Universität Stuttgart

Elastic Parallel Systems for High Performance Cloud Computing

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Stefan Kehrer

aus Tübingen

Hauptberichter: Prof. Dr. Wolfgang Blochinger

Mitberichter: Prof. Dr. Dr. h. c. Frank Leymann

Tag der mündlichen Prüfung: 05. August 2020

Institut für Architektur von Anwendungssystemen
der Universität Stuttgart

2020

Zusammenfassung

Hochleistungsrechnen (HPC) ermöglicht in vielen Bereichen von Wissenschaft und Wirtschaft signifikante Fortschritte. Neben der Erforschung fundamentaler Grand Challenges im klassischen Supercomputing werden parallele Anwendungen mittlerweile auch zur Beschleunigung von Berechnungen in Bereichen wie Produktdesign und -optimierung, Produktionsplanung, Finanzrisikomanagement, der medizinischen Diagnostik sowie in verschiedensten Forschungs- und Entwicklungsvorhaben eingesetzt. Gleichzeitig ist der Betrieb von Cluster Umgebungen, welche typischer Weise zur Ausführung paralleler Anwendungen benötigt werden, extrem kapitalintensiv, erfordert ein erhebliches Fachwissen und ist daher meist nur großen Organisationen vorbehalten, die von Skaleneffekten profitieren können.

Mit der Einführung des Cloud Computings steht eine alternative Ausführungsumgebung für parallele Anwendungen bereit, welche neue Charakteristika mit sich bringt: IT Ressourcen können nun bedarfsorientiert bezogen und nutzungsabhängig bezahlt werden, wodurch auch die elastische Skalierung einer Anwendung zur Laufzeit ermöglicht wird. Während Cloud Umgebungen bisher insbesondere für den Betrieb interaktiver Anwendungen herangezogen werden, interessieren sich auch HPC Nutzer für die Vorteile, die von Cloud Anbietern beworben werden: Unter anderem können Ressourcen entsprechend individueller Anforderungen konfiguriert werden,

bisherige Wartezeiten entfallen durch deren bedarfsorientierte Bereitstellung und die nutzungsabhängige Bezahlung vermeidet hohe Investitionsausgaben. Durch den Einsatz von Elastizität lassen sich die eingesetzten Ressourcen und damit auch die Ausführungszeit und Effizienz sowie die monetären Kosten eines Anwendungslaufs feingranular steuern. Während für das HPC optimierte Cloud Umgebungen bereits von Anbietern wie Amazon Web Services (AWS) und Microsoft Azure bereitgestellt werden, sind bestehende parallele Architekturen nicht dafür ausgelegt Elastizität zu nutzen.

Diese Arbeit adressiert Herausforderungen im entstehenden Forschungsfeld des High Performance Cloud Computings. Insbesondere beziehen sich die präsentierten Beiträge auf die neuartigen Möglichkeiten und Herausforderungen, die sich aus der Nutzung von Elastizität ergeben. Zuerst werden die Prinzipien elastischer paralleler Systeme sowie die damit zusammenhängenden Auswirkungen auf das Design paralleler Anwendungen diskutiert. Auf dieser Basis werden zwei exemplarische Architekturen elastischer paralleler Systeme präsentiert, die jeweils (1) einen Elastizitätssteuerungsmechanismus, der die Anzahl paralleler Verarbeitungseinheiten auf Basis benutzerdefinierter Ziele steuert, (2) ein Cloud-optimiertes Ausführungsmodell, welches die erforderliche Koordination und Synchronisation der Verarbeitungseinheiten automatisiert, und (3) ein Programmierkonzept zur vereinfachten Implementierung elastischer paralleler Anwendungen umfassen. Um die Auslieferung und das Deployment elastischer paralleler Anwendungen zu automatisieren, werden neue Konzepte zur Generierung der benötigten Artefakte auf Basis des zugehörigen Quellcodes vorgestellt, welche gleichzeitig anwendungsspezifische nicht-funktionale Eigenschaften berücksichtigen. Im Rahmen der Arbeit wird ein breites Spektrum an Umsetzungsmöglichkeiten zur Konstruktion elastischer paralleler Systeme diskutiert, das sowohl proaktive und reaktive Verfahren der Elastizitätssteuerung, als auch die Parallelverarbeitung mit Hilfe von virtuellen Maschinen (Infrastructure as a Service) und leichtgewichtigen Funktionen (Function as a Service) umfasst. Die beschriebenen Beiträge werden an Hand umfassender experimenteller Untersuchungen evaluiert.

Abstract

High Performance Computing (HPC) enables significant progress in both science and industry. Whereas traditionally parallel applications have been developed to address the grand challenges in science, as of today, they are also heavily used to speed up the time-to-result in the context of product design, production planning, financial risk management, medical diagnosis, as well as research and development efforts. However, purchasing and operating HPC clusters to run these applications requires huge capital expenditures as well as operational knowledge and thus is reserved to large organizations that benefit from economies of scale.

More recently, the cloud evolved into an alternative execution environment for parallel applications, which comes with novel characteristics such as on-demand access to compute resources, pay-per-use, and elasticity. Whereas the cloud has been mainly used to operate interactive multi-tier applications, HPC users are also interested in the benefits offered. These include full control of the resource configuration based on virtualization, fast setup times by using on-demand accessible compute resources, and eliminated upfront capital expenditures due to the pay-per-use billing model. Additionally, elasticity allows compute resources to be provisioned and decommissioned at runtime, which allows fine-grained control of an application's performance in terms of its execution time and efficiency as well as the related monetary

costs of the computation. Whereas HPC-optimized cloud environments have been introduced by cloud providers such as Amazon Web Services (AWS) and Microsoft Azure, existing parallel architectures are not designed to make use of elasticity.

This thesis addresses several challenges in the emergent field of High Performance Cloud Computing. In particular, the presented contributions focus on the novel opportunities and challenges related to elasticity. First, the principles of elastic parallel systems as well as related design considerations are discussed in detail. On this basis, two exemplary elastic parallel system architectures are presented, each of which includes (1) an elasticity controller that controls the number of processing units based on user-defined goals, (2) a cloud-aware parallel execution model that handles coordination and synchronization requirements in an automated manner, and (3) a programming abstraction to ease the implementation of elastic parallel applications. To automate application delivery and deployment, novel approaches are presented that generate the required deployment artifacts from developer-provided source code in an automated manner while considering application-specific non-functional requirements. Throughout this thesis, a broad spectrum of design decisions related to the construction of elastic parallel system architectures is discussed, including proactive and reactive elasticity control mechanisms as well as cloud-based parallel processing with virtual machines (Infrastructure as a Service) and functions (Function as a Service). To evaluate these contributions, extensive experimental evaluations are presented.

Contents

Chapter 1

Introduction

Since hitting the power wall, parallel processing has been considered the dominant paradigm to build future computing systems [ABD+09]. Parallel processing appears in many different forms, from bit-level to task-level parallelism, and affects many computing disciplines, from Embedded to High Performance Computing (HPC). The latter is considered in this thesis.

HPC makes use of parallel processing as the ultimate tool to solve very large problems in science and industry and to speed up their computation [ABD+09]. Applications stem from domains such as genomics, computational chemistry & physics, engineering, financial risk analysis, electronic design automation, machine learning, and many more. Typically, supercomputers and large compute clusters, operated in on-site data centers, have been employed to process highly compute-intensive problems in parallel. For researchers, making parallel programming easy remains the major challenge in this field due to the ongoing evolution of parallel computing systems and architectures.

More recently, the cloud evolved into an alternative execution environment for parallel applications [GEM+16; HNA17; NCR+18; VPB09]. Cloud environments come with attractive characteristics such as on-demand access

to compute resources, pay-per-use, and elasticity. Whereas the cloud has been mainly used to operate interactive multi-tier applications, HPC users are also interested in the benefits offered. These include full control of the resource configuration based on virtualization, fast setup times by using on-demand accessible compute resources, and eliminated upfront capital expenditures due to the pay-per-use billing model. Thus, by implementing the vision of utility computing [AFG+10; Par66], the cloud solves many problems that existed in traditional cluster environments for decades, e.g., limited freedom of configuration, long waiting times in job queues, and the strong financial background required to setup and maintain a large-scale cluster environment. Whereas using standard cloud offerings typically leads to a lower performance compared to traditional on-site clusters due to virtualization overhead and low throughput and high latency networks, during the last years, most of these issues have been addressed by introducing HPC-optimized cloud environments [GEM+16; NCR+18; YWW+14]. As of today, many cloud providers, including Amazon Web Services (AWS)[1] and Microsoft Azure[2], offer cloud environments optimized for HPC [AEJ18; ZLP17], which are also called *HPC-aware* [GFG+16; GKG+13]. An important milestone was reached in 2019 when a team of engineers from Descartes Labs led by Mike Warren built a cloud-based parallel system on top of AWS in an ad-hoc fashion that achieved rank 136 in the TOP500 list[3] [Des19].

Based on these developments, the HPC community raises the question: *How can parallel applications benefit from cloud-specific properties such as on-demand access to compute resources, pay-per-use, and elasticity?* This overarching research question motivates the contributions presented in this thesis, which address existing challenges related to the design, development, delivery & deployment, and management of cloud-aware parallel applications, i.e., applications that are optimized for cloud environments. Whereas HPC-aware cloud offerings enable a simple copy & paste migration of existing applications or complete cluster setups to the cloud, such an ap-

[1] https://aws.amazon.com.
[2] https://azure.microsoft.com.
[3] TOP500 list (June 2019): https://www.top500.org/lists/2019/06.

proach does not allow parallel applications to benefit from all cloud-specific properties. The contributions presented in this thesis specifically focus on the novel opportunities and challenges related to elasticity, which is often considered to be the most important cloud-specific property [APDM18; GB12]. Elasticity gives rise to a fundamentally new concept in HPC: The ability to control the physical parallelism of an application at runtime, which allows versatile optimizations. Therefore, an elasticity controller, which adapts the physical parallelism, has to understand the scaling behavior of a parallel system and related effects of adding and removing processing units on the execution time and the monetary costs of the computation, which have to be explicitly considered due to the pay-per-use billing model. Additionally, to enable elastic scaling, many challenges arise in the context of application development and parallel architectures, which have to support a dynamically changing number of processing units by providing appropriate coordination mechanisms. The research challenges that are addressed as well as the corresponding contributions are discussed in the following.

1.1 Research Challenges and Contributions

Several research challenges arise in the context of High Performance Cloud Computing. First, these research challenges are discussed in detail. Then, the research contributions addressing these challenges are summarized.

Research challenge 1: Handle design trade-offs to maximize parallel performance and to benefit from cloud-specific characteristics at the same time - From an architectural perspective, parallel systems (and applications) have been extensively researched and the design process is well understood [GGKK03]. Corresponding design knowledge is also captured in pattern languages for parallel computing that support the creative process of designing such systems. For instance, the OPL[1] pattern language [KMMS10; MMS07] supports pattern-based parallel design. However, to make use of elasticity, parallel systems are accompanied by an elasticity controller that adapts the number

[1] https://patterns.eecs.berkeley.edu.

of processing units. Dealing with a dynamically changing number of processing units leads to new requirements for traditionally designed parallel systems, which have to be considered during the design process [PARD13]. On the other hand, existing cloud-native design guidelines show how to design applications according to cloud-specific characteristics but do not consider parallel performance in terms of speedup and efficiency. Trade-off handling is required to consider parallel performance and the characteristics of cloud environments at the same time.

Research challenge 2: Ease the development of elastic parallel applications - Traditionally, programming models and development frameworks for parallel applications have been designed for a static number of processing units per application run. An elastic parallel application, on the other hand, has to be able (1) to make use of newly provisioned processing units and (2) to release existing processing units at runtime. This requirement leads to major implications, e.g., for task generation and mapping. Whereas the Message Passing Interface (MPI) version 2 (MPI-2) [GTL99], for instance, supports dynamic process management and thus a varying number of MPI processes, task generation and mapping have to be manually implemented, which is a difficult and error-prone task. To develop elastic parallel applications with only minor effort at the programming level, a new breed of programming models and development frameworks is required to hide these cloud-specific management operations from application developers. To abstract from low-level technical details, higher-level abstractions, which have to be optimized for specific parallel application classes, have to be investigated. In particular, novel runtime systems and frameworks are required that handle the system-level challenges related to elastic parallel processing, such as parallel execution and the coordination of distributed cloud resources, in a transparent manner.

Research challenge 3: Manage elastic parallel applications by considering execution time, efficiency, and monetary costs - The cloud provides metered resources on-demand, which have to be paid on a per-use basis. As a result, the monetary costs of computations have to be explicitly considered per application run. While for an (ideal) perfectly scalable parallel systems the

costs of computations are independent of the number of processing units employed, for a (realistic) non-perfectly scalable parallel system the costs typically increase with an increasing number of processing units due to overhead that increases with the number of processing units. In cloud environments, because one pays processing units per time unit, both using more processing units as well as using processing units for a longer period of time increases the monetary costs of a parallel computation. Consequently, for realistic parallel systems, it is not possible to achieve both a short execution time and low monetary costs of the computation at the same time. To handle this trade-off an elasticity controller is required, which adapts the number of processing units employed for the computation.

Research challenge 4: Ensure automated delivery and deployment for elastic parallel applications - Because cloud computing allows customers to build customized execution environments for their applications by means of self-service APIs, delivery and deployment automation approaches are widely used to shorten either the time-to-market or the time-to-result (e.g., to speed up experimentation cycles). To this end, DevOps and continuous delivery have been introduced, which aim at bridging the gap between development and operations by employing automation and self-service tools [HF10; HM11]. Furthermore, applications are typically assembled from existing components (such as libraries for numerical computations) by making use of private / public code repositories and open source software (OSS) to speed up their development. However, reused components might be unstable, resource-intensive, or untrustworthy. Thus, satisfying non-functional requirements such as reliability, efficiency, and security while ensuring rapid release cycles is a challenging task. Novel approaches are required to establish an *isolated context* for unstable, resource-intensive, or untrustworthy application components leading to improved security, fault containment, and / or fine-grained resource control while minimizing the overhead stemming from related isolation mechanisms. In this regard, existing automation approaches should be extended to consider and provide flexibility with respect to user-defined non-functional requirements.

In the following, the research contributions presented in this thesis are

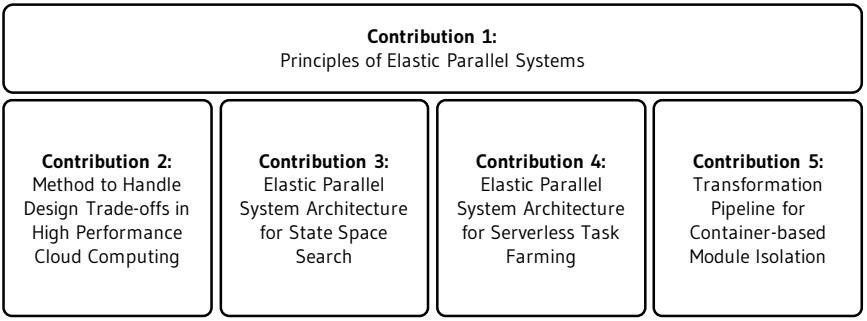

Figure 1.1: Overview of research contributions

summarized based on these research challenges. Figure 1.1 provides an overview of the research contributions.

Contribution 1 lays the foundation for addressing the aforementioned research challenges by formalizing the essential principles of elastic parallel systems. This is required to understand the opportunities and challenges related to elasticity. Existing parallel computing theory as well as traditional metrics to analytically evaluate parallel systems do not comprehensively consider elasticity, i.e., the ability to control the number of processing units at runtime. To address these issues, a conceptual framework is introduced to understand elasticity in the context of parallel systems, the term elastic parallel system is defined, and novel metrics for both elasticity control as well as the ex-post performance evaluation of elastic parallel systems are discussed.

Contribution 2 addresses *research challenge 1* by providing a method to handle existing design trade-offs in High Performance Cloud Computing. A novel approach to assess the cloud readiness of parallel applications based on the design decisions made is presented. By discovering and understanding the implications of these parallel design decisions on an application's cloud readiness, this approach supports the migration of parallel applications to the cloud. Among an assessment procedure and its underlying meta model, a corresponding instantiation is provided to structure this multi-dimensional

design space. For evaluation, an extensive case study comprising three variants of a parallel application is discussed.

