic parallel branch-and-bound applications, which aims to minimize programming effort. (4) We employ a canonical branch-and-bound application as well as several benchmarks to evaluate the performance of TASKWORK in our OpenStack-based private cloud environment.

This work is based on previous research contributions that have been published in the paper *TASKWORK: A Cloud-aware Runtime System for Elastic Task-parallel HPC Applications* [28], which has been presented at the *9th International Conference on Cloud Computing and Services Science*. We extend our former work by discussing the applicability of the presented concepts in the context of parallel tree search applications. Moreover, we provide an extensive evaluation of TASKWORK based on several benchmarks, which are commonly employed to evaluate architectures designed for parallel tree search.

[3] https://zookeeper.apache.org.

This work is structured as follows. In Sect. 2, we discuss the characteristics of parallel tree search applications as well as ZooKeeper and related work. Section 3 describes the conceptualization of a cloud-aware runtime system for elastic parallel tree search in the cloud. In Sect. 4, we present TASKWORK - our integrated runtime system for elastic parallel tree search applications. We elaborate on an elastic branch-and-bound development framework and describe its use in Sect. 5. The results of our extensive experimental evaluation are presented in Sect. 6. Section 7 concludes this work.

2 Fundamentals and Related Work

In this section, we examine the characteristics of parallel tree search applications, describe ZooKeeper, and discuss existing research closely related to our work.

2.1 Parallel Tree Search

We specifically focus on parallel tree search processing technique. Commonly employed meta-algorithms based on parallel tree search include branch-and-bound and backtracking search. They are typically used to solve enumeration, decision, and optimization problems - including boolean satisfiability, constraint satisfaction, and graph search problems - with many applications in fields such as biochemistry, electronic design automation, financial portfolio optimization, production planning and scheduling, as well as fleet and vehicle scheduling. These algorithms search solutions in very large state spaces and employ advanced branching and pruning operations/backtracking mechanisms to make the search procedure for problem instances of practical relevance efficient.

Parallel execution is most often accomplished by splitting the state space tree into tasks that can be executed independently of each other by searching a solution in the corresponding subtree. This approach is also called exploratory parallelism (or space splitting [13]). However, because the shape and size of the search tree (and its subtrees) are highly influenced by branching and pruning operations, these applications exhibit a high degree of irregularity. Thus, to exploit a large number of (potentially distributed) compute resources efficiently, task generation has to be executed in a dynamic manner by creating new tasks at runtime. Additionally, these newly generated tasks have to be distributed among compute nodes to avoid idling processing units. This procedure is also called dynamic task mapping (or task scheduling).

The high degree of irregularity constitutes the major source of parallel overhead and thus affects the performance and scaling behavior of parallel tree search applications. Moreover, additional communication requirements stem from knowledge sharing mechanisms that are required to implement meta-algorithms such as branch-and-bound and backtracking search. In this context, knowledge sharing often means communicating bounds [13] or lemmas [38] across tasks at runtime to make the search procedure more efficient by avoiding the exploration of specific subtrees.

Due to the dynamic exploration of the search space combined with problem-specific branching and pruning operations/backtracking mechanisms, resource requirements of parallel tree search applications are not known in advance. This makes them an ideal candidate for cloud adoption as cloud environments provide on-demand access to resources and enable an application to scale elastically. Moreover, they are less sensitive to heterogeneous processing speeds when compared to data-parallel, tightly-coupled applications [15,16].

2.2 ZooKeeper

ZooKeeper has been designed to ease the implementation of coordination, data distribution, synchronization, and meta data management in distributed systems [20]. Many prominent software projects rely on ZooKeeper including the Apache projects Hadoop[4] and Kafka[5]. It provides an interface that enables clients to read from and write to a tree-based data structure consisting of data registers called *znodes*. Internally, data is replicated across a set of ZooKeeper servers. Each ZooKeeper server accepts client connections and executes requests in FIFO order per client session. A feature called *watches* enables clients to register for notifications of changes without periodic polling. Each server answers read operations locally resulting in eventual consistency. On the other hand, ZooKeeper guarantees writes to be atomic [20]. ZooKeeper's design principles ensure both high availability of stored data and high-performance data access by providing a synchronous and an asynchronous API.

Specifically in cloud environments, coordination primitives such as leader election and group membership are essentially required to deal with a varying number of compute nodes. Based on ZooKeeper, leader election and group membership can be implemented in a straightforward manner [21]. However, specific challenges arise in the context of parallel tree search applications: Global variables have to be synchronized across tasks, which imposes additional dependencies, and as tasks can be generated at each node, a termination detection mechanism is required to detect when the computation has been completed. We show how to employ ZooKeeper to tackle these challenges.

2.3 Related Work

In the past, researchers mainly investigated how to make cloud environments HPC-aware [30]. By exploiting HPC-aware cloud offerings, many parallel applications benefit from an on-demand provisioned execution environment that can be payed on a per-use basis and individual configuration of compute resources, without any modifications to the application itself. This is specifically attractive for applications implemented based on the Single Program Multiple Data (SPMD) model (especially supported by MPI) [27]. However, we can also see a growing interest to make parallel applications cloud-aware [11,15–17,37] with

[4] http://hadoop.apache.org.
[5] https://kafka.apache.org.

the motivation to exploit either low-cost standard cloud offerings or to make use of advanced cloud features beyond a simple copy & paste migration approach [27]. Existing research discusses how to adapt parallel applications and parallel system architectures to make them cloud-aware. The authors of [12] propose the development of new frameworks for building parallel applications optimized for cloud environments and discuss the importance of application support with respect to elasticity. We follow this approach by presenting a runtime system that does most of the heavy lifting to implement elastic parallel applications.

The authors of [15] present an in-depth performance analysis of different applications. Based on their measurements, the authors describe several strategies to make both parallel applications cloud-aware and cloud environments HPC-aware. A major issue to make parallel applications cloud-aware is the specification of the optimal task size to balance various sources of overhead. In [17], the problem of fluctuations in processing times is addressed, which specifically affects tightly-coupled parallel applications. The authors introduce a dynamic load balancing mechanism that monitors the load of each vCPU and reacts to a measured imbalance. Whereas this approach is based on task overdecomposition to ensure dynamic load balancing, our runtime system actively controls the logical parallelism of an application to minimize task management overhead. However, it is still an open research question if applications without dynamic task parallelism can benefit from such an approach.

The authors of [37] employ the Work Queue framework to develop elastic parallel applications. The Work Queue framework is designed for scientific ensemble applications and provides a master/worker architecture with an elastic pool of workers. The presented case study considers a parallel application for replica exchange molecular dynamics (REMD), which can be considered to be iterative-parallel. The authors of [11] present an approach to enable elasticity for iterative-parallel applications by employing a master/worker architecture. They make use of an asynchronous elasticity mechanism, which employs non-blocking scaling operations. Whereas we specifically consider parallel tree search applications, TASKWORK also makes use of asynchronous scaling operations that do not block the computation.