Contribution 3 considers *research challenge 2* and *3* in the context of state space search. Well-known meta-algorithms based on this technique include branch-and-bound and backtracking search, which are typically employed in domains such as biochemistry, electronic design automation, production planning and scheduling, finite geometry, and many more. Based on an in-depth analysis of the characteristics of these applications, the design and implementation of a cloud-aware runtime system, which enables the implementation of elastic parallel applications by means of higher-level development frameworks, is presented. While domain experts are able to implement applications based on these frameworks, a reactive elasticity controller ensures automated management by provisioning and decommissioning metered compute resources at runtime according to user-defined cost and efficiency thresholds. Among an exemplary development framework for elastic parallel branch-and-bound, an extensive experimental evaluation is provided that also considers several benchmarks and novel experiments explicitly designed to evaluate elastic parallel systems.

Contribution 4 enables serverless parallel processing by providing a development and runtime framework specifically designed to free developers from both parallelism and resource management issues. *Research challenge 2* and *3* are addressed by presenting a novel approach to parallel cloud programming called serverless skeletons, which integrates a proactive elasticity controller that is able to adapt the physical parallelism per application run according to a user-defined goal, e.g., a specific execution time limit after which the result of the computation needs to be present. To evaluate the described concepts, a serverless farm skeleton is introduced, which supports the implementation of a wide range of applications. Two applications have been implemented based on a prototypical development and runtime framework: Numerical integration and hyperparameter optimization - a commonly applied technique in machine learning. Performance measurements are presented for both applications. Additionally, to address *research challenge 4*, the framework supports automated delivery and deployment of applications

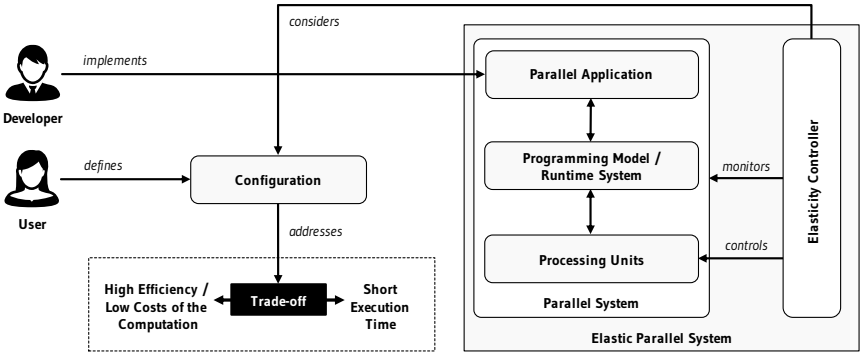

Figure 1.2: Scenario supported by the contributions of this thesis

and isolates individual application components according to a user-defined configuration.

Contribution 5 tackles *research challenge 4* by providing a transformation pipeline for container-based module isolation. By applying modular software development, applications can be rapidly constructed by composing existing, newly developed, and publicly available third-party modules. To satisfy non-functional requirements such as reliability, efficiency, and security, container virtualization is employed to isolate modules from each other according to a specification of isolation constraints. These non-functional requirements are satisfied by automatically transforming the modules comprised into a container-based system. To deal with the increased overhead that is caused by isolating modules from each other, the minimum set of containers required to satisfy the isolation constraints specified is calculated. Moreover, a prototypical transformation pipeline is presented that automatically transforms applications developed based on the Java Platform Module System (JPMS) into container-based systems. Container-based module isolation offers an alternative to service-level isolation - specifically designed for applications with high performance requirements.

By addressing all the aforementioned challenges, the contributions presented in the remainder of this thesis enable the scenario depicted in Fig-

ure 1.2. Therefore, design trade-offs are discussed in detail, programming abstractions are provided to ease the development of elastic parallel applications, and management is ensured by an elasticity controller that automatically controls the performance in terms of execution time and efficiency as well as the related monetary costs of the computation according to a user-defined configuration.

1.2 Scientific Publications

The contributions presented in this thesis are based on the following peer-reviewed international conference papers, peer-reviewed journal articles, and book chapters:

1. [KZSB20] S. Kehrer, D. Zietlow, J. Scheffold, W. Blochinger. "Self-tuning Serverless Task Farming using Proactive Elasticity Control." In: *Cluster Computing (in press)* (2020)
2. [KB20b] S. Kehrer, W. Blochinger. "Equilibrium: An Elasticity Controller for Parallel Tree Search in the Cloud." In: *The Journal of Supercomputing (in press)* (2020)
3. [KB20a] S. Kehrer, W. Blochinger. "Development and Operation of Elastic Parallel Tree Search Applications using TASKWORK." In: *Cloud Computing and Services Science*. Ed. by D. Ferguson, V. M. Muñoz, C. Pahl, M. Helfert. Springer, 2020, pp. 42–65
4. [KSB19] S. Kehrer, J. Scheffold, W. Blochinger. "Serverless Skeletons for Elastic Parallel Processing." In: *2019 IEEE 5th International Conference on Big Data Intelligence and Computing (DATACOM)*. IEEE, 2019, pp. 185–192
5. [KB19a] S. Kehrer, W. Blochinger. "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, pp. 57–69

6. [KB19b] S. Kehrer, W. Blochinger. “Elastic Parallel Systems for High Performance Cloud Computing: State-of-the-Art and Future Directions.” In: *Parallel Processing Letters* 29.02 (2019), pp. 1950006-1-1950006-20

7. [KB19d] S. Kehrer, W. Blochinger. “Model-Based Generation of Self-adaptive Cloud Services.” In: *Cloud Computing and Services Science*. Ed. by V. M. Muñoz, D. Ferguson, M. Helfert, C. Pahl. Springer, 2019, pp. 40–63

8. [KB19e] S. Kehrer, W. Blochinger. “TASKWORK: A Cloud-aware Runtime System for Elastic Task-parallel HPC Applications.” In: *Proceedings of the 9th International Conference on Cloud Computing and Services Science*. SciTePress, 2019, pp. 198–209

9. [KRB19] S. Kehrer, F. Riebandt, W. Blochinger. “Container-Based Module Isolation for Cloud Services.” In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 177–186

10. [KB19c] S. Kehrer, W. Blochinger. “Migrating Parallel Applications to the Cloud: Assessing Cloud Readiness based on Parallel Design Decisions.” In: *SICS Software-Intensive Cyber-Physical Systems* 34.2 (2019)

11. [KB18a] S. Kehrer, W. Blochinger. “AUTOGENIC: Automated Generation of Self-configuring Microservices.” In: *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. SciTePress, 2018, pp. 35–46

12. [KB18b] S. Kehrer, W. Blochinger. “TOSCA-based container orchestration on Mesos.” In: *Computer Science - Research and Development* 33.3 (2018), pp. 305–316

Additionally, further peer-reviewed publications of the author contributed to this work:

1. [KJZ16a] S. Kehrer, D. Jugel, A. Zimmermann. “Categorizing Requirements for Enterprise Architecture Management in Big Data Literature.” In: *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*. IEEE, 2016, pp. 1–8
2. [KJZ16b] S. Kehrer, D. Jugel, A. Zimmermann. “A systematic literature review of big data literature for EA evolution.” In: *Digital Enterprise Computing*. Ed. by D. Hertweck, C. Decker. Gesellschaft für Informatik, 2016, pp. 209–220
3. [JKSZ15b] D. Jugel, S. Kehrer, C. M. Schweda, A. Zimmermann. “Providing EA decision support for stakeholders by automated analyses.” In: *Digital Enterprise Computing*. Ed. by A. Zimmermann, A. Rossmann. Gesellschaft für Informatik, 2015, pp. 151–162
4. [HK15] T. Hamm, S. Kehrer. “Goal-oriented Decision Support in Collaborative Enterprise Architecture.” In: *Digital Enterprise Computing*. Ed. by A. Zimmermann, A. Rossmann. Gesellschaft für Informatik, 2015, pp. 175–182
5. [JKSZ15a] D. Jugel, S. Kehrer, C. M. Schweda, A. Zimmermann. “A decision making case for collaborative enterprise architecture engineering.” In: *INFORMATIK 2015*. Ed. by D.W. Cunningham, P. Hofstedt, K. Meer, I. Schmitt. Gesellschaft für Informatik, 2015, pp. 865–879

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows. First, Chapter 2 presents the fundamentals required to understand the contributions described in the following chapters. The principles of elastic parallel systems (contribution 1) are described in Chapter 3. Chapter 4 covers contribution 2 by providing a method to handle trade-offs that arise in the context of designing cloud-aware parallel applications. The design and implementation of a runtime system for elastic parallel state space search as well as a corresponding elasticity controller (contribution 3) are presented in Chapter 5. In Chapter 6, a

development and runtime framework for elastic parallel processing based on serverless computing platforms (contribution 4) is presented. Chapter 7 covers contribution 5 by describing a transformation pipeline for container-based module isolation. Finally, Chapter 8 concludes this thesis by summarizing the presented contributions and outlining future research opportunities that arise from them.

Chapter 2

Fundamentals and State-of-the-Art

This chapter lays the ground for understanding the contributions presented in this thesis. To this end, the state-of-the art is discussed based on related work. Section 2.1 provides an overview of parallel computing and High Performance Computing, describes existing approaches to parallel programming, and presents metrics for the analytical evaluation of parallel systems. Section 2.2 describes the cloud computing paradigm, serverless computing, as well as existing approaches for delivery and deployment automation. Related work in the emergent discipline of High Performance Cloud Computing is analyzed in Section 2.3.

2.1 Parallel and High Performance Computing

In the past, one could rely on increasing processor clock speeds, which increase the performance of an application without any code changes. However, with the end of frequency scaling, parallel computing has become the domi-

nant paradigm in computer architecture [ABD+09]. To make use of parallel hardware, a single problem has to be split into independent subproblems that can be solved simultaneously by multiple processing units. A major challenge of developing parallel applications is the inherent complexity that results from the need for task decomposition and mapping as well as synchronization and communication across a set of (potentially distributed) processing units. To ensure high performance, application developers have to carefully consider and balance different, conflicting sources of overhead, namely processor idling, communication, and excess computation [GGKK03].

High Performance Computing is concerned with the aggregation of compute resources that can be employed to solve complex problems in science and industry. Parallel processing is employed to speed up such computations. A major challenge in this field is the development of efficient and scalable parallel applications, which are typically operated on large compute clusters. Organizations that need to run parallel applications require access to such an environment or have to operate a compute cluster on their own. Sharing of cluster environments is accomplished by so-called cluster schedulers, which optimize resource utilization by planning the job schedule [UCL04; VD03]. As a consequence, HPC users often have to wait for their results due to long job scheduling times. Additionally, cluster schedulers typically enforce a fixed time limit upon which an application is terminated even if the computation has not been completed [PJ18].

Since 1993, the world's fastest supercomputers are regularly determined by the TOP500 project[1], which ranks supercomputers according to their LINPACK benchmark[2] performance measured in floating point operations per second (FLOPS). The label *supercomputer* is used to refer to very fast computing systems, but is not limited to a specific architecture [SMDS15]. Throughout the years, many different architectures have been considered to build supercomputers such as vector processing or massively parallel processing (MPP) systems. 90.6% of the world's largest supercomputers

[1] https://www.top500.org.
[2] http://www.netlib.org/benchmark/hpl.

ranked in the TOP500 list (June 2019)[1] are based on a cluster architecture. 9.4% are classified as MPP systems. As opposed to clusters, which are defined as integrated collections of independent nodes, massively parallel processing systems employ tightly-integrated processors, highly-optimized memory architectures, and high speed (most often proprietary) interconnect technology [DSSS05].

2.1.1 Parallel Programming

Parallel programming models (programming languages, development frameworks) have been researched extensively as a means to separate software development concerns from parallel execution concerns. By means of abstraction, implementation decisions related to parallel execution (such as communication, synchronization, or coordination) have to be made only once and not individually for each application [ST98]. As a result, applications based on higher-level abstractions can be developed faster and low-level errors related to parallel execution (e.g., deadlocks, starvation, and race conditions) are reduced. Different classification schemes have been described in this context [BST89; Fos00; ST98]. In the following, an exemplary low-level and an exemplary high-level programming abstraction are outlined.

The Message Passing Interface (MPI) [GLS14; GTL99] is an exemplary low-level programming abstraction that requires a developer to consider all aspects of parallel processing explicitly in an application's source code. The MPI standard defines the syntax and semantics of a message passing library, which can be implemented differently and thus optimized for specific types of parallel architectures. Parallel applications based on MPI can be developed as a set of communicating processes by following the Multiple Program Multiple Data (MPMD) programming model [Fos95]. However, also the Single Program Multiple Data (SPMD) technique is often used [KMMS10; MMS07]. SPMD allows the definition of a single program that is executed by each MPI process. At specific regions different branches of instructions are defined and executed only by a subset of the processes.

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

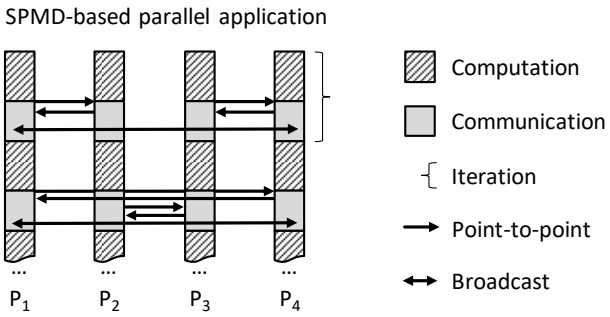

Figure 2.1: Many parallel applications are developed based on the Single Program Multiple Data (SPMD) model and rely on synchronous communication in globally defined communication phases.

These branches are typically distinguished by the process id (called rank), which is assigned to each process. Typically, a static number of processes is created at the beginning of the computation. A prototypical SPMD-based MPI application is shown in Figure 2.1. In each iteration, every MPI process executes local computations. Thereafter, a pair of MPI processes exchanges application-specific data via point-to-point communication. After all process pairs finished their data transfers, updates required globally are sent to other processes by means of an MPI broadcast primitive (cf. Figure 2.1).

An example of a higher-level programming abstraction are algorithmic skeletons [Col91; GC11], which structure parallel programs as a set of higher-order functions that abstract over common patterns of parallel coordination. Because parallel coordination is captured by the skeleton, developers are able to implement functional code without considering parallelism issues. A major benefit of using skeletons is that coordination is transparently handled, which substantially reduces runtime errors (e.g., due to deadlocks, starvation, and race conditions) when compared to low-level parallel programming models. Consequently, one can say that each skeleton comprises a built-in parallel behavior [DFH+93].

Algorithmic skeletons can be classified as either *task-parallel* with examples such as pipeline, farm, divide & conquer, and branch-and-bound or

data-parallel such as map and fold [DFH+93; DPP97; Kuc19]. Over the years, many skeleton frameworks and libraries have been developed for a variety of programming languages and parallel execution environments [ADT03; AMP14; BDO+95; GL10]. Whereas functional code is implemented by developers, provided compiling tools take care of automatically generating code for parallel execution to ease programming. Depending on the execution environment considered, parallel execution is based on POSIX threads [But97], OpenMP [CDK+01], MPI [GLS14; GTL99], OpenCL[1], or CUDA[2]. For instance, Muesli [KS05; Kuc19] is a C++ template library that supports parallel execution on top of MPI, OpenMP, and CUDA. During the last years, skeleton-based programming models have been widely adopted. Probably the most prominent example is MapReduce [DG08], which has been designed to simplify data processing on large, heterogeneous clusters of commodity hardware and hides data distribution and parallel execution.

2.1.2 Metrics and Analytical Evaluation

Parallel systems are employed to solve large problems described by a problem description or input I. One can distinguish between the input size I_{size}, which simply quantifies the size of a given input I, and the problem size W, which is defined as the number of basic computational steps that are required by the best sequential algorithm to solve the problem described by I on a single processing unit. The sequential execution time T_{seq} is defined as the wall-clock time required by the best sequential algorithm to solve the problem on a single processing unit. Under the assumption that it takes unit time to perform a single computational step, the problem size W of a problem described by I is equal to the sequential execution time T_{seq} [GGKK03]:

$$W(I) = T_{seq}(I) \tag{2.1}$$

[1] https://www.khronos.org/opencl.
[2] https://developer.nvidia.com/cuda-zone.

Whereas the execution time of sequential applications only depends on their input, the execution time of parallel applications depends not only on the input but also on the number of processing units that contribute to the computation as well as their computation and communication speeds [GGKK03]. Consequently, parallel applications cannot be evaluated in isolation. Rather, a parallel system is evaluated as a whole, i.e., a combination of a parallel application and a parallel architecture employed to process a specific problem.

The most commonly used performance metrics to evaluate parallel systems are parallel execution time, speedup, and parallel efficiency. These are defined as follows: Parallel execution time T_{par} is the time elapsed between the beginning and the end of a parallel computation. Speedup S quantifies the relative benefit (in terms of time) of solving a problem in parallel and is defined as the ratio of the sequential execution time T_{seq} to the parallel execution time T_{par}.

$$S(I,p) = \frac{T_{seq}(I)}{T_{par}(I,p)} \tag{2.2}$$

Parallel efficiency E quantifies the fraction of time for which the processing units perform useful work.

$$E(I,p) = \frac{S(I,p)}{p} = \frac{T_{seq}(I)}{T_{par}(I,p) \cdot p} \tag{2.3}$$

Parallel overhead O_{par} is expressed as the total time collectively spent by all processing units over and above the sequential execution time T_{seq} and can be modeled as follows [GGKK03]:

$$O_{par}(I,p) = p \cdot T_{par}(I,p) - T_{seq}(I) \tag{2.4}$$

2.2 Cloud Computing

At least since 2006, when Amazon Web Services (AWS) started to offer compute resources as a service to customers, cloud computing [AFG+10; Hay08; Ley09; LKN+09; MG+11] is receiving considerable attention from both researchers and practitioners. In the meantime, many other companies entered the market as cloud providers and cloud computing has been established as an economically viable implementation of the utility computing vision [AFG+10; Par66]: For instance, AWS reports on a net income which increased from $3.0 billion in 2017 to $10.1 billion in 2018 [Ama19].

Similar to the foundry model, which revolutionized the hardware industry by separating semiconductor fabrication from integrated circuit design, cloud computing separates the operation and usage of compute resources based on well-defined interfaces between cloud providers and customers [AFG+10]. Cloud providers benefit from economies of scale by buying and maintaining very large infrastructures, while customers benefit from on-demand accessible resources and a pay-per-use billing model, which substantially reduces financial risks [ZCB10]. The authors of [MG+11] define five essential characteristics of cloud computing, namely (1) on-demand self-service, (2) broad network access, (3) resource pooling, (4) rapid elasticity, as well as (5) measured service, three service models, namely (1) Software as a Service (SaaS), (2) Platform as a Service (PaaS), as well as (3) Infrastructure as a Service (IaaS), and four deployment models, namely (1) private cloud, (2) community cloud, (3) public cloud, and (4) hybrid cloud. Technically, cloud computing is based on recent achievements in virtualization technology, which enable isolation of customer workloads and resource pooling, i.e., sharing of compute resources across customers.

Based on on-demand accessible compute resources and pay-per-use, elasticity has been recognized as the most important cloud-specific property because it enables applications to scale dynamically as their workload increases / decreases at runtime [APDM18; GB12]. Since then, elasticity has been mainly employed in the context of interactive (multi-tier) applications that are designed to process (independent) user requests [BHS+19;

FLR+14; HKR13; MCTD13]. As the arrival rate of user requests increases over time, more compute resources can be provisioned to satisfy quality of service (QoS) requirements. Similarly, compute resources are decommissioned if the arrival rate decreases. The entity that controls the number of compute resources, e.g., virtual machines, is called *elasticity controller*. Elasticity control can be implemented as a manual or automated process, whereas automation is typically preferred. Automated elasticity control approaches derive scaling actions either by means of reactive or proactive mechanisms [GB12]. Reactive mechanisms use specified conditions (e.g., thresholds on monitored metrics) to control elasticity. Proactive mechanisms rely on forecasting to adapt a system according to its predicted behavior in the future. Typically, it depends on the characteristics of the application if reactive or proactive mechanisms yield better results. Both approaches might also be combined into a hybrid approach. Beyond interactive multi-tier applications, elasticity has also been researched in the context of workflows [CB14; MH11] and stream processing [RM19]. Detailed analyses of existing research on elasticity as well as classifications of elasticity mechanisms are presented in [APDM18; GB12; JS15; LML14; QCB18].

From a parallel computing perspective, the idea of adapting the number of compute resources employed by an application 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 [Fei96]. 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 [UCL04; VD03]. In cloud computing, on the other hand, the cloud customer controls the number of processing units by means of an application-specific elasticity controller.

Due to on-demand access to compute resources via self-service APIs and elasticity in cloud environments, a high degree of automation is required. Au-

tomated delivery and deployment approaches are discussed in Section 2.2.2. Moreover, to ensure automated management, cloud computing also evolved into an application domain for concepts from disciplines such as autonomic computing [KC03; PH05; ST05] and self-aware computing [KLB+17]. For instance, self-manageable cloud services are discussed in [Bra09]. In [KKB14], autonomic computing principles are applied to avoid service-level agreement (SLA) violations. The authors of [TBB+15; TBB+17] propose an architecture for self-managing cloud-native applications.

2.2.1 Serverless Computing

Serverless computing can be seen as a natural evolution of existing cloud service models and is heavily influenced by microservices, container virtualization, and event-driven programming [KY17; vTT+18]. Whereas the microservices architectural style [New15; STH18; Zim17] propagates the development and operation of fine-grained services, container virtualization helped to back this trend from a technological side. Following these developments, serverless computing enables function-level elasticity by decoupling compute from storage, which is a common approach to build cloud-native applications [CDE+13; VGS+17]. The compute tier is represented by stateless FaaS (Function as a Service) functions[1] and the storage tier is given by backend services (Backend as a Service) such as databases, message queues, and caching systems [JSS+19]. Because FaaS functions themselves are not individually addressable (point-to-point communication is not supported), they can only communicate via shared backend services [HFG+19].

Typically, each FaaS function is executed in an event-driven manner, i.e., it is triggered when a specific event occurs or a user request is received. FaaS functions can also be invoked via platform-specific APIs. Technically, user code is executed in a sandboxed environment (such as a container) with a specific amount of compute resources (CPU, memory). State-of-the-art

[1] A function in the serverless computing context is called FaaS function in the following not to be confused with programming-level functions.

serverless computing platforms include AWS Lambda[1], Azure Functions[2], Google Cloud Functions[3], Alibaba Cloud Function Compute[4], and IBM Cloud Functions[5]. A detailed comparison can be found in [WLZ+18]. Moreover, open source platforms are available to be operated in a private cloud setting or on top of IaaS cloud offerings. Prominent examples are Apache OpenWhisk[6], Fission[7], Fn Project[8], and Kubeless[9].

2.2.2 Delivery and Deployment Automation

Even though fast software delivery and deployment have always been major concerns in industry and science, either to shorten the time-to-market or the time-to-result, the need for automated solutions became prevalent with the emergence of the cloud computing paradigm. Cloud environments allow customers to build customized execution environments for their applications in an automated manner, e.g., by means of deployment scripts that call self-service APIs or PaaS solutions offered by a cloud provider [EBF+17; WALS18]. In this regard, novel concepts, methods, and tools evolved to replace manual and typically error-prone processes related to the delivery and deployment of cloud applications.

Continuous delivery [Che15; HF10; HM11] is an emerging paradigm to streamline the delivery of software artifacts, which heavily relies on automation to enable fast feedback cycles. A so-called continuous delivery pipeline (or deployment pipeline) [HF10] models and automates all steps from development to operation including source code retrieval from private / public code repositories, compilation, software testing, and deployment. Such pipelines also allow for consideration of user-defined configurations,

[1] https://aws.amazon.com/lambda.
[2] https://azure.microsoft.com/en-us/services/functions.
[3] https://cloud.google.com/functions.
[4] https://www.alibabacloud.com/products/function-compute.
[5] https://www.ibm.com/cloud/functions.
[6] https://openwhisk.apache.org.
[7] https://fission.io.
[8] https://fnproject.io.
[9] https://kubeless.io.

e.g., to control non-functional aspects of an application. Continuous delivery does not necessarily include automated deployment [LMP+15], e.g., if the user requires full control on when to run the application. However, the deployment step itself is most often also automated.

A wide range of deployment automation approaches have been proposed. They might be provider-specific (e.g., CloudFormation[1]), based on existing tools (e.g., Chef[2] or Puppet[3]), or based on standards such as the Topology and Orchestration Specification for Cloud Applications (TOSCA) [OAS13], which defines a modeling language for portable cloud services. An overview of existing deployment approaches is given in [WALS18]. Deployment systems typically process so-called deployment models (or deployment templates), which describe an application's components and their relationships. A deployment model might be imperative, i.e., the model describes all the steps required to deploy the application explicitly, or declarative, i.e., the model only describes the desired outcome while the deployment system automatically derives the required steps [EBF+17]. Many different modeling languages have been proposed in literature. A systematic review of cloud modeling languages is presented in [BBF+18].

More recently, containers have been used to package, deliver, and deploy applications [BGO+16; KQ17; PHP16; Tur14]. Container virtualization establishes a sandboxed execution environment for a group of processes (typically representing an application) on the operating system level. A host can execute multiple containers independently. Isolation is enforced by operating system mechanisms like control groups and namespaces [Tur14]. While machine-level virtualization provides a higher degree of isolation, container virtualization requires significantly less resources (especially memory) such that the number of containers a machine can host is considerably higher compared to the number of possible virtual machines. Docker[4], an open-source implementation of container virtualization, gained momentum

[1] https://aws.amazon.com/cloudformation.
[2] https://www.chef.io.
[3] https://puppet.com.
[4] https://www.docker.com.

during the last years [Tur14]. An application can be described using a *Dockerfile* representing the build configuration. The Docker build process creates a *Docker image*, which defines a self-contained artifact to deploy a *Docker container* running one or more application or middleware components. The resulting image captures an ordered collection of filesystem changes (layers) and a runtime configuration for the container. Thus, the context of an application or middleware component in a container is the same in both development and production environments [BO16]. State-of-the-art container runtime environments such as Kubernetes[1] support the deployment of container-based applications. Therefore, most often declarative deployment models are used, which reference the required container images.

2.3 High Performance Cloud Computing

With the emergence of cloud offerings, researchers from the HPC domain executed existing applications and benchmarks in cloud environments to gain insights into their performance characteristics and benefits [EH08; JRM+10; ZLZ+11]. Since then, the discipline of High Performance Cloud Computing investigates how to operate parallel applications in cloud environments [NCR+18].

Early studies mainly focus on the overhead of virtualization (CPU time-sharing and memory overcommitment) as well as heterogeneous network bandwidth and latencies. It has been recognized that the characteristics of cloud environments, such as resource pooling and virtualization, result in major drawbacks specifically for large-scale, tightly-coupled applications [GM11]. Based on these findings, basically two research approaches emerged [GFG+16]: (1) Research efforts that investigate how to make cloud environments HPC-aware and (2) research efforts that investigate how to make parallel applications cloud-aware. Research efforts from the first category address the recognized gap between cloud environments and parallel applications by proposing HPC-aware adaptations of cloud environments, whereas

[1] https://kubernetes.io.

research efforts from the second category analyze how applications can be adapted to benefit from cloud-specific characteristics. Whereas both research streams follow different goals, they complement each other. For instance, the benefits of HPC-aware cloud environments can also be leveraged by a cloud-aware parallel application that makes use of elastic scaling. The contributions presented in this thesis belong to the second category.

2.3.1 HPC-aware Cloud Environments

Whereas existing parallel applications operated in standard cloud environments often suffer from heterogeneous processing speeds as well as low network throughput and high network latency, which are well-known effects of resource pooling and virtualization [GEM+16; YWW+14], HPC-aware cloud environments limit these negative side-effects by means of the following concepts:

- CPU affinity: HPC-aware cloud environments ensure CPU affinity both at the application and at the hypervisor level. As a result, vCPUs are mapped to physical CPU cores leading to improved cache locality and higher cache hit rates [GKG+13]. This concept is also referred to as CPU pinning.
- HPC-aware virtual machine (VM) placement policies: Standard cloud schedulers do not ensure co-location of VMs. Existing work has shown that HPC-aware VM placement effectively resolves this issue and leads to significant performance gains [GKM+13].
- Guaranteed network performance: HPC-aware cloud environments are based on novel concepts such as single root input/output virtualization (SR-IOV) to ensure guaranteed network bandwidth and latency with constant quality of service while supporting resource pooling and network virtualization [MKH13; ZLP17].
- Disabled VM migration: Typically, VM migration is disabled to avoid environmental overhead.

- Disabled memory overcommitment: Memory overcommitment leads to preemption and paging and is typically disabled [FBB08].
- Lightweight virtualization: Container-based virtualization (OS-level virtualization) ensures lower overheads compared to hypervisor-based virtualization [ZLP16].

With the introduction of HPC-aware cloud environments, more parallel application classes can benefit from on-demand access to compute resources and a pay-per-use billing model. Specifically, for tightly-coupled SPMD-based parallel applications, HPC-aware cloud offerings provide an execution environment that can be used analogously to an HPC cluster but allows individual configuration of compute resources. Moreover, elasticity can be employed. However, note that HPC-aware cloud environments are typically more expensive compared to standard cloud environments. This mainly results from the specialized hardware employed and the limited opportunities of the cloud provider to operate multiple, diverse customer workloads across the physical infrastructure (multitenancy) [GFG+16; GKG+13]. Moreover, VM placement policies limit the provider's VM placement options.

2.3.2 Cloud-aware Parallel Applications

In this section, existing approaches to make parallel applications cloud-aware are discussed. Basically, two different categories have been identified: (1) approaches that only discuss how parallel applications have to be adapted to limit the negative side-effects of cloud environments such as heterogeneous processing speeds and varying network latencies and (2) approaches that investigate on how to make use of elastic scaling. Table 2.1 shows a classification of related work discussed in the following.

Several benchmarks and parallel applications executed in traditional cluster and cloud environments are compared with respect to their performance by the authors of [GFG+16]. Additionally, 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-

Table 2.1: Classification of existing work on cloud-aware parallel applications

Related Work	Elasticity Control Mechanism	Application Class
Gupta et al. [GFG+16]	None	Iterative MPI
Gupta et al. [GSKM13]	None	Iterative MPI
Rajan et al. [RCIT11]	None	Iterative Master/Worker
Rajan et al. [RTA+13]	None	(Iterative) Master/Worker
Vu et al. [VD14]	None	Irregular Task-parallel
Galante et al. [GR17]	Reactive	Iterative Master/Worker
Da Rosa Righi et al. [RRC+16]	Reactive	Iterative Master/Worker
Da Rosa Righi et al. [RRC+15]	Reactive	Iterative Master/Worker
Rodrigues et al. [RRC+18]	Hybrid	Iterative Master/Worker
Da Rosa Righi et al. [RRR+18]	Hybrid	Iterative Master/Worker
Shankar et al. [SKP+18]	Reactive	Linear Algebra
Raveendran et al. [RBA11]	Hybrid	Iterative MPI
Rathnayake et al. [RLT17]	Proactive	N-Body Simulation
Rajan et al. [RT17]	Hybrid	Split-Map-Merge
Haussmann et al. [HBK19]	Reactive	Irregular Task-parallel

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 authors also discuss the effect of problem size on the performance of parallel applications in cloud environments. It has been recognized that for some applications the communication / computation ratio decreases with increasing problem sizes. In these cases, the negative effects of low network throughput and high network latency can be reduced. The authors compare an iterative MPI-based application to a novel implementation that is based on CHARM++ [AGJ+14] and makes use of overdecomposition. They report on significantly decreased execution times, which can be achieved by combining overdecomposition and overlapping computation and communication. The authors of [GSKM13] 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. It is discussed how (tightly-coupled) iterative MPI-based application can be converted into a cloud-aware parallel applications. Here, also overdecomposition is used.

Several directives for building cloud-aware parallel applications are presented in [RCIT11]. Moreover, the Work Queue framework [BRA+11] is used to implement a prototypical cloud-aware parallel application. 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 originally implemented based on MPI to an elastic application. The measured performance is comparable to the MPI-based version of the application. Several other applications are discussed by the authors in [RTA+13].

The authors of [VD14] address irregular task-parallel applications and present a work stealing algorithm for load balancing that selects victims (other processing units) based on the measured network link latency. Processing units with a lower latency are preferred for stealing operations. The presented algorithm is self-adaptive and has been shown to outperform other load balancing mechanisms that do not consider network link latency.

As shown in Table 2.1, related work also investigates on how to make use of elastic scaling. In [GR17], it is discussed how parallel applications can be enhanced to make use of elasticity. According to the authors, new tools and frameworks are required to re-engineer parallel applications. Additionally, an approach that enables programming-level control of the number of processing units is described. By employing a so-called elasticity API, developers are able to implement scaling actions as part of the application's source code. Several ideas how to transform a given iterative MPI-based application into an elastic one are discussed. Based on an analysis of research opportunities, the authors also state that specifically irregular applications can benefit from elasticity because their parallel performance is hard to predict. However, concrete approaches are not described.