Task-based parallelism was originally designed to exploit shared memory architectures and used by systems such as Cilk [7]. A major characteristic of task-parallel approaches is that tasks can be assigned dynamically to worker threads, which ensures load balancing and thus effectively reduces idle time. This approach also provides attractive advantages beyond shared memory architectures and has been adopted for different environments including compute clusters [2,5,6] and grids [1]. As a result, the distributed task pool model has attracted considerable research interest. The authors of [33] present a skeleton for C++, which supports distributed memory parallelism for branch-and-bound applications. Their skeleton uses MPI communication mechanisms and is not designed to be cloud-aware. The authors of [9] present a distributed task pool implementation based on the parallel programming language X10, which follows the Partitioned Global Address Space (PGAS) programming model. COHE-

SION is a microkernel-based platform for desktop grid computing [4,39]. It has been designed with an emphasis on task-parallel problems that require dynamic problem decomposition and also provides an abstraction layer for developers based on its system core. Whereas COHESION supports similar applications, it is designed to tackle the challenges of desktop grids such as limited connectivity and control as well as high resource volatility. In contrast to desktop grids, cloud resources can be configured to consider application-specific requirements and controlled by employing an elasticity controller [24]. Moreover, compute resources are billed by a cloud provider, whereas desktop grids make use of available resources donated by contributors.

3 Constructing a Cloud-Aware Runtime System

To particularly benefit from cloud-specific characteristics, developing elastic parallel applications is a fundamental problem that has to be solved [12]. At the core of this problem lies the required dynamic adaptation of parallelism. At all times, 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. Traditionally, the number of processing units has been considered as static. In cloud environments, however, the number of processing units can be scaled at runtime by employing an elasticity controller. As a result, applications have to dynamically adapt the degree of logical parallelism based on a dynamically changing physical parallelism. At the same time, adapting the logical parallelism and mapping the logical parallelism to the physical parallelism incurs overhead (in form of excess computation, communication, and idle time). Consequently, elastic parallel applications have to continuously consider a trade-off between the perfect fit of logical and physical parallelism on the one side and minimizing overhead resulting from the adaptation of logical parallelism and its mapping to the physical parallelism on the other. Hence, enabling elastic parallel computations leads to many system-level challenges that have to be addressed to ensure a high efficiency.

Because we specifically focus on parallel tree search applications, which require dynamic task parallelism, the degree of logical parallelism can be defined as the current number of tasks. We argue that a cloud-aware runtime system is required that transparently controls the parallelism of an application to ensure elastic scaling. Figure 1 shows our conceptualization of such a runtime system. It allows developers to mark parallelism in the program, automatically adapts the logical parallelism by generating tasks whenever required, and exploits available processing units with maximum efficiency by mapping the logical parallelism to the physical parallelism. An application based on such a runtime system is elastically scalable: Newly added compute nodes automatically receive tasks by means of dynamic decomposition and load balancing. A task migration mechanism releases compute nodes that have been selected for decommissioning (cf. Fig. 1). Our approach is not limited to any specific cloud management approach or tooling: An elasticity controller may comprise any kind of external

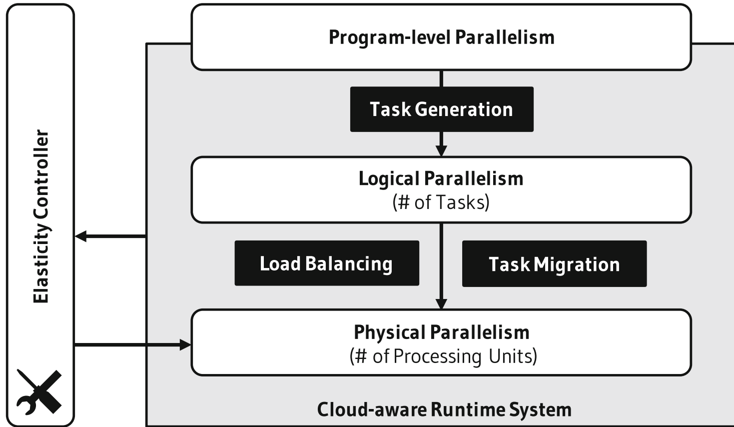

Fig. 1. The described cloud-aware runtime system adapts the logical parallelism by generating tasks dynamically, handles load balancing and task migration, and thus enables elastic parallel computations [28].

decision making logic (e.g., based on execution time, the quality of results, or monetary costs) that finally adapts the number of processing units (i.e., the physical parallelism). An example for such an elasticity controller is given in [19], where monetary costs are considered to control the physical parallelism. In this work, we focus on elastic parallel computations and address related system-level challenges.

Besides elasticity, the characteristics of cloud environments lead to new architectural requirements that have to be considered by parallel applications [25]. Due to virtualization and resource pooling (leading to CPU timesharing and memory overcommitment), fluctuations in processing times of individual processing units can often be observed [15]. Thus, in cloud environments, tasks should be coupled in a loosely manner by employing asynchronous communication methods. Similarly, inter-node synchronization should be loosely coupled while guaranteeing individual progress. A runtime system built for the cloud has to provide such asynchronous communication and synchronization mechanisms thus releasing developers from dealing with these low-level complexities.

4 Design and Implementation of TASKWORK

In this section, we describe the design and implementation of TASKWORK, a cloud-aware runtime system specifically designed for parallel tree search applications according to the principles discussed in Sect. 3. TASKWORK comprises several components that enable elastic parallel computations (cf. Ⓐ, Fig. 2) and solve coordination problems based on ZooKeeper (cf. Ⓑ, Fig. 2). Based on these system-level foundations, higher-level development frameworks and programming models can be built (cf. Ⓒ, Fig. 2), which facilitate the implementa-

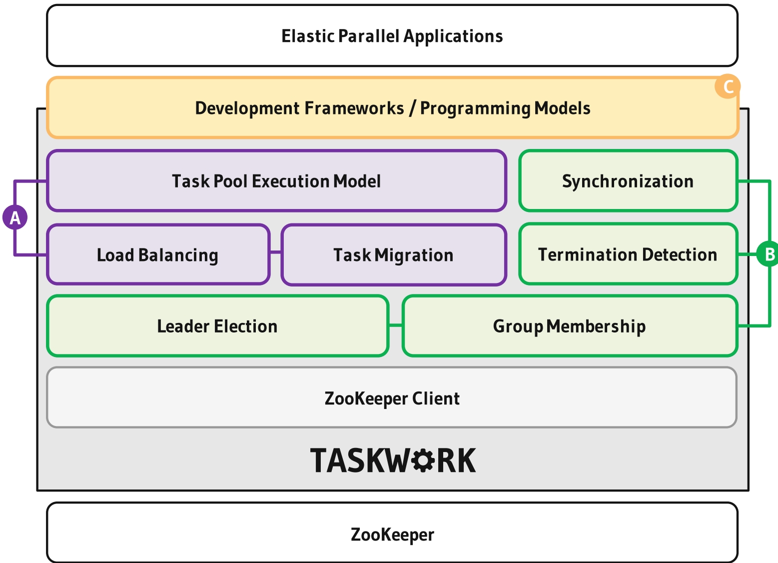

Fig. 2. The components of TASKWORK enable elastic parallel computations based on the task pool execution model, solve coordination problems based on ZooKeeper, and support the construction of higher-level development frameworks and programming models [28].

tion of elastic parallel applications. TASKWORK enables distributed memory parallelism by coordinating a set of distributed compute nodes based on the task pool execution model. Our research prototype is implemented in Java.