The authors of [RRC+16] describe a reactive elasticity control mechanism for iterative applications. The presented concept called AutoElastic supports the automated transformation (source-to-source translation) of existing MPMD-based MPI-2 applications with a master/worker architecture into elastic parallel applications. MPI-2 features dynamic process management and thus supports a varying number of MPI processes [GTL99]. Consequently, elasticity can be employed by adapting the number of worker processes for each iteration by means of an elasticity controller that derives the number of worker processes required based on monitoring data and defined thresholds. The presented concepts are evaluated by using a numerical integration application which simulates different workload patterns (e.g., ascending, descending, and wave workload). The authors state that their approach is specifically designed for embarrassingly parallel applications. Scaling actions are based on upper and lower CPU utilization thresholds. Similar concepts are discussed by the authors in [RRC+15]. A hybrid elasticity controller for iterative applications that enhances the aforementioned concepts is presented in [RRC+18]. It is based on a technique called live thresholding, which is also used in [RRR+18]. Live thresholding dynamically adapts the thresholds of a reactive elasticity controller, which is implemented as a closed feedback-loop architecture. Workload patterns are detected by comparing the last two average load values calculated based on monitored time series data and simple exponential smoothing. Similarly to the evaluation presented in [RRC+16], only the utilization of processing units is monitored.

In [SKP+18], the opportunities related to elasticity in the context of a parallel application for distributed Cholesky decomposition are discussed. As the execution time increases it becomes harder to efficiently employ processing units thus overhead increases and the workload intensity decreases. The authors argue that static resource allocation is inefficient in this case and present an architecture for parallel processing on serverless computing platforms accomplished with a reactive elasticity control (auto-scaling) mechanism. The scaling policy underlying the elasticity control mechanism considers a trade-off between fast execution and low monetary costs and can be manually selected upfront. The resulting execution engine called

numpywren specifically targets linear algebra workloads that should be operated on serverless computing platforms and has been shown to be more cost-efficient compared to other frameworks.

Proactive and reactive elasticity control mechanisms are combined by the authors of [RBA11] to transform MPI-based iterative-parallel applications into elastic applications. They describe how to adapt existing MPI-based applications to deal with a dynamically changing number of processing units. The presented approach basically terminates a running application (with check-pointing) and restarts the application with a different number of processing units. Termination can only be applied at the end of an iteration. The described elasticity controller is designed to optimize several user-defined constraints such as the desired execution time. The elasticity controller estimates the execution time of the application based on the number of iterations and the average iteration time. 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. The presented approach is built upon the assumption that, in general, a lower number of processing units can be exploited more efficiently and thus also incurs lower monetary costs to process a problem (as discussed in Section 3.1). Consequently, overhead is not directly measured but implicitly considered by scaling down when the user-defined execution time can also be met with less compute resources.

The trade-off between the monetary costs of a parallel computation and the quality of the results obtained is discussed in [RLT17]. 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 employs predictions based on an application model that is parameterized using previous measurement results. Their selection is based on mapping the

resource requirements to the available resources of a specific configuration in terms of the number of instructions that have to be / can be executed. 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 (cf. Section 3.1) are not considered.

The authors of [RT17] specifically address applications that can be developed and executed using the so-called *split-map-merge* paradigm. According to the authors this includes bag-of-tasks, bulk synchronous parallel, and Map-Reduce applications. The authors consider the cost-time product as objective function. Therefore, the automated decision-making process considers (1) information on the input problem, (2) an application model built for split-map-merge applications, (3) a user-defined objective function (in this case the cost-time product as a function of the number of processing units), and (4) information on the execution environment (e.g., processing speed, network bandwidth) obtained by measuring sample workloads. Consequently, this approach combines proactive and reactive elasticity control mechanisms. Predictions based on the application model are used to select the optimal number of processing units while changing characteristics of the execution environment are handled in a reactive manner by continuously evaluating the objective function. For instance, when the characteristics of the cloud environment (e.g., processing speed, network bandwidth) are expected to change at runtime, an elasticity controller monitors the environment and continuously evaluates the objective function based on monitoring data. When the optimal (with respect to the user-defined objective function) number of processing units changes, the elasticity controller dynamically adapts the resource configuration by adding or removing processing units.

Elasticity-related opportunities and challenges for irregular task-parallel applications are discussed in [HBK19]. Their workload is unknown a priori and hard to predict. The authors 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 prevents 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 function, thus allowing cost-based selection of the number of processing units. Because one cannot reason about the effects on execution time, speedup, efficiency, and monetary costs in absolute terms, the authors present a reactive elasticity controller for heuristic cost optimization: The cost function is approximated based on metrics monitored at runtime. Therefore, the elasticity controller continuously monitors the application and evaluates the defined objective function (minimize the monetary costs based on the presented cost model). The authors empirically evaluate their elasticity controller by comparing the results of their heuristic cost optimization approach with the minimum monetary costs (which can be obtained by measuring the scalability of the application with exemplary input problems).

Findings & Analysis: Negative effects stemming from heterogeneous processing speeds and varying network latencies can be effectively reduced by employing asynchronous communication and overdecomposition, which enables dynamic load balancing across processing units. However, it has also been recognized that such an approach cannot be applied to all applications. In the context of parallel applications with frequent communication and synchronization, asynchronous communication and overdecomposition leads to fundamental performance issues and thus cannot be applied. Related work also shows how to employ elasticity. Either proactive, reactive, or hybrid elasticity control mechanisms can be employed. However, which elasticity control mechanism has to be selected largely depends on the characteristics of the class of parallel applications considered. For instance, whereas the scaling behavior of applications based on the split-map-merge paradigm can be predicted using an application model, the scaling behavior of irregular task-parallel applications is unknown upfront and hard to predict, which requires reactive elasticity control. Existing approaches to elasticity control consider different non-functional aspects such as utilization, monetary costs, or execution time to adapt the number of processing units in an automated manner. For instance, the authors of [RT17] use the cost-time product to create a single-objective optimization problem. The authors of [HBK19]

employ the concept of opportunity costs, which can be used to express time in terms of costs, thus enabling a purely cost-driven optimization. However, as of today, a comprehensive and generally applicable understanding of elasticity in the context of parallel applications does not exist. Moreover, related work mainly addresses iterative applications and proposed solutions to make parallel applications cloud-aware are most often limited to IaaS cloud environments. Whereas the need for elasticity control mechanisms and programming abstractions for further application classes (such as irregular applications) and emerging cloud paradigms (such as serverless computing) has been identified, only few existing approaches can be found in literature.

Chapter 3

Principles of Elastic Parallel Systems

Existing parallel computing theory as well as traditional metrics to analytically evaluate parallel systems do not comprehensively consider the novel ability to control the number of processing units at runtime. To address these issues, a conceptual framework to understand elasticity in the context of parallel systems is introduced, the term *elastic parallel system* is defined, and a novel metric called *workload efficiency* to analytically reason about a parallel system at runtime is provided. Moreover, the performance metrics *elastic speedup* and *elastic efficiency* are defined to enable the ex-post performance evaluation of elastic parallel systems. Based on the conceptual framework, the key findings are discussed in detail.

This chapter is structured as follows. Section 3.1 introduces a conceptual framework to understand elasticity in the context of parallel systems. Ex-post performance metrics to evaluate elastic parallel systems are presented in Section 3.2. Section 3.3 concludes this chapter by discussing how elasticity can be beneficially employed in the context of different application classes.

3.1 Elastic Parallel Systems

In this section, a conceptual framework is introduced to understand elasticity in the context of parallel systems. The conceptual framework is visualized in Figure 3.1. It shows an *elastic parallel system*, which is defined as a parallel system[1] accompanied by an elasticity controller that controls the number of processing units at runtime. Therefore, the elasticity controller reasons about the scaling behavior of the parallel system to make decisions on the optimal number of processing units. Existing parallel computing theory does not comprehensively consider this novel ability to control the number of processing units at runtime. For instance, traditional metrics to analytically evaluate parallel systems such as speedup assess the performance averaged over the entire execution time [BT89; GGKK03; JA02]. By definition, speedup can only be calculated, when the computation has been completed as it depends on the parallel execution time. This also holds for the parallel efficiency. Additionally, the definitions of speedup and parallel efficiency are based on a fixed number of processing units p. Elastic parallel systems, on the other hand, are characterized by a dynamic number of processing units, which can be adapted at runtime: Thus, the number of processing units is a function of time $p(t)$ that is implicitly defined by the elasticity controller. As a result, existing metrics to analytically evaluate parallel systems cannot be used by the elasticity controller to reason about the scaling behavior of a parallel system at runtime. The conceptual framework described in the following provides the fundamental concepts to discuss the opportunities and challenges that arise in the context of elasticity. On this basis, a novel metric called *workload efficiency* is discussed to analytically reason about the scaling behavior of a parallel system at runtime. Moreover, the performance metrics *elastic speedup* and *elastic efficiency* are defined to enable the ex-post evaluation of elastic parallel systems.

Traditionally, elasticity has been considered in the context of interactive (multi-tier) applications, i.e., applications that process independent user

[1] A parallel system is a combination of a parallel algorithm (parallel application, programming model / middleware) and a parallel architecture (hardware) [GGKK03].

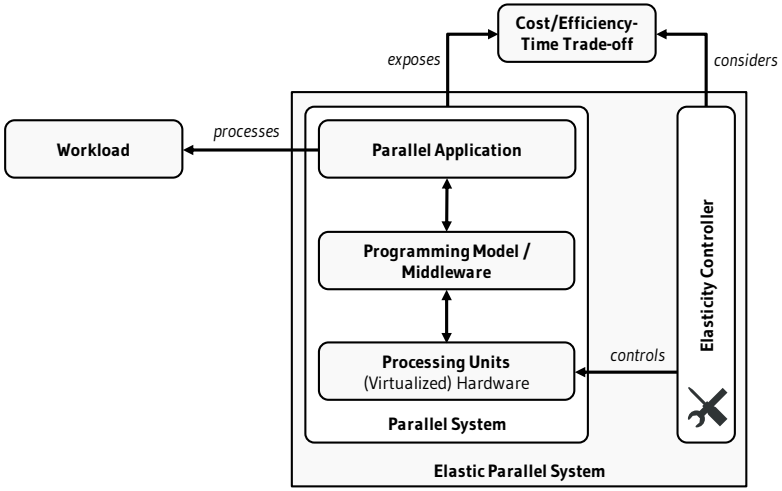

Figure 3.1: Conceptual framework for understanding elasticity in the context of parallel systems

requests. Thus, the workload has been described in terms of user requests processed by an application [HKR13]. Whenever the arrival rate of user requests varies over time, these applications benefit from elasticity by dynamically adapting the number of processing units to control the response time [HKR13]. On the other hand, the workload of parallel systems cannot be described as independent user requests. Instead, it depends on a single problem, which requires a specific number of *(basic) computational steps* to be solved [GGKK03]. Moreover, parallel overhead occurs in the manifestation of idling, communication, and excess computation. To model parallel overhead in the framework, one can distinguish between three different types of steps that can be executed by a processing unit: (1) *essential* computational steps, (2) *non-essential* computational steps, and (3) *idling* steps. All sources of overhead can be modeled as one (or more) of these types. For instance, communication and synchronization of data across distributed processing units leads to idling steps and additional computational steps that are not executed by a sequential algorithm and thus can be classified as

non-essential.

Under this consideration, the workload of a parallel system is defined as follows:

Definition 3.1 (Workload)
The workload $W(I)$ of a parallel system is given by the total number of essential computational steps executed based on a defined input I.

The input I describes a specific problem that has to be solved and the workload is given by the total number of essential computational steps required to solve the problem. Note that, under the assumption that it takes unit time to perform a single computational step, the workload is equivalent to the sequential execution time T_{seq}. Note also that the proposed definition of workload is in consequence identical to the definition of the *problem size* as typically considered in parallel computing theory (cf. Section 2.1.2).

With elasticity, one has the ability to adapt the number of processing units, which are employed to process a specific problem, at runtime. This adaptation changes the physical parallelism of a parallel system, which is modeled as the processing rate.

Definition 3.2 (Processing Rate)
The processing rate $R(p(t), \tau)$ of a parallel system describes the total number of steps that can be executed (by all employed processing units) in time interval τ.

The elasticity controller can dynamically adapt the processing rate of a parallel system by adding or removing processing units (cf. Figure 3.1). Additionally, the processing rate is affected by every change of the processing rate of a single processing unit: Individual processing rates might change over time due to virtualization and resource pooling.

The major motivation for parallel processing is to shorten the application's execution time by adding more processing units to the computation. However, due to the inherent overhead, one cannot assume linear scalability.

Instead, the elasticity controller has to consider a *cost/efficiency-time trade-off*: Whereas adding more processing units effectively reduces the execution time, a higher number of processing units also leads to a lower efficiency and thus results in higher monetary costs [HBK19; RT17]. This leads to two conflicting optimization goals: (1) Reduce monetary costs & maximize efficiency vs. (2) shorten execution time.

In the following, the cost/efficiency-time trade-off inherent to parallel systems is discussed in more detail by comparing them to an (ideal) perfectly scalable parallel system based on a given cost model. Therefore, the monetary costs of a parallel computation in the cloud C_{par} are modeled as follows:

$$C_{par}(I, p(t)) = T_{par}(I, p(t)) \cdot \bar{p} \cdot c_{\pi}, \tag{3.1}$$

where T_{par} is the execution time, $\bar{p}$ is the time-averaged number of processing units employed, and c_{π} the price of one processing unit per time unit.

For a perfectly scalable parallel system, the execution time decreases, by definition, linearly with an increasing number of processing units. By revisiting Equation 3.1, one can see that the monetary costs to process a given workload with a perfectly scalable parallel system are independent of the number of processing units employed[1]. Thus, in theory, an unbounded number of processing units can be employed to speed up the computation without increasing a computation's monetary costs. Realistic parallel systems, on the other hand, are not perfectly scalable. Whereas employing more processing units reduces the execution time, it also leads to higher overhead (and thus lower efficiency) [EZL89]. In cloud environments, one has to pay processing units (or compute resources in general) per time unit. As a consequence, one does not only pay for the essential computational steps executed but also for the overhead that occurs (in form of non-essential computational steps and idling steps).

Figure 3.2 depicts three different application runs with different numbers

[1] This only holds under the assumption that there are zero provisioning costs.

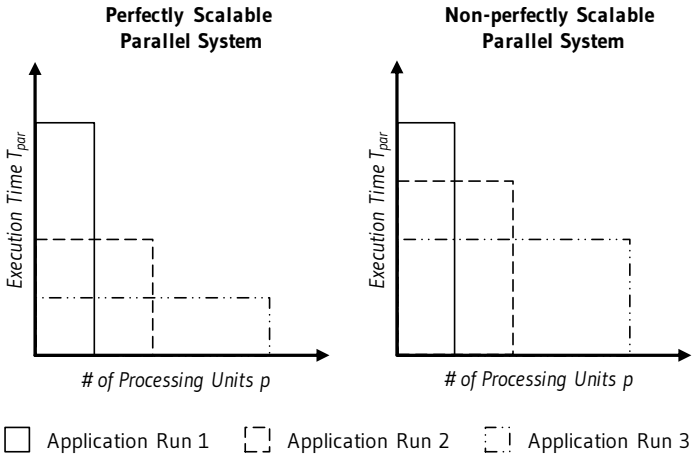

Figure 3.2: Whereas the monetary costs of a perfectly scalable parallel system are independent of the number of processing units employed, for parallel systems that are not perfectly scalable, the monetary costs increase with the number of processing units. Here, the monetary costs are expressed as the sizes of the areas shown.

of processing units for a perfectly and a non-perfectly scalable parallel system. The areas shown for each application run correspond to the monetary costs of a parallel computation in the cloud C_{par} (cf. Equation 3.1). As one 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 leading to higher monetary costs. This results from a higher number of non-essential computational steps and idling steps, which increases with the number of processing units. How many non-essential computational steps and idling steps occur for which number of processing units depends on the scaling behavior of the parallel system considered.

To dynamically adapt the number of processing units, the elasticity controller has to reason about the cost/efficiency-time trade-off at runtime. Therefore, the *workload efficiency* - a novel metric to analytically reason

about the scaling behavior of a parallel system at runtime - is introduced. In the following, the *workload intensity* is introduced based on which the workload efficiency can be defined.

Definition 3.3 (Workload Intensity)
The workload intensity $WI(I, p(t), \tau)$ of a parallel system describes the total number of essential computational steps executed in time interval τ.

The workload intensity quantifies the amount of useful work executed by all employed processing units as a function of time. The term *workload intensity* is adopted from [HKR13], where it has been used in the context of interactive (multi-tier) applications. Based on the processing rate and the workload intensity, the workload efficiency is defined as follows:

Definition 3.4 (Workload Efficiency)
The workload efficiency $WE(I, p(t), \tau)$ of a parallel system describes the ratio of the workload intensity to the processing rate in time interval τ: $WE(I, p(t), \tau) = \frac{WI(I,p(t),\tau)}{R(p(t),\tau)}$

The workload efficiency quantifies the percentage of time in which all processing units employed by a parallel system execute essential (basic) computational steps, i.e., computational steps that are also executed by a corresponding sequential implementation, within a defined time interval. Note that the definition of workload efficiency differs from the common notion of utilization, which can be defined as follows:

Definition 3.5 (Utilization)
The utilization $U(I, p(t), \tau)$ of a parallel system describes the ratio of the total number of essential and non-essential computational steps executed to the processing rate in time interval τ.