In this section, we briefly describe the well-known task pool execution model that we use to manage tasks, before the components of TASKWORK are described in detail.

4.1 Task Pool Execution Model

The task pool execution model [14] decouples task generation and task processing by providing a data structure that can be used to store dynamically generated tasks and to fetch these tasks later for processing. It has been extensively used in the context of parallel tree search applications [33,38,39]. We employ the task pool execution model as a foundation to enable elastic parallel computations according to the concepts depicted in Fig. 1: The task pool manages tasks generated at runtime (defining the logical parallelism) and provides an appropriate interface for load balancing and task migration mechanisms that enable elastic parallel computations.

The task pool execution model can be implemented in a centralized or a distributed manner. The centralized task pool execution model refers to a task pool located at a single compute node that is accessed by all other compute nodes to store and fetch tasks. In the context of distributed memory parallelism,

this means that tasks always have to be transferred over the network, e.g., for load balancing purposes. The centralized task pool execution model is easy to implement because the centralized instance has complete knowledge on the state of the system, e.g., which compute node is executing which task. On the other hand, the centralized task pool might become a sequential bottleneck for large number of compute nodes accessing the task pool. The distributed task pool execution model, on the other hand, places a task pool instance at each compute node. It thus decouples compute nodes from each other leading to a highly scalable system. On the contrary, coordination becomes a non-trivial task because individual compute nodes only have partial knowledge. This specifically holds in cloud environments, where compute nodes are provisioned and decommissioned at runtime.

In this work, we employ the distributed task pool execution model, which enables compute nodes to store generated tasks locally and, in general, provides a better scalability [33]. Whereas the distributed task pool execution model leads to an asynchronous system, thus matching the characteristics of cloud environments, one has to deal with the aforementioned challenges. To deal with these drawbacks, we enhance it with scalable coordination and synchronization mechanisms based on ZooKeeper.

4.2 Leader Election

TASKWORK implements ZooKeeper-based leader election to designate a single coordinator among the participating compute nodes. This coordinator takes care of submitting jobs to the system, processes the final result, and controls cloud-related coordination operations such as termination detection. ZooKeeper renders leader election a rather trivial task [21]. Therefore, each node tries to write its endpoint information to the */coordinator* znode. If the write operation succeeds, the node becomes the coordinator. Otherwise, if the */coordinator* znode exists, the node participates as compute node.

4.3 Group Membership

As compute nodes might be added or removed at runtime by means of elastic scaling, cloud-based systems are highly dynamic. Thus, a group membership component is required, which provides up-to-date views on the instance model, i.e., the list of all currently available compute nodes. To this end, compute nodes automatically register themselves during startup by creating an ephemeral child znode under the */computeNodes* znode containing their endpoint information. The creation of the child znode makes use of ZooKeeper's *sequential* flag that creates a unique name assignment [20]. Changes in group membership are obtained by all other compute nodes by watching the */computeNodes* znode.

4.4 Load Balancing

Load balancing is a fundamental aspect in cloud environments to exploit newly added compute resources efficiently. Moreover, it is a strong requirement of parallel tree search applications due to dynamic problem decomposition. Load balancing can be accomplished by either sending tasks to other compute nodes (work sharing) or by fetching tasks from other nodes (work stealing) [7]. Because sending tasks leads to overhead, we favor work (task) stealing because communication is only required when a compute node runs idle. Load balancing is accomplished by observing changes in the local task pool. Whenever the local task pool is empty and all worker threads are idle, *task stealing* is initiated. Task stealing is an approach where idle nodes send work requests to other nodes in the cluster. These nodes answer the request by sending a task from their local task pool to the remote node.

Because the distributed task pool execution model lacks knowledge about which compute nodes are busy and which are idling, we employ randomized task stealing [8]. To deal with a changing number of compute nodes over time, up-to-date information on the currently available compute nodes is required. This information is provided by the group membership component (cf. Sect. 4.3).

4.5 Task Migration

To enable the decommissioning of compute resources at runtime, unfinished work has to be sent to remaining compute nodes. This is ensured by TASKWORK's task migration component. Compute nodes that have been selected for decommissioning store the current state of tasks being executed, stop their worker threads, and send all local tasks to remaining compute nodes. Technically, the task migration component registers for the POSIX SIGTERM signal. This signal is triggered by Unix-like operating systems upon termination, which allows TASKWORK to react to a requested termination without being bound to specific cloud management tooling but instead relying on operating system mechanisms. Also note that POSIX signals are supported by state-of-the-art container runtime environments such as Kubernetes[6], where they are used to enable graceful shutdown procedures. As a result, TASKWORK can be controlled by any cloud management tool (provided by a specific cloud provider or open source) and hence enables a best-of-breed tool selection. Furthermore, this approach ensures that TASKWORK can be deployed on any operating system that supports POSIX signals and the Java Runtime Environment (JRE), thus ensuring a high degree of portability.

Application developers simply have to specify an optimal interruption point in their program to support task migration. The `migrate` operation can be used to check if a task should be migrated (for an example see Sect. 5.2). TASKWORK employs weak migration of tasks. This means that a serialized state generated

[6] https://kubernetes.io.

from a task object is transferred across the network. To facilitate the migration process, application-specific snapshotting mechanisms can be provided by developers.

4.6 Termination Detection

Traditionally, distributed algorithms for termination detection (wave-based or based on parental responsibility) have been preferred due to their superior scalability characteristics [14]. However, maintaining a ring (wave-based) or tree (parental responsibility) structure across compute nodes in the context of an elastically scaled distributed system imposes significant overhead. To deal with this issue, TASKWORK employs ZooKeeper-based termination detection, which has been described in [28]. In summary, this approach maintains a tree-based task dependency structure stored in ZooKeeper, which is dynamically updated at runtime.