Utilization does not distinguish between essential and non-essential computational steps, with the latter stemming from overhead, e.g., in form of excess computation.

Whenever the workload efficiency changes at runtime, the elasticity controller has to evaluate the cost/efficiency-time trade-off. Such changes result from (1) a changing workload intensity, (2) a changing processing rate, (3) a combination of both. Under the assumption that the processing rate is constant, a changing workload intensity is typically related to the parallel algorithm, the input data considered (problem to be solved), and the current state of the computation: Excess computation and communication overhead are considered in form of non-essential computational steps and idling steps, which both affect the workload intensity. Moreover, changing characteristics of the execution environment (e.g., network latency and network bandwidth) affect the workload intensity by introducing additional overhead. A changing processing rate is typically related to a changing number of processing units or changing processing rates of individual processing units (i.e., varying processing speed due to virtualization and resource pooling).

Figure 3.3 visualizes examples of the three aforementioned scenarios. Figure 3.3 (a) shows a scenario, where the processing rate is constant, but the workload intensity decreases as the execution time increases. This is related to growing overhead, which makes it harder to efficiently exploit all available processing units. Figure 3.3 (b) shows a scenario, where the processing rate doubles because the elasticity controller provisions additional processing units, but the workload intensity does not change. This could be the result of not sending work to the newly provisioned processing units, which consequently do not contribute to the computation. Figure 3.3 (c) shows a scenario, where the processing rate doubles because the elasticity controller provisions additional processing units. Because more processing units contribute to the computation, the workload intensity also increases. However, it does not double because the parallel system is not perfectly scalable. In all three cases, the workload efficiency changes at runtime and thus the elasticity controller has to dynamically evaluate the cost/efficiency-time trade-off.

To evaluate the cost/efficiency-time trade-off, the elasticity controller combines knowledge about efficiency, monetary costs, and time. Whether the execution time T_{par} and thus also the monetary costs C_{par} can be predicted

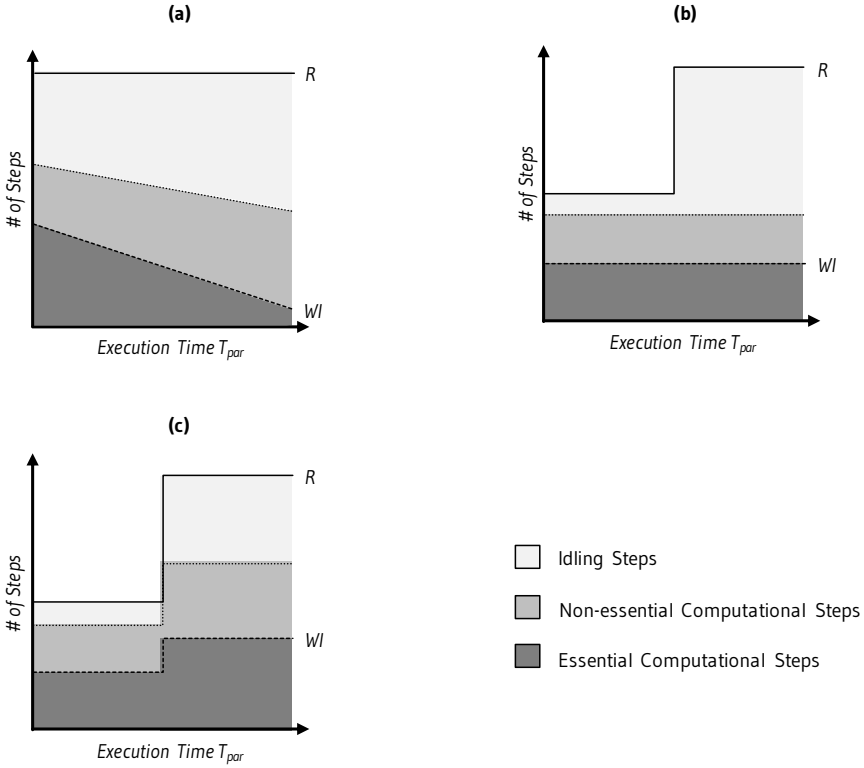

Figure 3.3: Different scenarios of workload efficiency changes at runtime

or not largely depends on the application class considered. However, the workload efficiency metric enables the elasticity controller to analytically reason about the scaling behavior of the parallel system at runtime, time can be considered as wall-clock time, and the current monetary costs per time unit can be calculated by multiplying the current number of processing units with c_π (cf. Equation 3.1). Section 3.3 discusses how this affects the construction of elastic parallel system architectures.

3.2 Ex-post Performance Metrics

As discussed initially, traditional performance metrics such as speedup S and efficiency E are functions of a specific input I and a fixed number of processing units p. As a result, these metrics cannot adequately be used for ex-post evaluation of elastic parallel systems. To deal with this issue, the authors of [RCRR16] define elastic speedup and elastic efficiency as functions of (1) an initial number of VMs, (2) a lower bound for the quantity of VMs, and (3) an upper bound for the quantity of VMs. They use these metrics to compare application runs with and without elastic scaling. However, these definitions are tailored to their implementation of a threshold-based elasticity controller and are not generally applicable. The authors of [CCB+12] also provide a definition of elastic speedup, but focus on High Throughput Computing (HTC) and performance in terms of response time. In the following, general definitions of elastic speedup and elastic efficiency are provided based on the former discussions.

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

$$S_{elastic}(I, p(t)) = \frac{T_{seq}(I)}{T_{par}(I, p(t))} \tag{3.2}$$

$$E_{elastic}(I, p(t)) = \frac{S_{elastic}(I, p(t))}{\bar{p}} = \frac{T_{seq}(I)}{T_{par}(I, p(t)) \cdot \bar{p}}, \tag{3.3}$$

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

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. Whereas elastic speedup and elastic efficiency can be used to compare different elastic parallel systems with respect to their performance, similar

values do not imply that similar scaling strategies are used by those systems. Consequently, a detailed comparison of the elasticity control mechanisms employed is required to understand their performance.

3.3 Chapter Summary and Discussion

In this chapter, existing parallel computing theory is extended to consider elasticity, i.e., the ability to control the number of processing units of a parallel system at runtime. A conceptual framework is introduced to understand elasticity in the context of parallel systems, the term elastic parallel system is defined, and novel metrics for both elasticity control as well as the ex-post performance evaluation of elastic parallel systems are discussed. The principles presented in Section 3.1 provide the basic constructs to discuss the opportunities and challenges related to elasticity in the context of specific application classes. A major challenge for making parallel applications elastic is the cost/efficiency-time trade-off, which has to be handled by an elasticity controller. Therefore, either reactive or proactive elasticity control mechanisms can be employed. Whereas reactive elasticity control mechanisms adapt the number of processing units based on measured runtime metrics, proactive elasticity control mechanisms employ predictions to select the number of processing units according to user-defined goals. Examples of both reactive and proactive elasticity control mechanisms are discussed in the following chapters.

The major challenge related to reactive elasticity control is to define appropriate runtime metrics to consider the cost/efficiency-time trade-off. The workload efficiency as described in Section 3.1 provides a useful means to enable reactive elasticity control in order to handle the cost/efficiency-time trade-off at runtime even though only limited knowledge about the efficiency, monetary costs, and execution time is available. However, note that the workload efficiency has to be interpreted with care. For instance, the currently measured workload efficiency might be extremely low because it has been measured during a global communication phase. Thus, simply

removing processing units based on a measured low workload efficiency might not be appropriate. Rather, monitoring data has to be combined with knowledge about the parallel algorithm, input data, the parallel system architecture, and the current state of the computation to reason about the optimal number of processing units. Chapter 5 shows how to employ a reactive elasticity controller to handle the cost/efficiency-time trade-off in the context of state space search. Applications based on the state space search technique are input-dependent and their execution time and scaling behavior are hard to predict. Thus, selecting the number of processing units a priori is a difficult task. These applications benefit from dynamically adapting the number of processing units according to user-defined thresholds.

The major challenge related to proactive elasticity control is the generation of a model that allows accurate predictions. It is shown that for some applications predictions based on input data allow a proactive elasticity controller to select the number of processing units according to user-defined goals. To generate a prediction model, the scaling behavior has to be considered. Based on the resulting model, input data, and the user-defined goals, an elasticity controller then determines the optimal number of processing units. Chapter 6 describes how to construct an elastic parallel system architecture that enables parallel computations based on serverless computing platforms. This system architecture also includes a proactive elasticity controller that predicts the required number of processing units to meet a user-defined execution time limit after which the result of the computation needs to be present while minimizing the associated monetary costs. The underlying prediction model is obtained and refined in an automated manner by employing supervised learning to infer the scaling behavior from labeled performance measurement data of previous application runs.

As defined in Section 3.1, on a conceptual level, an elastic parallel system simply comprises a traditional parallel system. However, from an architectural perspective, it is a fundamental difference if a parallel system is designed for a static or a dynamic number of processing units, which also affects the implementation of parallel applications. Architectural challenges are discussed in more detail in Chapter 4. In particular, a systematic method

to assess the impact of parallel design decisions on the cloud readiness of an application is described.

To ease the implementation of elastic parallel applications, higher-level programming abstractions can be employed, which separate software development concerns from parallel execution concerns (cf. Section 2.1.1). However, a major issue related to elastic parallel applications is the requirement to support a dynamic number of processing units. Therefore, two different approaches can be used: On-demand task generation and overdecomposition. Whereas with on-demand task generation new tasks are created by splitting work from existing ones, by following an overdecomposition strategy, more tasks than actually required are created (e.g., at the beginning of the computation). Chapter 5 and 6 show how to enhance established parallel execution models and programming abstractions in this regard. As part of Chapter 5, a runtime system for elastic parallel state space search as well as an exemplary development framework for elastic branch-and-bound is presented. Elastic parallel computations are ensured by employing on-demand task generation (called potential splitting in Chapter 5). As part of Chapter 6, a novel approach to parallel cloud programming called serverless skeletons is described and evaluated based on a prototypical development and runtime framework. In this case, elastic parallel computations are ensured by employing an overdecomposition strategy.

Chapter 4

Design Trade-offs in High Performance Cloud Computing

Cloud-native [ABLS13; FLR+11; FLR+14; KQ17; LBWW17] and parallel application design [GGKK03; KMMS10; MMS01; MMS07] follow conflicting goals. Whereas cloud-native applications are strictly designed according to the specific characteristics of cloud environments, parallel application design aims at uncompromising performance maximization in terms of speedup and efficiency. Trade-off handling is required to consider parallel performance and the characteristics of cloud environments at the same time.

To address this issue, a systematic approach for assessing the cloud readiness of parallel applications based on the design decisions made is presented. The assessment is based on a meta model that defines a multi-dimensional scale for measuring the cloud readiness of parallel applications. A model based on the meta model is created, which captures the impact of essential parallel design decisions on the cloud readiness of an application. To eval-

uate this model and the assessment procedure, an extensive case study is described, which discusses the cloud readiness of three parallel applications.

This chapter is structured as follows. The procedure to handle design trade-offs as well as the underlying meta model are presented in Section 4.1. Section 4.2, 4.3, and 4.4 comprise an instantiation of the meta model to enable the assessment procedure. To evaluate this model, an extensive case study is described in Section 4.5. Related work is presented in Section 4.6. Finally, Section 4.7 concludes this chapter.

4.1 Meta Model & Cloud Readiness Assessment

Cloud-native design guidelines [ABLS13; LBWW17] and cloud pattern languages [FLR+11; FLR+14] are driven by prescriptive *cloud application properties* (such as the IDEAL properties [FLR+14]) that have to be considered during application design. As a result, cloud-native design describes an ideal cloud application, i.e., an application with maximum cloud readiness. However, parallel applications are traditionally designed with the ultimate goal to maximize parallel performance, which restricts cloud-native design. Consequently, parallel application design has to consider design trade-offs related to cloud-specific characteristics.

The goal is to provide a systematic approach for handling these design trade-offs, which helps to understand implications of parallel design decisions on an application's cloud readiness. Therefore, prescriptive cloud application properties are applied to assess the cloud readiness of parallel applications that are targeted for cloud migration. In the following, an assessment procedure based on the underlying meta model depicted in Figure 4.1 is described.

A parallel application has several characteristics that are the result of specific *(parallel) design decisions* made to construct an application for a specific problem following an iterative approach [KMMS10; MMS01]. Making these design decisions leads to descriptive application characteristics, which are called *parallel application properties* in the following. Explicit

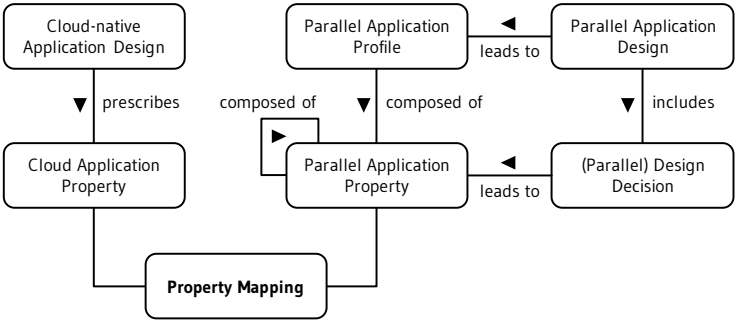

Figure 4.1: Meta model that enables the assessment of a parallel application's cloud readiness

documentation of parallel application properties leads to an abstract profile of the application. The resulting artifact is called *parallel application profile*.

Some of the aforementioned cloud application properties are influenced by parallel application properties. By systematically correlating both sets of application properties with *property mappings*, one is able to assess their *conceptual fitness*. As cloud application properties describe an application with maximum cloud readiness, property mappings allow us to qualify the impact on cloud readiness resulting from a parallel design decision as either positive or negative:

Let C be the set of all cloud application properties; let P be the set of all parallel application properties; and let $M \subseteq C \times P$ be the set of property mappings. The *conceptual fitness function* λ is defined as follows:
$\lambda : M \to \Psi$, where $\Psi = \{+, -\}$, $+$ refers to a positive, and $-$ to a negative conceptual fitness. Thus, the conceptual fitness function λ maps a property mapping $m \in M$ to a specific conceptual fitness value $\psi \in \Psi$.

An instantiation based on the meta model defines C, P, M, as well as the graph of the function λ, i.e., the set of all ordered pairs $(m, \lambda(m))$. Such a model enables the assessment of a parallel application profile $P_{app} \subseteq P$, which describes a concrete application.

A cloud readiness assessment based on the resulting model expects an

Algorithmus 4.1 Cloud Readiness Assessment

```
1: procedure AssessCloudReadiness(P_app)
2:     R ← {}                                          ▷ cloud readiness set
3:     for each p ∈ P_app do
4:         for each c ∈ C do
5:             if (c, p) ∈ M then
6:                 ψ = λ(c, p)                         ▷ calculate conceptual fitness
7:                 R ← R ∪ {(p, c, ψ)}
8:             end if
9:         end for
10:    end for
11:    return R
12: end procedure
```

argument P_{app} and evaluates its cloud readiness based on λ. The result of the cloud readiness assessment is defined as the *cloud readiness set* $R \subseteq P_{app} \times C \times \Psi$. Algorithm 4.1 defines the *cloud readiness assessment* procedure that calculates R based on a given P_{app}.

Such a systematic approach structures this highly multi-dimensional design space based on application properties and allows finding the sweet spots to improve an application's cloud readiness. Whereas further aggregation of R could be applied, this avoids finding the sweet spots, which is important to optimize applications with respect to their cloud readiness. Section 4.5 describes the results of three exemplary cloud readiness assessments based on archetypal parallel applications.

In the following, an instantiation of the described meta model is constructed that captures the set of cloud application properties C (cf. Section 4.2) and the set of parallel application properties P resulting from parallel design decisions (cf. Section 4.3). Further, the set of property mappings M is derived and the graph of the conceptual fitness function λ is specified (cf. Section 4.4).

4.2 Cloud Application Properties

Cloud-native applications are characterized by specific properties that enable them to benefit from cloud environments effectively [FLR+14; LBWW17]. In the following, the set of *cloud application properties* C is described in detail. Each $c \in C$ is identified by its initial letter.

Distribution (d): Cloud-native applications have to be decomposed into separate application components that can be distributed to utilize cloud resources. This is also required for horizontal scaling [FLR+14].

Elasticity (e): Cloud applications are expected to be horizontally scalable. Therefore, new resources are provisioned to scale up and existing resources are decommissioned to scale down. The cloud application has to make use of new resources upon provisioning and it has to free existing resources for decommissioning. This makes a cloud-native application elastic, i.e., it is able to employ added resources based on horizontal scaling [TBB+17]. Dynamic adaptation of resources has a huge impact on the cost metric since resources are billed on a pay-per-use basis.

Isolated State (i): Cloud environments typically provide an availability guarantee for the whole platform or infrastructure and not for the components provisioned [FLR+14]. Thus, customers cannot expect individual components to be highly available. To cope with this issue, cloud-native applications encapsulate state information in as less application components as feasible, i.e., most application components should be designed as stateless. One can differentiate between session state, i.e., the state of an interaction handled by the application, and application state, i.e., the data handled by the application [FLR+14]. Both, session state and application state should be managed outside of an application component to facilitate horizontal scaling. This is typically achieved by decoupling compute from storage [CDE+13; VGS+17].

Loose Coupling (l): Cloud-native applications consist of a set of connected application components that might change at runtime (e.g., by adding or removing resources) [FLR+14]. Loose coupling between these components facilitates procedures such as scaling and failure recovery by reducing

dependencies on each other.

Finally, **automated management** is described as important application property [FLR+14]. All formerly mentioned properties support the ease of management at runtime. However, as management is an operations-related property it cannot be directly linked to parallel application design on a conceptual level. Nevertheless, property mappings allow us to derive implications on management (for an example see Section 4.5).

4.3 Parallel Application Design & Properties

Parallel application design has been described in several forms including top-down design processes [GGKK03] and pattern languages [KMMS10; MMS01; MMS07]. To make use of parallelism, typically a single, large problem has to be split into tasks. Several strategies to decompose a problem into tasks are described in the following. Then, the essence of parallel application design is summarized in form of parallel design decisions. Each parallel design decision comprises several parallel application properties.

In *data decomposition*, tasks are partitioned by splitting the data structures they operate on. This is useful for processing large data structures [MMS01]. Depending on the requirements of the application, input, output, or intermediate data can be considered to determine the decomposition [GGKK03]. *Recursive decomposition* refers to a decomposition strategy, where work is recursively divided by splitting tasks until an atomic granularity threshold is reached. Recursive task decomposition is typically used in divide-and-conquer approaches. *Exploratory decomposition* is required whenever a computation dynamically expands a search space. In this case, the search space can be split into smaller parts and tasks, meant to apply the search procedure to these parts in parallel. *Speculative decomposition* relates to processing different options of a future computation in parallel to speed up the overall progress [KMMS10]. Later on, only a single option of the formerly processed ones is chosen. The result of this option can be directly returned, all other options have to be discarded [GGKK03].

Task Generation: An important design decision is how task generation is performed. Task generation might either be *static* or *dynamic*. With static task generation all tasks are specified before their parallel execution starts. On the other hand, dynamic task generation enables the generation of new tasks at runtime. However, static task generation yields a better parallel efficiency by reducing task generation overhead and thus is used whenever applicable [MMS01].

Task Mapping: Similar to task generation, task mapping can be either *static* or *dynamic*. Task mapping refers to the process of mapping generated tasks to available compute nodes. Whereas statically generated tasks might either be mapped statically or dynamically, dynamically generated tasks require dynamic task mapping. Dynamic task mapping can further be categorized into *centralized* and *decentralized* approaches [GGKK03].

Task Interaction: Non-trivial parallel applications are characterized by task interactions. Task interactions might either be *structured* or *unstructured*. Structured task interaction refers to a known interaction pattern that can be exploited for optimization purposes [MMS01], whereas *unstructured* interaction is unpredictable by nature.

Size of Associated Data: Tasks typically have some kind of data attached that is required for processing. The size of associated data might either be *small* or *large*. Within application-specific bounds, the size of associated data can be adjusted.

Communication Model: In a *synchronous* model, components proceed simultaneously and all participating parties of an interaction are actively involved in the communication process by either writing or reading data (in shared memory environments) or sending or receiving messages (in distributed memory environments) [MMS07]. A synchronous model requires knowledge on the time for an interaction to happen and the speed of individual processing components. On the other hand, an *asynchronous* model poses lesser restrictions on the participants of an interaction.

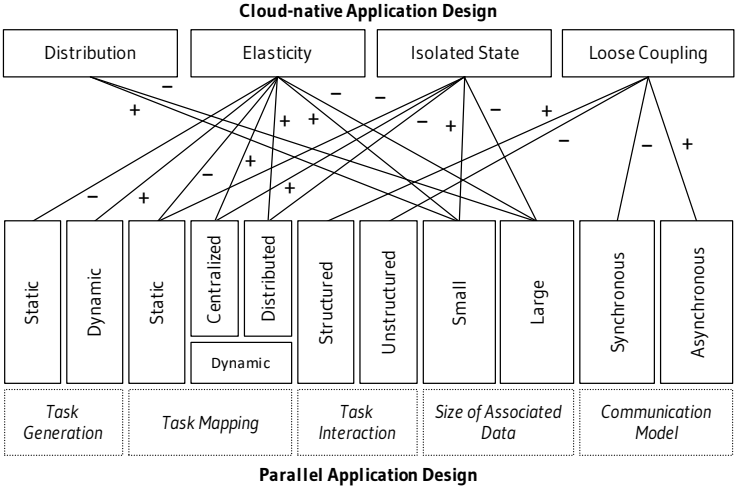

Figure 4.2: Property mappings $m \in M$ and their conceptual fitness $\psi \in \Psi$ assigned by λ

4.4 Property Mappings

In the following, the set of *property mappings* M is discussed and the graph of the conceptual fitness function λ is specified. This section is organized according to the cloud application properties (cf. Section 4.2). For each parallel application property within a subsection, the conceptual fitness is discussed in a short text paragraph. Figure 4.2 visualizes the resulting set of property mappings M and their conceptual fitness assigned by λ.

In the following, a conceptual fitness $\psi \in \Psi$ resulting from the evaluation of $\lambda(c, p)$, where $c \in C$ and $p \in P$, reads as c^{ψ} in the context of p.

4.4.1 Distribution

Tasks may be either processed in parallel on a shared memory multiprocessor computer system or in a distributed memory multiprocessor computer system. The latter refers to a distributed system of compute nodes, in which each compute node manages its own memory. In the cloud, only distributed

computing supports horizontal scaling and thus distribution is an important property of cloud applications.

Small Size of Associated Data (d^+): A small size of associated data enables rapid distribution across compute nodes.

Large Size of Associated Data (d^-): Large size of associated data favors local computation over time-consuming distribution.

4.4.2 Elasticity

In traditional HPC environments, compute resources are allocated in a static manner. Thus, the concept of elasticity is a new aspect that has to be considered by parallel applications. However, there are many applications suffering from static resource provisioning. Specifically, parallel applications with unpredictable resource requirements may benefit from rapid elasticity in cloud environments [GEM+16].

Static Task Generation (e^-): When static task generation is applied, the number of tasks is usually matched to available compute nodes. Whereas this maximizes parallel efficiency by reducing overhead, it limits elastic scaling of the application as the number of tasks generated defines the achievable scalability for a parallel application. Moreover, task sizes cannot be changed at runtime, which might lead to idle time whenever processing speed of individual compute nodes varies.

Dynamic Task Generation (e^+): Dynamic task generation allows to adapt the number of tasks generated at runtime and thus naturally leads to a scalable application. Similarly, dynamic adaptation of task sizes can be applied. However, task granularity has to consider application-specific restrictions.

Static Task Mapping (e^-): Static task mapping limits elastic scaling. As tasks are inherently bound to components, they cannot be rescheduled across the distributed system. Thus, one cannot employ newly provisioned compute nodes effectively and cannot decommission existing ones.

(Cent. / Dist.) Dynamic Task Mapping (e^+): Dynamic task mapping is a good match for elasticity. It provides the mechanisms to map tasks to newly added nodes in the distributed system. However, management has

to consider the rescheduling of tasks that have been mapped to nodes that should be removed as part of an elasticity action.

Small Size of Associated Data (e^+): If associated data is small, rescheduling of tasks is much easier and thus supports rapid elasticity.

Large Size of Associated Data (e^-): If associated data is large, rescheduling of tasks is hard and limits rapid elasticity.

4.4.3 Isolated State

For parallel applications, isolating state in a minimum of components most often leads to performance degradation as communication is required whenever data has to be fetched from a remote host.

Static Task Mapping (i^-): Static task mapping leads to distributed state and thus requires stateful components. Typically, data decomposition is combined with static task mapping. Therefore, state is distributed across all components with each component applying the same algorithm to a local data set. This technique is called the *owner computes rule* [GGKK03].

Centralized Dynamic Task Mapping (i^+): Centralized dynamic task mapping supports isolating state in a minimum of components. Following this approach, only a single, central component stores tasks and schedules these tasks across the other components for parallel processing.

Distributed Dynamic Task Mapping (i^-): In contrast to centralized dynamic task mapping, task mapping is based on a distributed algorithm, which avoids isolated state. Nevertheless, this might be required to optimize parallel performance, e.g., whenever a centralized algorithm would become a bottleneck.

Small Size of Associated Data (i^+): If associated data is small, tasks can be stored in a minimum of application components without heavy performance penalties.

Large Size of Associated Data (i^-): If associated data is large, parallel efficiency suffers from external storage because transferring tasks leads to high communication overheads.

4.4.4 Loose Coupling

Due to virtualization (leading to CPU time-sharing and memory overcommitment), differences with respect to processing times of individual components are the common case in standard cloud environments. Whereas tight coupling might be more efficient in general, such an approach produces overhead in form of idle time because the slowest component forces all others to wait. Moreover, tightly coupled components limit elasticity management.

Structured Task Interaction (l^+): Structured task interactions comprise regular interaction patterns that can be exploited to ensure loose coupling. Whenever one can isolate complex interaction patterns between tasks, one can use this information to guide task mapping in order to minimize coupling across component boundaries.

Unstructured Task Interaction (l^-): Unstructured task interactions do not comprise exploitable patterns and thus most often lead to suboptimal task distribution across the system.

Synchronous Comm. Model (l^-): Synchronous communication leads to tightly coupled participants of an interaction. Whenever the participants of an interaction progress collectively, synchronous communication can be applied. This cannot be expected to be the case in standard cloud environments.

Asynchronous Comm. Model (l^+): Asynchronous communication decouples participants of an interaction by non-blocking mechanisms. This avoids idle time by allowing the requester to proceed with additional computational tasks while waiting for a response.

4.5 Case Study

In this section, several parallel applications for protein folding, a problem widely recognized as grand challenge, are studied. Among many other applications, protein folding has major implications for research into Alzheimer's disease and many forms of cancer. As protein folding is a computationally intensive problem, cloud services for protein folding would enable scientists to benefit from a pay-per-use billing model and rapid experimentation.

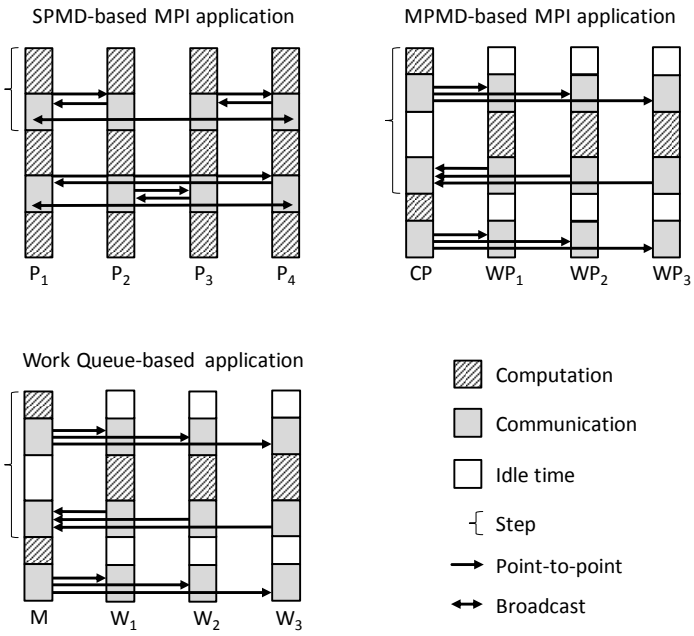

Figure 4.3: Computational steps and interaction patterns of exemplary parallel applications

First, a common method to solve the protein folding problem is described. Thereafter, it is shown how to assess the cloud readiness of three archetypal parallel applications employing this method. Table 4.1 summarizes the corresponding parallel application properties.

4.5.1 Replica Exchange Molecular Dynamics

Replica exchange molecular dynamics (REMD) is a common approach for the protein folding problem [BSVI07; SO99]. To solve the problem, one has to find the lowest energetic configuration of a protein consisting of amino acid residues by rotating and folding those spatially. Considering the whole configuration space, a multi-dimensional energy landscape emerges, where each position in this landscape corresponds to a certain arrangement of the

amino acids and the associated energy.

Typically, multiple simulations are applied to the configuration space of a protein, where each simulation requires a specific temperature as input. However, the energy landscape comprises many minima and barriers between these minima are difficult to cross. Whereas high temperatures allow crossing these barriers more easily, low temperatures enable a more precise sampling in a specific region but often get trapped in local minima [ED05]. Consequently, simulation results are inherently bound to the initial conditions of the system, which determine the energy space that is explored. A common approach to resolve this issue is called replica exchange molecular dynamics. Therefore, multiple simulations, which enhance sampling of the potential energy surface by random walks in energy space, are run in parallel [SO99]. Each simulation is started with a replica (i.e., an initial configuration) and a different temperature. As the simulation proceeds, energy values are evaluated based on this temperature. Neighboring replicas exchange their temperature values at runtime leading to updated configurations. An exchange is performed according to a probabilistic acceptance rule. The goal is to produce statistically valid sampling that is close to the ergodic ideal to approach a global minimum.

From a parallel computing perspective, simulations are applied to multiple replicas in parallel. Further, the simulation proceeds in a stepwise approach by executing parallel simulations followed by a replica exchange procedure between neighboring replicas, in which temperature values are exchanged.

In the following, three parallel applications for REMD are analyzed and their cloud readiness is assessed.

4.5.2 SPMD-based MPI Application

This application for REMD is based on MPI-1. MPI-1 is still the dominant approach to implement parallel applications. Such programs typically employ the Single Program Multiple Data (SPMD) technique [KMMS10; MMS07]. In each step, every process executes the simulation based on a replica and a different temperature value. Thereafter, the replica exchange procedure is

carried out as follows. Every pair of neighboring processes exchanges their temperatures based on the acceptance rule. Only one of the two processes makes this decision and sends the result to its neighbor. This is accomplished with SPMD by allowing only the process with the lower rank to execute the corresponding instructions. If an exchange is required, temperatures are sent to each other accordingly. After all process pairs finished the exchange procedure, updated temperatures are sent to all processes by means of an MPI broadcast primitive (cf. Figure 4.3).

Each replica can be processed in parallel without the consideration of task dependencies. However, task interactions result from the replica exchange procedure applied after local simulations have been finished.

4.5.2.1 Parallel Application Profile

The task decomposition technique applied here is data decomposition because each task executes the same procedure for a replica using different temperatures. Replicas are considered as primary data structure in this context. The generation of these tasks is *static* and also *static task mapping* is applied, i.e., each MPI process executes one task. However, task interactions are required for the replica exchange procedure. The *task interaction* can be described as *unstructured*. Following the SPMD approach results in a *synchronous communication model*, where processes are required to be in the same state, e.g., to execute the replica exchange procedure. Moreover, the application makes use of collective communication to broadcast updated temperatures to all processes. The size of *associated data* (e.g., replicas) is rather *small*.

4.5.2.2 Cloud Readiness

Static task generation is applied (e^-). Moreover, each process gets assigned a task at the beginning of the program. This is also done statically (e^-, i^-). The program is not able to make use of new resources or to free existing resources. Thus, it is hard to include elasticity support for this application.

Table 4.1: Parallel application properties of exemplary applications and their impact on cloud readiness

Design Decision	SPMD-based MPI application	MPMD-based MPI application	Work Queue-based application
Task Generation	Static (e^-)	Dynamic (e^+)	Dynamic (e^+)
Task Mapping	Static (e^-, i^-)	Cent. Dyn. (e^+, i^+)	Cent. Dyn. (e^+, i^+)
Task Interaction	Unstructured (l^-)	Structured (l^+)	Structured (l^+)
Communication Model	Synchronous (l^-)	Synchronous (l^-)	Asynchronous (l^+)
Size of Associated Data	Small (d^+, e^+, i^+)	Small (d^+, e^+, i^+)	Small (d^+, e^+, i^+)

Furthermore, the owner computes rule is employed, which leads to application state. Consequently, none of the participating processes can be designed as stateless. Unstructured task interactions (l^-) lead to changing communication partners with respect to the point-to-point primitive (cf. Figure 4.3), which avoids cloud-specific optimizations. The computation proceeds in a stepwise manner by employing synchronous, blocking primitives (l^-). Thus, each pair of processes is tightly coupled for the replica exchange procedure and all processes are required to participate in the broadcast to update temperature information. A single straggler process can thus slow down the whole application. The small size of associated data affects the cloud readiness positively (d^+, e^+, i^+).