4.7 Synchronization of Global Variables

As discussed in Sect. 2.1, many meta-algorithms such as branch-and-bound rely on knowledge sharing across tasks at runtime to make the search procedure more efficient by avoiding the exploration of specific subtrees. TASKWORK supports knowledge sharing in form of global variables that are automatically synchronized across tasks. Global variables can be used to build application-specific development frameworks or programming models. The process of synchronization considers three hierarchy levels: (1) task-level variables, which are updated for each task executed by a worker thread, (2) node-level variables, which are updated on each compute node, and (3) global variables. Task-level variables are typically updated by the implemented program and thus managed by the application developer. To synchronize node-level variables, we provide two operations: `getVar` for obtaining node-level variables and `setVar` for setting node-level variables. Whenever a node-level variable changes its value, we employ ZooKeeper to update this variable globally, which enables synchronization across all distributed compute nodes. These generic operations allow developers to address application-specific synchronization requirements, while TASKWORK handles the process of synchronization.

By following this approach, small-sized variables can be synchronized across the distributed system. However, frequent data synchronization leads to overhead and should be used carefully and only for small data.

4.8 Development Frameworks and Programming Models

TASKWORK enables the construction of higher-level development frameworks and programming models based on a generic task abstraction that allows the specification of custom task definitions. The essential idea is that, as outlined in Sect. 3, developers only mark program-level parallelism while task generation,

load balancing, and task migration are handled automatically thus ensuring elastic parallel computations. To define program-level parallelism, application developers specify an application-specific `split` operation based on the generic task abstraction to split work from an existing task. Afterwards, this `split` operation can be used for implementing any application program that dynamically creates tasks (for an example see Sect. 5.2).

Two execution modes for the splitting mechanism are provided: *Definite* and *potential splitting*. Whereas definite splitting directly creates new tasks by means of the `split` operation, potential splitting adapts the logical parallelism (number of tasks) in an automated manner. By following the second approach, application developers also implement the `split` operation in an application-specific manner, but only specify a potential splitting point in their application program with the `potentialSplit` operation. In line with the conceptualization discussed in Sect. 3, the `potentialSplit` operation is used to mark program-level parallelism and TASKWORK decides at runtime whether to create new tasks or not depending on the current system load. Thus, potential splitting automatically adapts the number of tasks generated and thus controls the logical parallelism of the application (cf. Fig. 1). As a result, TASKWORK manages the trade-off between perfect fit of logical and physical parallelism and minimizing overhead resulting from task generation and task mapping as discussed in Sect. 3. Different policies can be supplied to configure how this trade-off is handled. For example, tasks can be generated on-demand, i.e., when another compute node requests a task by means of work stealing (cf. Sect. 4.4). Alternatively, tasks can be generated when the number of tasks in the local task pool drops below a configurable threshold. By default, TASKWORK uses the on-demand task generation policy. We recognized that on-demand task generation is, in many cases, more efficient because formerly generated tasks might contain a subtree that has already been proven to be obsolete. Thus, threshold-based task generation often results in unnecessary transferal of tasks over the network, leading to additional overhead.

5 Elastic Branch-and-Bound Development Framework

In this section, we describe a development framework for elastic parallel branch-and-bound applications based on TASKWORK's generic task abstraction. Branch-and-bound is a well-known meta-algorithm for search procedures. It is considered to be one of the major computational patterns for parallel processing [3]. In the following, we briefly explain the branch-and-bound approach and show how to employ our framework to develop an example application.

5.1 Branch-and-Bound Applications

We explain the branch-and-bound approach by employing the Traveling Salesman Problem (TSP) as canonical example application. The TSP states that a salesman has to make a tour visiting n cities exactly once while finishing at the city he starts from. The problem can be modeled as a complete graph with n

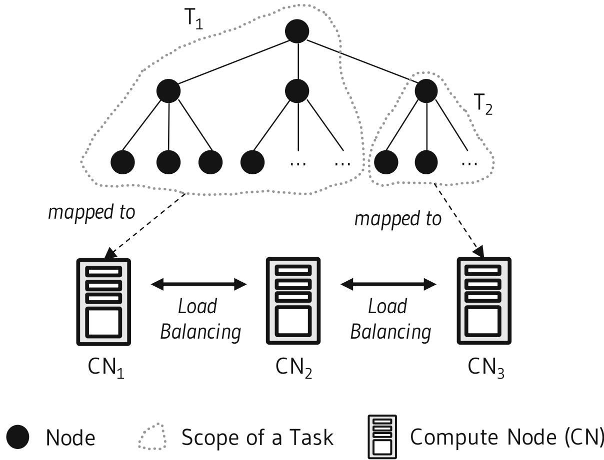

Fig. 3. To enable parallel processing, the state space tree is cut into several tasks, each capturing a subproblem of the initial problem. Note that tasks have to be created at runtime to avoid load imbalance.

vertices, where each vertex represents a city and each edge a path between two cities. A nonnegative cost $c(i, j)$ occurs to travel from city i to city j. The optimization goal is to find a tour whose total cost, i.e., the sum of individual costs along the paths, is minimum [10].

All feasible tours can be explored systematically by employing a *state space tree* that enumerates all states of the problem. The initial node (the root node of the state space tree) represents the city the salesman starts from. From this and all following cities, the salesman can follow any path to travel to one of the unvisited cities, which is represented as a new node in the state space tree. At some point, all cities have been visited thus leading to a leaf node in the state space tree, which represents a tour. Each node can be evaluated with respect to its cost by summing up the individual costs of all paths taken. This also holds for leaf nodes in the state space tree representing a tour. A search procedure can be applied that dynamically explores the complete state space tree and finally finds a tour with minimum cost. However, brute force search cannot be applied to large state space trees efficiently. Instead of enumerating all possible states, branch-and-bound makes use of existing knowledge to search many paths in the state space tree only implicitly. We describe the underlying principles, which make the search procedure efficient, in the following.

If the current node is not a leaf node, the next level of child nodes is generated by visiting all unvisited cities that are directly accessible. Each of these child nodes leads to a set of disjoint tours. Generating new nodes is referred to as *branching*. If the current node is a leaf node, we evaluate the tour represented by this node with respect to its total cost.

At runtime, the tour whose total cost is known to be minimum at a specific point in time defines an upper bound for the ongoing search procedure. Any intermediate node in the state space tree that evaluates to a higher cost can be proven to lead to a tour with higher total costs and thus has not to be explored any further. On the other hand, lower bounds can be calculated by solving a relaxed version of the problem based on the current state [40]. We calculate the lower bound by adding the weight of a minimum spanning tree (MST) of the not-yet visited cities to the current path [2,40]. The MST itself is calculated based on Prim's algorithm [35]. We can prune parts of the state space tree if the calculated lower bound of the current node is larger or equal to the current upper bound (because the TSP is a minimization problem). The pruning operation is essential to make branch-and-bound efficient.

Following the branch-and-bound approach, a problem is decomposed into subproblems at runtime. Each of these subproblems captures a part of the state space tree and can be solved in parallel. Technically, these subproblems are described by a set of tasks, which can be distributed across available compute nodes. However, several challenges arise when we map branch-and-bound applications to parallel architectures: Work anomalies are present, which means that the amount of work differs between sequential and parallel processing as well as across parallel application runs. Additionally, branch-and-bound applications are highly irregular, i.e., task sizes are not known a priori and unpredictable by nature. Consequently, solving the TSP requires the runtime system to cope with dynamic problem decomposition and load balancing to avoid idling processing units. Every task that captures a specific subproblem can produce new child tasks (cf. Fig. 3). Thus, termination detection is another strong requirement to detect if a computation has been completed. Additionally, updates on the upper bound have to be distributed fast to enable efficient pruning for subproblems processed simultaneously in the distributed system.

5.2 Design and Use of the Development Framework

In the following, we describe a development framework for elastic branch-and-bound on top of TASKWORK. We employ the TSP as an example application to show how to use the framework. Elastic parallel applications can be implemented with this framework without considering low-level, technical details.

TASKWORK provides a generic task abstraction that can be used to build new development frameworks and programming models. In the context of branch-and-bound, we define a task as the traversal of the subtrees rooted at all unvisited input nodes. Additionally, each task has access to the graph structure describing the cities as vertices and the paths as edges. This graph structure guides the exploratory construction of the state space tree. All visited cities are marked in the graph. This representation allows to split the currently traversed state space tree to generate new tasks.

New tasks have to be created at runtime to keep idling processing units and newly added ones busy. Therefore, the branch-and-bound task definition

```
1  public void search() {
2    while(!openNodes.isEmpty()){
3      if(migrate()) return;
4
5      Node currentNode = openNodes.getNext();
6
7      getUpperBound();
8
9      Node[] children = currentNode.branch();
10     for(Node child : children) {
11       if(child.isLeafNode()) {
12         if(child.getCost() < current_best_cost){
13           current_best_cost = child.getCost();
14           current_best_tour = child.getPath();
15           setUpperBound();
16         }
17       }else if(child.getLowerBound() < current_best_cost){
18         openNodes.add(child);
19       }
20     }
21
22     potentialSplit();
23   }
24 }
```

Fig. 4. The elastic branch-and-bound development framework allows developers to implement parallel search procedures without considering low-level details such as concurrency, load balancing, synchronization, and task migration [28].

allows the specification of an application-specific `split` operation. This operation branches the state space tree by splitting off a new task from a currently executed task. This split-off task can be processed by another worker thread running on another compute node. To limit the amount of tasks generated, we make use of TASKWORK's potential splits, i.e, the `split` operation is only triggered, when new tasks are actually required. As depicted in Fig. 4, here, the `potentialSplit` operation is executed after a node has been evaluated. TASKWORK decides if a split is required. If so, it executes the application-specific `split` operation that takes nodes from the `openNodes` list to create a new (disjoint) task. Otherwise it proceeds regularly, i.e., it evaluates the next node. In the following, we describe how to implement task migration, bound synchronization, and termination detection based on TASKWORK.

Task Migration. To enable task migration, developers check if migration is required (cf. Fig. 4). In this case, a task simply stops its execution. The migration process itself is handled by TASKWORK. This means that a compute node that has been selected for decommissioning automatically stops all running worker threads, pushes the affected tasks to the local task pool, and starts the migration of these tasks to other compute nodes (cf. Sect. 4.5).

Bound Synchronization. Pruning is based on a global upper bound. In case of the TSP, the total cost of the best tour currently known is used as the global upper bound. The distribution of the current upper bound is essential to avoid

excess computation (due to an outdated value). By employing the synchronization component (cf. Sect. 4.7), we initiate an update of the global upper bound whenever the local upper bound is better then the current global upper bound observed. Technically, we specify an update rule that compares the total costs of two tours. If a better upper bound has been detected, TASKWORK ensures that the new upper bound is propagated through the hierarchy levels of the parallel system. At the programming level, `getUpperBound` and `setUpperBound` (cf. Fig. 4) are implemented based on the `getVar` and `setVar` operations (cf. Sect. 4.7).

Termination Detection. Activating termination detection enables parallel applications to register for a termination event, which can be also used to retrieve the final result. In this case, the final result is a tour whose total cost is minimum and thus solves the TSP.

6 Experimental Evaluation

In this section, we present and discuss several experiments to evaluate TASKWORK. First, we describe our experimental setup. Second, we introduce the benchmark applications for parallel tree search that are used for the evaluation. Third, we report on the parallel performance and scalability by measuring speedups and efficiencies for both the TSP application implemented with the elastic branch-and-bound development framework and the described benchmark applications. Finally, we measure the effects of elastic scaling on the speedup of an application to assess the inherent overheads of dynamically adapting the number of compute nodes at runtime.

6.1 Experimental Setup

Compute nodes 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 private 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. Each compute node runs a single worker thread to process tasks and is connected to one of three ZooKeeper servers (forming a ZooKeeper cluster). Our experiments were performed during regular multi-tenant operation.

6.2 Benchmark Applications

Because work anomalies occur in the context of our branch-and-bound application, we additionally use two benchmarks for parallel tree search to rigorously evaluate TASKWORK. Work anomalies result from the search procedure being executed in parallel by different compute nodes on different subtrees of the search

tree. As a result, the amount of work significantly differs between sequential and parallel processing as well as across parallel application runs. We describe the benchmark applications in the following.

Unbalanced Tree Search (UTS). Unbalanced Tree Search [32] is a benchmark designed to evaluate task pool architectures for parallel tree search. UTS enables us to generate synthetic irregular workloads that are not affected by work anomalies and thus support a systematic experimental evaluation. Different tree shapes and sizes as well as imbalances can be constructed by means of a small set of parameters, where each tree node is represented by a 20-byte descriptor. This descriptor is used as a random variable based on which the number of children is determined at runtime. A child node's descriptor is generated by an SHA-1 hash function based on the parent descriptor and the child's index. As a result, the generation process is reproducible due to the determinism of the underlying hash function.

We generate UTS problem 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 [9,32,34]. 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. The problem instances employed for our measurements are UTS_1 ($r = 19$, $b = 4$, $d = 16$) and UTS_2 ($r = 19$, $b = 4$, $d = 17$).