Automated Management: MPI-1 does not provide means to dynamically add or remove processes at runtime. As a result, failed processes cannot be replaced. Furthermore, failing processes force the whole application to stop execution. After a single failure, the whole application has to be restarted. This is especially negative for long running or large-scale simulations.

4.5.3 MPMD-based MPI Application

This application is implemented based on MPI-2, which introduces dynamic process management and thus supports a varying number of MPI processes

[GTL99]. Moreover, it employs the Multiple Program Multiple Data (MPMD) parallel programming style [Fos95]. MPMD executes different programs for different MPI processes. In this case, two different programs are used: One for a coordinator and one for workers. Based on these programs, every worker process (WP) executes the simulation and the coordinator process (CP) controls each step by starting an iteration, collecting the simulation results from each worker process, and preparing the next iteration. Thus, the coordinator process executes the replica exchange procedure based on the collected results locally, generates new tasks, and sends these tasks to the worker processes (cf. Figure 4.3).

4.5.3.1 Parallel Application Profile

The task decomposition applied here is data decomposition because each task executes the same procedure for a replica using a different temperature. The task definition in this context differs from the SPMD-based MPI application. Here, a task is defined as a simulation based on a specific replica and a specific temperature for a specific step. The coordinator process collects all simulation results after each step and subsequently generates tasks for the next step. Thus, task generation is *dynamic* and task mapping is *centralized dynamic*. Task dependencies are managed by the coordinator process. *Task interactions* can be described as *structured*. All worker processes communicate with the coordinator process after each step. The application relies on point-to-point communication and uses asynchronous send and synchronous receive operations. This leads to a *synchronous communication model*, where the coordinator process waits for the results of all worker processes in a blocking manner. The size of *associated data* is *small*.

4.5.3.2 Cloud Readiness

Task generation as well as task mapping is dynamic (e^+) or centralized dynamic (e^+, i^+), respectively. At the beginning of each iteration, new worker processes can be added or removed by means of MPI-2 primitives. This

enables elasticity actions after each iteration. It depends on the execution time of an iteration how fast the application can make use of available worker processes and how fast existing ones can be released. Furthermore, worker processes receive all required information from the coordinator process in each step. Thus, they are designed as stateless components. Only the coordinator process stores simulation results of each step. This also leads to structured task interactions (l^+). Synchronous receive operations are used, which leads to tightly coupled processes (l^-). A single straggler process can thus slow down the whole application. The small size of associated data affects the cloud readiness positively (d^+, e^+, i^+).

Automated Management: MPI-2 allows adding and removing processes at runtime, but failed processes cannot be replaced by the application as the corresponding management is missing. Besides, synchronous blocking receive operations force the whole application to wait infinitely long for results of failed worker processes. In this case, the whole application has to be restarted. Moreover, MPI-2 does not provide the means to implement elasticity management. Implementing the corresponding management actions would improve the cloud readiness of the application. The authors of [RRC+16; RRR+18] provide such an approach that transforms iterative MPI-based applications into automatically managed, elastic applications.

4.5.4 Work Queue-based Application

The third application is a parallel application based on the Work Queue framework [BRA+11; RCIT11]. The Work Queue framework is designed for scientific ensemble applications and provides a master/worker architecture with an elastic pool of workers. In contrast to the MPMD-based MPI application, fault tolerance is provided by handling worker failures in an automated manner.

This application makes use of the master/worker architecture. The master (M) generates the configuration and input files required for each simulation step. These files are then farmed out to a set of workers (W) for parallel simulation. After each step, the master collects the output files generated by

each worker, tries to exchange replicas, and generates the configuration and input files for the next step (cf. Figure 4.3). The master thus coordinates the simulation of multiple workers that can be distributed across compute nodes.

4.5.4.1 Parallel Application Profile

The task decomposition applied here is also data decomposition. The generation of these tasks is *dynamic* as new tasks are generated in each step. Similarly, *centralized dynamic task mapping* is applied to distribute the tasks across workers. The *task interaction* can be described as *structured*. The *communication model* is *asynchronous*. The application does not make use of collective communication, which enables dynamic adaptation. Work Queue automatically restarts failed tasks and does not block upon worker failures. Nevertheless, each logical step is finalized by the master. The size of *associated data* is *small*.

4.5.4.2 Cloud Readiness

The Work Queue framework underlying the application automatically provides an elastic worker pool. Tasks are dynamically generated (e^+) and mapped to newly added workers (e^+, i^+). A worker can be removed easily whenever it finished a task. Furthermore, workers receive all required information from the master. Thus, they are designed as stateless components. Only the master stores the simulation results of each step. Additionally, this leads to structured task interactions (l^+).

Tasks are farmed out and simulation results are collected in an asynchronous manner (l^+). However, each iteration requires finalization by the master. As an optimization to this barrier after each iteration, the master can be designed to execute two logical iterations concurrently. This is possible because only simulation results of neighboring workers dependent on each other. In case of a single delayed task executed on a straggling worker, the submission of only one other task has to be delayed by the master. All other

tasks can be generated and mapped to the available workers. Following this approach, the resulting idle time can be used for additional system tasks, e.g., to execute a failed task again. This optimization speeds up the whole application and is specifically useful for large-scale deployments with many worker instances. The small size of associated data affects the cloud readiness positively (d^+, e^+, i^+).

Automated Management: Elasticity actions can be executed more easily as workers are designed as stateless components. Newly added workers automatically receive tasks from the master and failed workers do not force the whole application to halt. Therefore, the framework tracks the execution of tasks and automatically restarts lost tasks.

4.5.5 Discussion

In this case study, two parallel applications based on technology stacks that are widely adopted in the HPC community as well as one application based on a novel execution framework have been analyzed. Referring to Table 4.1, the cloud readiness of the parallel applications assessed increases from left to right. The impact on cloud readiness is shown for each parallel application property by displaying the conceptual fitness of the corresponding property mappings (cf. Section 4.4).

The results of the assessment enable a comparison of different applications with respect to the parallel design decisions made and their corresponding cloud readiness. Moreover, the overview in Table 4.1 ensures the identification of sweet spots to optimize their cloud readiness.

Based on these results, further decisions with respect to cloud migration can be made. Beyond design-relevant aspects, these decisions have to consider further constraints including the benefits gained and the money spent to optimize a specific parallel application. For example, it might not be worth the effort to optimize an existing MPI-1 application if it is short-running and relies on specific libraries that are not available for other frameworks. Such an application might be operated in HPC-aware cloud environments without making use of elasticity. On the other hand, one might decide to adopt a

newer version of MPI that provides dynamic process management as a first step towards elastic scaling.

Since all applications are designed for REMD, this shows that parallel application design provides several degrees of freedom to trade parallel performance for cloud readiness. For example, Figure 4.3 illustrates that centralized dynamic task mapping, which improves the cloud readiness of a parallel application, on the other hand leads to idle time, which in turn decreases parallel efficiency. In this case, one accepts a decreased parallel performance to ensure an application's cloud readiness.

4.6 Related Work

Whereas cloud environments have been compared to HPC clusters and supercomputers [JRM+10], a comparison of characteristics of applications developed for these environments is missing. In this chapter, such an approach is presented that supports the assessment of parallel applications with respect to their cloud readiness. The proposed approach thus provides decision support by formalizing parallel application properties resulting from parallel application design and mapping these properties to cloud-specific application characteristics.

Designing parallel applications is a complex task involving human effort that cannot be automated. However, pattern languages for parallel computing support the creative process of application design. The OPL[1] pattern language, which has been created by merging two existing approaches [KMMS10; MMS07], represents the most promising approach towards pattern-oriented parallel application design. It has been conceptualized from a pure parallel computing perspective and the design goal to be generally applicable. Designing parallel applications for the cloud is beyond the scope of existing pattern languages.

Several directives how to design parallel cloud applications are given in [RCIT11]. Whereas the authors apply these directives to an exemplary par-

[1] https://patterns.eecs.berkeley.edu.

allel application, they do not provide a systematic approach to handle design trade-offs in parallel application design. Similarly, the authors of [PARD13] analyze different types of applications with respect to their characteristics. They state that *alternate formulations* have to be found for these applications to benefit from cloud services. This chapter aims at guiding trade-off handling for parallel application design (e.g., in the context of cloud migration). It thus enables these alternate formulations that are required to benefit from cloud-specific properties.

4.7 Chapter Summary and Discussion

The understanding of implications of parallel design decisions on an application's cloud readiness is a key requirement for successful cloud migration. To support this understanding, a systematic approach to assess the cloud readiness of parallel applications is presented. This approach structures the multi-dimensional design space of constructing parallel applications for cloud environments based on application properties. As a result, parallel design decisions can be evaluated with respect to their effects on cloud readiness, which constitutes the basis for cloud migration and provides hints towards optimization of an application's cloud readiness.

The presented assessment procedure can also be applied to assess applications implemented based on TASKWORK (which is a runtime system described as part of Chapter 5) or the serverless farm skeleton (which is a programming abstraction described as part of Chapter 6) to gain insights into related design considerations and trade-offs. These design trade-offs are summarized and discussed in Section 5.4.10 and Section 6.2.6, respectively.

Chapter 5

Elastic Parallel State Space Search

In this chapter, the opportunities and challenges related to elasticity are discussed for applications based on the state space search technique. Well-known meta-algorithms based on state space search include branch-and-bound and backtracking search. Their computation and communication patterns are input-dependent, unstructured, and evolving during the computation [GRV95; SW03]. For parallelization, the state space is typically modeled as a tree to which the search process is applied. Therefore, such applications are called parallel tree search applications in the following.

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 [LW15]. 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 [PJ18], 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 adapting the number of processing units at runtime according to measured runtime metrics. This novel ability and related challenges are discussed to develop and operate elastic parallel tree search applications. Based on the findings, TASKWORK, a cloud-aware runtime system that provides a comprehensive foundation for implementing and operating elastic parallel tree search applications, and Equilibrium, an elasticity controller that dynamically adapts the number of processing units according to user-defined cost and efficiency thresholds, are presented. Additionally, a development framework for elastic parallel branch-and-bound applications is provided, which aims to minimize programming effort. An extensive evaluation of a prototypical elastic parallel system architecture based on the presented concepts has been conducted and the corresponding experiments in an OpenStack-based private cloud environment are discussed in detail.

This chapter is structured as follows. Section 5.1 describes the characteristics of parallel tree search applications and algorithms. In Section 5.2 the problem statement is outlined. Section 5.3 describes the conceptualization of a cloud-aware runtime system for elastic parallel tree search in the cloud. In Section 5.4, TASKWORK - an integrated runtime system for elastic parallel tree search applications - is presented. An elastic branch-and-bound development framework and its use is described in Section 5.5. Equilibrium - a reactive elasticity controller - is presented in Section 5.6. The results of an extensive experimental evaluation are presented in Section 5.7. Related work is discussed in Section 5.8. Section 5.9 concludes this chapter.

5.1 Parallel Tree Search, Algorithms & Applications

Tree search (also called tree traversal) refers to a processing technique for visiting nodes in a tree structure. Many advanced algorithms for state space search 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. 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 input-dependent and 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. 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 [KS92], biochemistry [SSTP09], electronic design automation [SBS96], finite geometry [Arc18], model checking [BBCR10], automotive product configuration [SKK03], financial portfolio optimization [BL09], production planning and scheduling [Ron05], as well as fleet and vehicle scheduling [Ral03].

The most common approach to make use of parallel processing is *exploratory parallelism*. Therefore, different subtrees of the search tree are explored in parallel by a set of (potentially distributed) processing units. By following a task-parallel approach, each task represents the traversal of the subtree rooted at a specific node of the tree. This approach is also visualized in Figure 5.1. 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

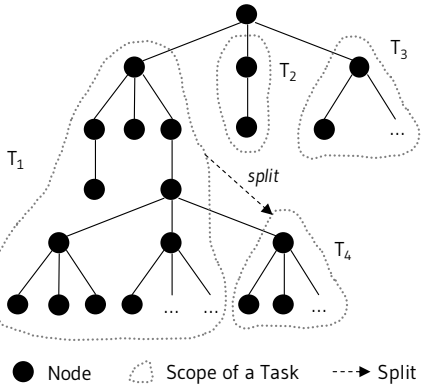

Figure 5.1: For parallelization, the search tree is cut into tasks, each capturing a subproblem of the initial problem. Because the search tree is dynamically constructed and its size and shape are not known a priori, new tasks have to be dynamically created by splitting an unexplored subtree from the search tree of an existing task.

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) as well as task sizes (defined by the shape and size of specific subtrees) are hard to predict upfront. Consequently, their computation and communication patterns are input-dependent, unstructured, and evolving during the computation [GRV95; SW03]. The irregular nature of these applications also constitutes the major source of parallel overhead and dramatically affects their parallel performance and scaling behavior.

5.2 Problem Statement & 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 [ABD+09]. 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. Consequently, it is hard to predict their execution time and scaling behavior. This also means that the monetary costs for solving a specific problem with a specific number of processing units are unknown a priori. 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. Section 3.1). 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 chapter, this novel opportunity of elastic scaling as well as the

related challenges are addressed for parallel tree search applications. It is discussed how to construct a runtime system for elastic parallel tree search and an elasticity controller that dynamically adapts the number of processing units according to user-defined cost and efficiency thresholds.

5.3 Constructing a Cloud-aware Runtime System

To benefit from cloud-specific characteristics, developing elastic parallel applications is a fundamental problem that has to be solved [GEM+16]. 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 this chapter specifically focuses on parallel tree search applications, which require dynamic task parallelism, the degree of logical parallelism can be defined as the current number of tasks. A cloud-aware runtime system is required that transparently controls task generation, load balancing, and task migration to ensure elastic scaling. Figure 5.2 shows the conceptualization of such a runtime system. It allows developers to mark

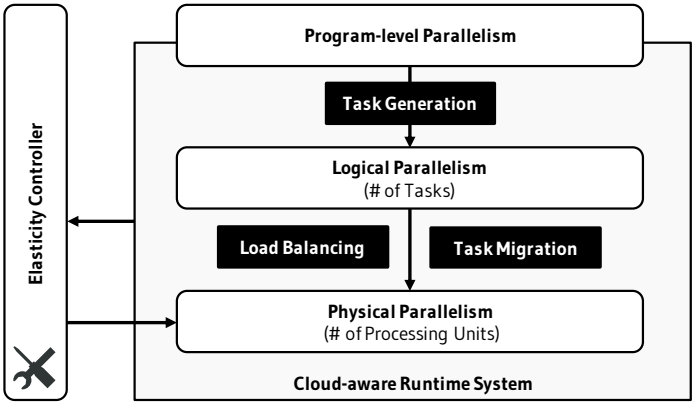

Figure 5.2: The described cloud-aware runtime system adapts the logical parallelism by generating tasks dynamically, handles load balancing and task migration, and thus enables elastic parallel computations.

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. Figure 5.2). The proposed approach is not limited to any specific cloud management approach or tooling: An elasticity controller may comprise any kind of external 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 Section 5.6.

Besides elasticity, the characteristics of cloud environments lead to new architectural requirements that have to be considered by parallel applications (cf. Chapter 4). Due to virtualization and resource pooling (leading to CPU time-sharing and memory overcommitment), fluctuations in process-

ing times of individual processing units can often be observed [GFG+16]. 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.

The following sections basically describe the components shown in Figure 5.2. In Section 5.4, a cloud-aware runtime system called TASKWORK, which is specifically designed for parallel tree search applications, is presented. Then, Section 5.5 outlines an elastic branch-and-bound development framework implemented on top of TASKWORK as well as its use by means of an exemplary application. Finally, a reactive elasticity controller called Equilibrium is described in Section 5.6.

5.4 Design and Implementation of TASKWORK

In this section, the design and implementation of TASKWORK, a cloud-aware runtime system specifically designed for parallel tree search applications according to the principles discussed in Section 5.3, is described. TASKWORK comprises several components that enable elastic parallel computations (cf. Ⓐ, Figure 5.3) and solve coordination problems based on ZooKeeper (cf. Ⓑ, Figure 5.3). Based on these system-level foundations, higher-level development frameworks and programming models can be built (cf. Ⓒ, Figure 5.3), which facilitate the implementation of elastic parallel applications. TASKWORK enables distributed memory parallelism by coordinating a set of distributed compute nodes based on the task pool execution model. The research prototype is implemented in Java.