WaitBenchmark. This benchmark was taken from [38], where it has been used in the context of parallel satisfiability (SAT) solving to systematically evaluate task pool architectures. The irregular nature of these applications is modeled by the benchmark as follows. To simulate the execution of a task, a processing unit has to wait T seconds. The computation is initialized with a single root task with a wait time T_{init}. At runtime, tasks can be dynamically generated by splitting an existing task. Splitting a task $Task_{parent}$ is done by subtracting a random fraction T_{child} of the remaining wait time T_R and generating a new task $Task_{child}$ with T_{child} as input:

$$Task_{parent}\{T_R\} \rightarrow (Task_{parent'}\{T_R - T_{child}\}, Task_{child}\{T_{child}\}). \quad (1)$$

6.3 Basic Parallel Performance

We report on the basic parallel performance of TASKWORK by measuring speedups and efficiencies for the TSP application implemented with the elastic branch-and-bound development framework. To evaluate the parallel performance, we solved 5 randomly generated instances of the 35 city symmetric TSP. Speedups and efficiencies are based on the execution time T_{seq} of a sequential implementation executed by a single thread on the same VM type. Table 1 shows the results of our measurements with three parallel program runs per TSP instance. As we can see, the measured performance is highly problem-specific.

Table 1. Performance measurements of TSP instances [28].

Problem instance	T_{seq} [s] (1 VM)	T_{par} [s] (60 VMs)	Speedup S [#] (60 VMs)	Efficiency E [%] (60 VMs)
$TSP35_1$	1195	32.9 ± 2.0	36.3	60.48
$TSP35_2$	1231	55.7 ± 4.0	22.1	36.87
$TSP35_3$	2483	103.5 ± 2.1	24.0	39.99
$TSP35_4$	3349	115.5 ± 6.3	29.0	48.31
$TSP35_5$	10286	167.4 ± 12.4	59.5	99.20

6.4 Scalability

To evaluate the scalability of a parallel system, one has to measure the standard metrics in parallel computing, i.e., the parallel execution time T_{par}, the speedup S, and the parallel efficiency E, for different numbers of processing units. $T_{par}(I,p)$ is the parallel execution time for a given input I measured with p processing units and $T_{seq}(I)$ is the sequential execution time for a given input I. The problem size $W(I)$ is defined as the number of (basic) computational steps in the best sequential algorithm to solve a problem described by I [14]. Under the assumption that it takes unit time to perform a single computational step, the problem size is equivalent to the sequential execution time $T_{seq}(I)$. To evaluate the scalability of TASKWORK, we report on two different measurement approaches: First, we measured the speedup with the UTS benchmark and the WaitBenchmark for a fixed problem size. Second, we measured the so-called *scaled speedup* with the WaitBenchmark. The scaled speedup of a parallel system is obtained by increasing the problem size linearly with the number of processing units [14]. We discuss the results in the following.

First, we report on the scalability of TASKWORK with a fixed problem size, which is thus independent of the number of processing units contributing to the computation. Figure 5 depicts the results of our measurements for three different problem instances: Two problem instances of the UTS benchmark UTS_1 ($r=19$, $b=4$, $d=16$) and UTS_2 ($r=19$, $b=4$, $d=17$) and a problem instance of the WaitBenchmark with a fixed initial wait time of the root task of $T_{init} = 600$ [s].

Second, we report on the scalability of TASKWORK with a problem size that is increased linearly with the number of processing units. In the case of the WaitBenchmark, the input is defined as the initial wait time of the root task T_{init}. The problem size can be defined as $W(T_{init}) = T_{init}$. Moreover, the sequential execution time $T_{seq}(T_{init})$ required by the best sequential algorithm to solve a problem described by T_{init} is $T_{seq}(T_{init}) = T_{init}$. This makes it easy to create a fixed problem size per processing unit, which requires us to increase the problem size W with the number of processing units p employed by the parallel system. For our measurements, we defined an initial wait time of the root task of $T_{init}(p) = p \cdot 60$ [s]. The speedups and efficiencies obtained are depicted in Fig. 6. The results of our measurement show close to linear speedups. A parallel

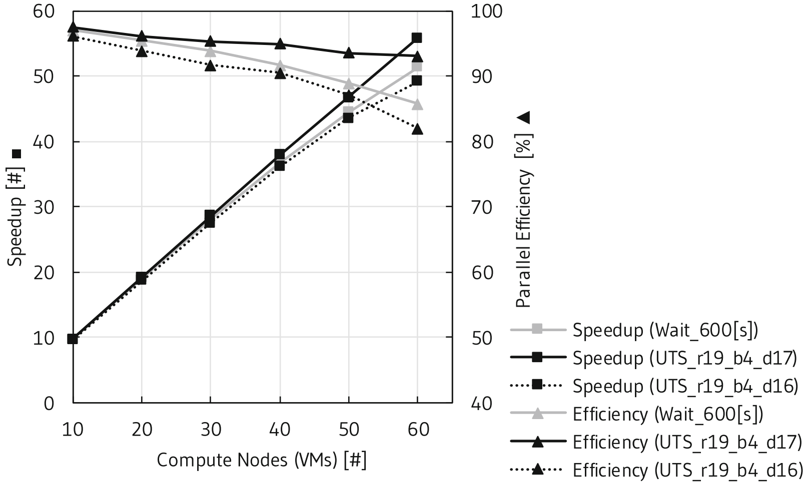

Fig. 5. The problem instances shown are UTS_1 ($r=19$, $b=4$, $d=16$), UTS_2 ($r=19$, $b=4$, $d=17$), and the WaitBenchmark with an initial wait time of the root task of $T_{init} = 600$ [s]. Speedups and efficiencies given are arithmetic means based on 3 parallel program runs for 6 setups leading to 54 measurements in total.

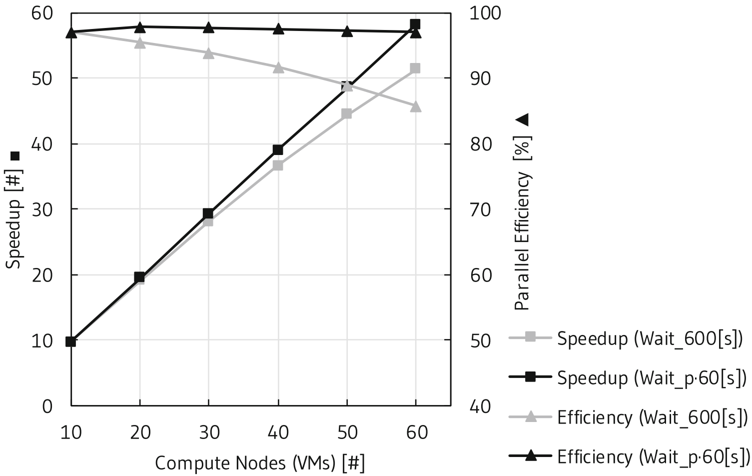

Fig. 6. The problem instances shown are the WaitBenchmark with an initial wait time of the root task of $T_{init}(p) = p \cdot 60$ [s] and the WaitBenchmark with an initial wait time of the root task of $T_{init} = 600$ [s]. Speedups and efficiencies given are arithmetic means based on 3 parallel program runs.

system is considered to be scalable when the scaled speedup curve is close to linear [14]. As expected, the (scaled) speedup curve is much better compared to the one obtained by our scalability measurements with $T_{init} = 600$ [s], which are also depicted in Fig. 6.

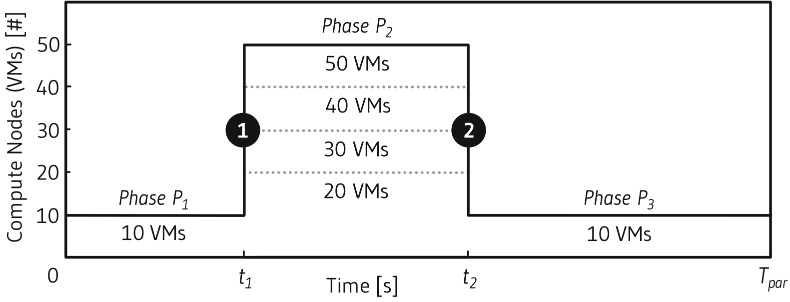

Fig. 7. We adapt the number of compute nodes (physical parallelism) at times t_1 and t_2 and measure the effects on the elastic speedup $S_{elastic}$ [28].

6.5 Elastic Scaling

In the cloud, compute resources can be provisioned or decommissioned at runtime by means of an elasticity controller. To make use of newly provisioned compute resources, the runtime system has to adapt to this change (cf. Sect. 3). A fundamental question that arises in this context is: *How fast* can resources be effectively employed by the application? This is a fundamentally new perspective on parallel system architectures that also has to be considered for evaluation purposes.

We propose a novel experimental method that shows the capability of a parallel system to dynamically adapt to a changing number of compute resources. Because parallel systems are designed with the ultimate goal to maximize parallel performance, our method evaluates the effects of dynamic resource adaptation on performance in terms of the speedup metric. Our experiment is described in Fig. 7 and comprises three phases. We start our application with 10 compute nodes (VMs) in Phase P_1. At time t_1, we scale out by adding more VMs to the computation. To evaluate the elastic behavior without platform-specific VM startup times, we employ VMs that are already running. At time t_2, we decommission the VMs added at t_1. At phase transition ❶, TASKWORK ensures task generation and efficient load balancing to exploit newly added compute nodes. At phase transition ❷, the task migration component ensures graceful decommissioning of compute nodes (cf. Sect. 4.5). We can easily see if newly added compute resources contribute to the computation by comparing the measured elastic speedup $S_{elastic}$ (speedup with elastic scaling) with the *baseline speedup* $S_{baseline}$ that we measured for a static setting with 10 VMs. To see how effectively new compute resources are employed by TASKWORK, we tested several durations for Phase P_2 as well as different numbers of VMs added (cf. Fig. 7) and calculated the percentage change in speedup S_{change} as follows:

$$S_{change} = \frac{(S_{elastic} - S_{baseline})}{S_{baseline}} \cdot 100 \tag{2}$$

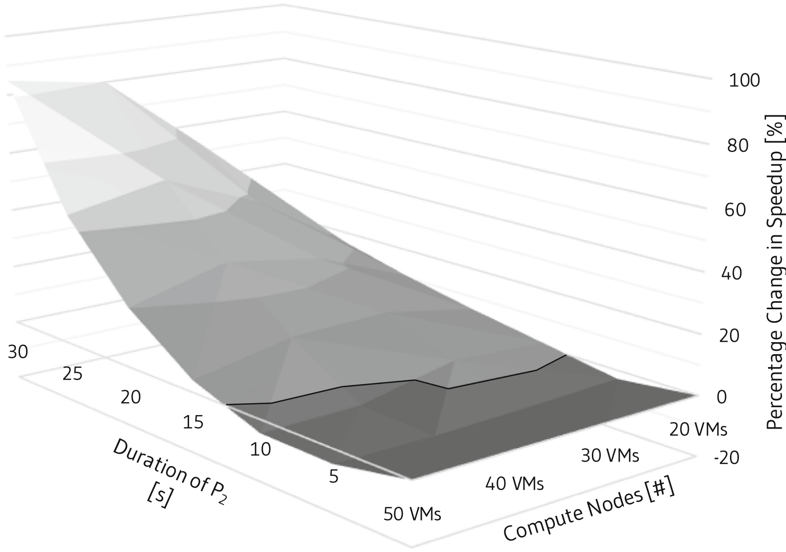

Fig. 8. The percentage change in speedup is calculated based on different durations of Phase P_2 and different numbers of compute nodes (VMs) added to the parallel computation at runtime. The number of VMs shown is the total number of VMs employed in Phase P_2.

S_{change} allows us to quantify the relative speedup improvements based on elastic scaling. Both $S_{elastic}$ and $S_{baseline}$ are arithmetic means calculated based on three program runs.

For our measurements, we employ the TSP application implemented based on the elastic branch-and-bound development framework. To avoid work anomalies, we disabled pruning to evaluate elastic scaling. All measurements are based on a TSP instance with 14 cities. The results of our measurements are depicted in Fig. 8, which shows the percentage change in speedup achieved for different durations of Phase P_2 and different numbers of compute nodes (VMs) added to the computation at runtime. 40 VMs added (leading to 50 VMs in total in Phase P_2) can be effectively employed in 15 s. Higher speedup improvements can be achieved by increasing the duration of Phase P_2. We also see that for a duration of only 10 s, adding 40 VMs even leads to a decrease in speedup whereas adding 20 VMs leads to an increase in speedup (for the same duration). This effect results from the higher overhead (in form of task generation, load balancing, and task migration) related to adding a higher number of VMs. On the other hand, as expected, for higher durations of Phase P_2, employing a higher number of VMs leads to better speedups. Note that the percentage of time spent in Phase P_2 (with respect to the total execution time) affects the actual percentage change in speedup, but not the effects that we have described.

7 Conclusion

In the presented work, we tackle the challenge of developing and operating elastic parallel tree search applications. We discuss related system-level challenges and show how to enable elastic parallel computations as well as cloud-aware coordination of distributed compute resources for the application class considered. Based on our findings, we present a novel runtime system that manages the low-level complexities related to elastic parallel applications to ease their development. Elastic parallel computations are enabled by means of load balancing, task migration, and application-specific task generation, which requires only minor effort at the programming level. Whereas the described development framework is specifically designed for elastic parallel branch-and-bound applications, other application classes that generate tasks at runtime (e.g., n-body simulations [18]) might also benefit from the design principles presented.

Many challenges are left on the path towards elastic parallel applications. Our long-term goal is to understand how to design, develop, and manage elastic parallel applications and systems. Therefore, we investigate design-level, programming-level, and system-level aspects [25,28] as well as delivery and deployment automation [22,23,26,29]. In this context, we are confident that TASKWORK provides a solid foundation for future research activities.

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

References

1. Anstreicher, K., Brixius, N., Goux, J.-P., Linderoth, J.: Solving large quadratic assignment problems on computational grids. Math. Program. **91**(3), 563–588 (2001). https://doi.org/10.1007/s101070100255
2. Archibald, B., Maier, P., McCreesh, C., Stewart, R., Trinder, P.: Replicable parallel branch and bound search. J. Parallel Distrib. Comput. **113**, 92–114 (2018)
3. Asanovic, K., et al.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
4. Blochinger, W., Dangelmayr, C., Schulz, S.: Aspect-oriented parallel discrete optimization on the cohesion desktop grid platform. In: Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006, CCGRID 2006, vol. 1, pp. 49–56, May 2006
5. Blochinger, W., Küchlin, W., Ludwig, C., Weber, A.: An object-oriented platform for distributed high-performance symbolic computation. Math. Comput. Simul. **49**, 161–178 (1999)
6. Blochinger, W., Michlin, W., Weber, A.: The distributed object-oriented threads system DOTS. In: Ferreira, A., Rolim, J., Simon, H., Teng, S.-H. (eds.) IRREGULAR 1998. LNCS, vol. 1457, pp. 206–217. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0018540
7. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)

8. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999)
9. Bungart, M., Fohry, C.: A malleable and fault-tolerant task pool framework for x10. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 749–757. IEEE (2017)
10. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 3rd edn. (2009)
11. Da Rosa Righi, R., Rodrigues, V.F., Da Costa, C.A., Galante, G., De Bona, L.C.E., Ferreto, T., et al.: AutoElastic: automatic resource elasticity for high performance applications in the cloud. IEEE Trans. Cloud Comput. **4**(1), 6–19 (2016)
12. Galante, G., De Bona, L.C.E., Mury, A.R., Schulze, B., Da Rosa Righi, R., et al.: An analysis of public clouds elasticity in the execution of scientific applications: a survey. J. Grid Comput. **14**(2), 193–216 (2016)
13. Gendron, B., Crainic, T.G.: Parallel branch-and-branch algorithms: survey and synthesis. Oper. Res. **42**(6), 1042–1066 (1994)
14. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Pearson Education, London (2003)
15. Gupta, A., et al.: Evaluating and improving the performance and scheduling of HPC applications in cloud. IEEE Trans. Cloud Comput. **4**(3), 307–321 (2016)
16. Gupta, A., et al.: 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, December 2013
17. Gupta, A., Sarood, O., Kale, L.V., Milojicic, D.: 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, May 2013
18. Hannak, H., Blochinger, W., Trieflinger, S.: A desktop grid enabled parallel Barnes-Hut algorithm. In: 2012 IEEE 31st International Performance Computing and Communications Conference, pp. 120–129 (2012)
19. Haussmann, J., Blochinger, W., Kuechlin, W.: Cost-efficient parallel processing of irregularly structured problems in cloud computing environments. Cluster Comput. **22**(3), 887–909 (2018). https://doi.org/10.1007/s10586-018-2879-3
20. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC 2010, p. 11, Berkeley (2010)
21. Junqueira, F., Reed, B.: ZooKeeper: Distributed Process Coordination. O'Reilly Media, Inc., Sebastopol (2013)
22. Kehrer, S., Blochinger, W.: AUTOGENIC: automated generation of self-configuring microservices. In: Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER), pp. 35–46. SciTePress (2018)
23. Kehrer, S., Blochinger, W.: TOSCA-based container orchestration on Mesos. Comput. Sci. Res. and Dev. **33**(3), 305–316 (2018)
24. Kehrer, S., Blochinger, W.: Elastic parallel systems for high performance cloud computing: state-of-the-art and future directions. Parallel Process. Lett. **29**(02), 1950006 (2019)
25. Kehrer, S., Blochinger, W.: Migrating parallel applications to the cloud: assessing cloud readiness based on parallel design decisions. SICS Softw. Intensive Cyber-Phys. Syst. **34**(2), 73–84 (2019)

26. Kehrer, S., Blochinger, W.: Model-based generation of self-adaptive cloud services. In: Muñoz, V.M., Ferguson, D., Helfert, M., Pahl, C. (eds.) CLOSER 2018. CCIS, vol. 1073, pp. 40–63. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29193-8_3
27. Kehrer, S., Blochinger, W.: A survey on cloud migration strategies for high performance computing. In: Proceedings of the 13th Advanced Summer School on Service-Oriented Computing. IBM Research Division (2019)
28. Kehrer, S., Blochinger, W.: TASKWORK: a cloud-aware runtime system for elastic task-parallel HPC applications. In: Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER), pp. 198–209. SciTePress (2019)
29. Kehrer, S., Riebandt, F., Blochinger, W.: Container-based module isolation for cloud services. In: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 177–186 (2019)
30. Mauch, V., Kunze, M., Hillenbrand, M.: High performance cloud computing. Future Gener. Comput. Syst. **29**(6), 1408–1416 (2013)
31. Netto, M.A.S., Calheiros, R.N., Rodrigues, E.R., Cunha, R.L.F., Buyya, R.: HPC cloud for scientific and business applications: taxonomy, vision, and research challenges. ACM Comput. Surv. (CSUR) **51**(1), 8:1–8:29 (2018)
32. Olivier, S., et al.: UTS: an unbalanced tree search benchmark. In: Almási, G., Caşcaval, C., Wu, P. (eds.) LCPC 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72521-3_18
33. Poldner, M., Kuchen, H.: Algorithmic skeletons for branch and bound. In: Filipe, J., Shishkov, B., Helfert, M. (eds.) ICSOFT 2006. CCIS, vol. 10, pp. 204–219. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70621-2_17
34. Posner, J., Fohry, C.: Hybrid work stealing of locality-flexible and cancelable tasks for the APGAS library. J. Supercomput. **74**(4), 1435–1448 (2018)
35. Prim, R.C.: Shortest connection networks and some generalizations. Bell Syst. Tech. J. **36**(6), 1389–1401 (1957)
36. Rajan, D., Thain, D.: Designing self-tuning split-map-merge applications for high cost-efficiency in the cloud. IEEE Trans. Cloud Comput. **5**(2), 303–316 (2017)
37. Rajan, D., Canino, A., Izaguirre, J.A., Thain, D.: Converting a high performance application to an elastic cloud application. In: IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom), pp. 383–390 (2011)
38. Schulz, S., Blochinger, W.: Parallel sat solving on peer-to-peer desktop grids. J. Grid Comput. **8**(3), 443–471 (2010)
39. Schulz, S., Blochinger, W., Held, M., Dangelmayr, C.: COHESION - a microkernel based desktop grid platform for irregular task-parallel applications. Future Gener. Comput. Syst. **24**(5), 354–370 (2008)
40. Sedgewick, R.: Algorithms. Addison-Wesley Publishing Co., Inc, Boston (1984)