First, ZooKeeper and the task pool execution model are described briefly before the components of TASKWORK are presented.

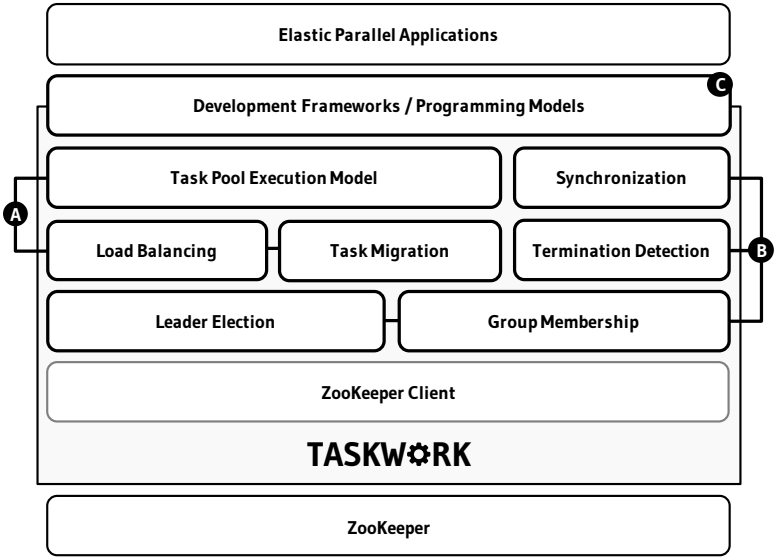

Figure 5.3: The components of TASKWORK enable elastic parallel computations based on the task pool execution model, solve coordination problems based on ZooKeeper, and support the construction of higher-level development frameworks and programming models.

5.4.1 ZooKeeper

ZooKeeper has been designed to ease the implementation of coordination, data distribution, synchronization, and meta data management in distributed systems [HKJR10]. Many prominent software projects rely on ZooKeeper including the Apache projects Hadoop[1] and Kafka[2]. It provides an interface that enables clients to read from and write to a tree-based data structure consisting of data registers called *znodes*, which can also be created as *ephemeral* or *sequential* znodes. Whereas ephemeral znodes only exist as long as the session that created the znode is active, sequential znodes have the value of a monotonically increasing counter appended to their name. Internally, data is

[1] http://hadoop.apache.org.
[2] https://kafka.apache.org.

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. ZooKeeper provides sequential consistency and guarantees writes to be atomic [HKJR10]. To execute multiple operations atomically, ZooKeeper also supports transactions. 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 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 [JR13]. 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. It is shown how to employ ZooKeeper to tackle these challenges.

5.4.2 Task Pool Execution Model

The task pool execution model [GGKK03] decouples task generation and task processing by providing a (distributed) 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 [PK08a; SB10; SBHD08]. The task pool execution model is employed as a foundation to enable elastic parallel computations according to the concepts depicted in Figure 5.2: The task pool manages tasks generated at runtime (establishing 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 often 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 a 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.

TASKWORK employs the distributed task pool execution model, which enables compute nodes to store generated tasks locally and, in general, provides a better scalability [PK08a]. 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, the distributed task pool execution model is enhanced with scalable coordination and synchronization mechanisms based on ZooKeeper.

5.4.3 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 [JR13]. 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.

5.4.4 Group Membership

As compute nodes might be added or removed at runtime for 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 [HKJR10]. Changes in group membership are obtained by all other compute nodes by watching the */computeNodes* znode.

5.4.5 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) [BJK+96]. Because sending tasks often leads to overhead, one typically favors work (task) stealing because communication is only required when a compute node runs idle. To trigger load balancing, changes in the local task pool are observed. 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, randomized task stealing [BL99] is employed. 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. Section 5.4.4).

5.4.6 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, 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 Section 5.5.2). TASKWORK employs weak migration of tasks. This means that a (user-defined) serialized state generated from a task object is transferred across the network. To facilitate the migration process, application-specific snapshotting mechanisms can be provided by developers.

5.4.7 Termination Detection

Traditionally, distributed algorithms (token- or weight-based) have been preferred for termination detection due to their superior scalability characteristics [GGKK03]. However, maintaining a ring (token-based approach) or tree (weight-based approach) structure across compute nodes in the context of an elastically scaled distributed system imposes significant overhead. To deal with this issue, a novel termination detection algorithm based on ZooKeeper's design principles is proposed.

Termination detection has to consider that tasks are continuously generated at any time and on any compute node in the system. However, an algorithm can make use of the fact that new tasks are always split from existing tasks, ultimately leading to a tree-based task dependency structure. The proposed termination detection algorithm employs this structure as follows: Every task in the system maintains a list of its children. During the life cycle of a task, the *taskID* of a child task is appended to this list when it is split from the parent task. Moreover, a global task list is used, which is updated whenever a task is completed. To initialize this global task list, the *taskID* of the root task is added to the list. At runtime, the global task list is updated for every completed task considering its *taskID* as well as the *taskIDs* of all its child tasks. Such an update procedure works as follows: Each *taskID* is either added to the global list if it is not contained in the global list or removed from the global list if it is contained. By following this update procedure, termination can be deduced from an empty global task list because all *taskIDs* are added and removed exactly once: Either by an update procedure triggered after the task has been completed or by an update procedure triggered by the completion of the task's parent task. To ensure that each *taskID* leads to exactly two updates, TASKWORK guarantees that tasks are completed exactly once. Based on this assumption, the termination detection algorithm ensures that the update procedure leads to exactly one create and exactly one delete operation per *taskID*.

In the following, a ZooKeeper-based implementation is described that is comprised of two parts: One for a coordinator that is determined by

the leader election component (cf. Algorithm 5.1) and one for compute nodes (cf. Algorithm 5.2). The notation of these algorithms is partially based on the asynchronous event-based composition model introduced in [CGR11] to model the interactions across TASKWORK's components. The global task list is stored in ZooKeeper and initialized with the *taskID* of the root task (cf. Algorithm 5.1, C-1). This is required because the root task has no parent task, i.e., the update procedure with its *taskID* is triggered only once: After the completion of the root task. At runtime, a finite number of update operations add or remove *taskIDs*. Termination can be deduced by the coordinator from an empty global task list. Therefore, a ZooKeeper *watch* is set to receive notifications on changes of the list: C-2 of Algorithm 5.1 triggers a termination event if all *taskIDs* have been removed.

Algorithm 5.2 implements the mechanisms to update the task list at runtime. An update is executed by the `TrackTasks` procedure, which is called by a worker thread whenever a task is completed (cf. Algorithm 5.2, CN-0). If a specific *taskID* is contained in the task list, the removal of this *taskID* is requested (cf. Algorithm 5.2, line 12). If a *taskID* is not contained, its creation is requested (line 14). To avoid false positive termination decisions, one has to ensure that the update procedure is executed atomically. ZooKeeper transactions are employed to enforce an atomic update of the global task list. This guarantees that termination cannot be detected due to an intermediate system state, in which the global task list is empty until the full update procedure (related to a single completed task) has been executed. As the transaction might fail, e.g., if a *taskID* should be created that has been created by another compute node before the transaction at hand gets processed by ZooKeeper, an asynchronous callback is passed to the *commit* operation that handles potential failures (cf. Algorithm 5.2, line 17). In the following, these potential failures are discussed.

The transaction has been constructed as a set of *create* and *delete* operations. Each *create* operation might fail. In case of such a failure, another compute node added the *taskID* before the local request was processed. Moreover, delayed reads might lead to a falsely constructed create operation. On the other hand, *delete* operations are executed exactly once because they

Algorithmus 5.1 Termination Detection - Coordinator

```
Task Pool, instance tp.
Application, instance app.
ZooKeeper Client, instance zk.
 1: upon event <Init> do                                              ▷ C-0
 2:     if !zk.exists('/globallist') then
 3:         zk.create('/globallist');
 4:     end if
 5:     if !zk.exists('/txnIDs') then
 6:         zk.create('/txnIDs');
 7:     end if
 8: upon event <tp, RootTaskEnqued | task> do                         ▷ C-1
 9:     zk.getChildren('/globallist');
10:     zk.create('/globallist/' + TaskID(task));
11: upon event <zk, Watch | w> do                                     ▷ C-2
12:     if Type(w) = NodeChildrenChanged then
13:         if zk.getChildren('/globallist') = ∅ then
14:             trigger <app, Terminated>;
15:         end if
16:     end if
```

are only requested if the corresponding *taskID* is contained in the list (cf. Algorithm 5.2, line 11-12). Additionally, connection losses might also lead to a failed transaction and, by default, handling connection losses is a non-trivial task with ZooKeeper [JR13]. There are cases, in which one cannot know if a transaction, which is interrupted by a connection loss, has been processed or not. Simply executing the transaction again could compromise the termination detection algorithm, finally leading to a false positive or missing termination event. In the following, it is described how the proposed algorithm deals with these issues.

CN-2 of Algorithm 5.2 shows the implementation of the `TxnCallback` procedure, which handles the failures described above. Therefore, transactions that fail are simply retried by calling the `TrackTasks` procedure again. To deal with connection losses, however, the presented algorithm has been enhanced to include an additional *create* operation in each transaction constructed by the `TrackTasks` procedure. This operation creates a unique

Algorithmus 5.2 Termination Detection - Compute Node

```
Worker Thread, instance wt.
ZooKeeper Client, instance zk.
 1: upon event <wt, completedTask | task> do                          ▷ CN-0
 2:     taskIDs ← TaskID(task) ∪ ChildTaskIDs(task);
 3:     TrackTasks(taskIDs, null);
 4: procedure TrackTasks(taskIDs, txnID)                              ▷ CN-1
 5:     txn ← zk.transaction();
 6:     if txnID = null then
 7:         txnID ← generateTxnID();
 8:     end if
 9:     txn.create('/txnIDs/' + txnID);
10:     for each taskID ∈ taskIDs do
11:         if zk.exists('/globallist/' + taskID) then
12:             txn.delete('/globallist/' + taskID);
13:         else
14:             txn.create('/globallist/' + taskID);
15:         end if
16:     end for
17:     txn.commit(TxnCallback, taskIDs, txnID);
18: end procedure
19: procedure TxnCallback(taskIDs, txnID)                            ▷ CN-2
20:     if failed ∨ lost connection then
21:         if !zk.exists('/txnIDs/' + txnID) then
22:             TrackTasks(taskIDs, txnID);
23:         end if
24:     end if
25: end procedure
```

znode */txnIDs/txnID*, where *txnID* is a globally unique ID generated for each transaction (cf. Algorithm 5.2, line 6-8). This allows us to check if a transaction has been processed or not in case of a connection loss. CN-2 of Algorithm 5.2 employs the *txnID* to avoid a repeated execution of committed transactions thus making them idempotent. Even if the *txnID* exists but one reads an outdated ZooKeeper server state (in which it seems to be nonexistent), the *txnID* passed to the `TrackTasks` procedure avoids a repeated commit thus finally maintaining a consistent system state. As ZooKeeper guarantees that clients read up-to-date values within a certain time bound,

retrying a commit of a transaction with an existing *txnID* stops eventually (cf. Algorithm 5.2, CN-2).

5.4.8 Synchronization of Global Variables

As discussed in Section 5.1, 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, two operations are provided: `getVar` for obtaining node-level variables and `setVar` for setting node-level variables. Whenever a node-level variable changes its value, ZooKeeper is employed 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.

5.4.9 Development Frameworks & 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 Section 5.3, developers only mark program-level paral-

lelism 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 Section 5.5.2).

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 Section 5.3, 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. Figure 5.2). 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 Section 5.3. 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. Section 5.4.5). 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. In many cases, on-demand task generation is more efficient because formerly generated tasks might contain a subtree that has already been proven to be obsolete (cf. Section 5.1). Thus, threshold-based task generation often results in unnecessary transferal of tasks over the network, leading to additional overhead.

5.4.10 Design Trade-offs

In this section, the design trade-offs made are discussed based on the assessment approach described in Chapter 4.

TASKWORK has been specifically designed for applications based on the state space search technique that make use of exploratory decomposition (cf. Section 5.1). The generation of tasks is *dynamic* as new tasks are generated at runtime by means of a task splitting mechanism (cf. Section 5.4.9). To distribute tasks across workers, *distributed dynamic task mapping* is applied in line with the distributed task pool execution model (cf. Section 5.4.2). The runtime system generates tasks (e^+), which are automatically mapped to newly added nodes by means of a load balancing mechanism (e^+). As a result, applications based on TASKWORK can be scaled out dynamically. Note that the distributed task pool execution model leads to stateful compute nodes (i^-), but enables a scalable parallel system (cf. Section 5.4.2). Additionally, a migration mechanism releases compute nodes that have been selected for decommissioning, thus compute resources can also be removed easily (cf. Section 5.4.6). Due to the irregular nature of these applications, task interactions are unstructured (l^-). TASKWORK performs all communication and synchronization tasks in an asynchronous manner (l^+). The small size of associated data affects the cloud readiness positively (d^+, e^+, i^+).

5.5 Elastic Branch-and-Bound Development Framework

In this section, a development framework for elastic parallel branch-and-bound applications is described 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 [ABD+09]. In the following, the branch-and-bound approach is briefly explained. Then, it is shown how to employ the framework to develop an example application.

5.5.1 Branch-and-Bound Applications

The branch-and-bound approach is explained 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 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 [CLRS09].

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. 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 due to combinatorial explosion. 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. In the following, the underlying principles, which make the search procedure efficient, are described.

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, the tour represented by this node is evaluated 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 [Sed84]. The TSP application used in the experimental evaluation (cf. Section 5.7) calculates the lower bound by adding the weight of a minimum spanning tree (MST) of the not-yet visited cities to the current path [AMM+18; Sed84]. The MST itself is calculated based on Prim's algorithm [Pri57]. One 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 feasible for large problems.

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 one maps 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 hard to predict. Consequently, solving the TSP requires the runtime system to cope with dynamic problem decomposition and load balancing to avoid significant idling of processing units. Every task that captures a specific subproblem can produce new child tasks. 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.

5.5.2 Design and Application of the Development Framework

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

TASKWORK provides a generic task abstraction that can be used to build development frameworks and programming models. In the context of branch-and-bound, one can 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 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. The task split off can be processed by another worker thread running on another compute node. To limit the number of tasks generated, one can make use of TASKWORK's potential splits, i.e, the `split` operation is only triggered, when new tasks are actually required. As depicted in Figure 5.4, 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, it is described 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. Figure 5.4). In this case, a task simply stops its execution. The migration process itself is handled by TASKWORK. This means that a

```
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 }
```

Figure 5.4: 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.

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. Section 5.4.6).

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 timely distribution of the current upper bound is essential to avoid excess computation (due to an outdated value). By employing the synchronization component (cf. Section 5.4.8), an update of the global upper bound is initiated whenever the local upper bound is better than the current global upper bound observed. Technically, an update rule is specified 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. Figure 5.4) are implemented based on the `getVar` and `setVar` operations (cf. Section 5.4.8).

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.

5.6 Elasticity Control with Equilibrium

Whereas TASKWORK enables applications to automatically adapt to a changing number of compute nodes (processing units), in addition, an elasticity controller is required that comprises the decision-making logic for adding and removing processing units at runtime. In this section, the opportunities and challenges related to elasticity are discussed in the context of parallel tree search applications. Based on the findings, Equilibrium - a reactive elasticity controller for elastic parallel tree search - is presented.

5.6.1 Opportunities and Challenges of Elasticity Control

In this section, it is analyzed and discussed 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.

5.6.1.1 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?" [SCHW17]. 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, one rather has 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 Section 5.1, the scaling behavior of parallel tree search applications depends on the processed problem and is hard to predict. Whereas this renders static resource provisioning impractical, one can continuously monitor a parallel system and adapt the number of processing units by means of elastic scaling to meet a user-defined target efficiency. Thus, 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 (cf. Section 3.2), which requires that the sequential execution time T_{seq} and the parallel execution time T_{par} are known (cf. Equation 3.3). 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. The challenge of defining an appropriate runtime metric that approximates the elastic efficiency and constructing a corresponding elasticity controller is addressed in Section 5.6.2.

5.6.1.2 Cost-based Elasticity Control

First, the monetary costs of executing a sequential tree search application in a cloud environment are discussed. 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:

$$C_{seq}(I) = T_{seq}(I) \cdot c_{\pi} = W(I) \cdot c_{\pi}, \tag{5.1}$$

where T_{seq} is the sequential execution time, W is the size of an input problem described by I, and c_{π} is the price for one processing unit per time unit.

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 $\frac{C_{seq}}{W}$ remain constant for sequential tree search applications. In the best case, this should also hold for parallel tree search applications. In the following, it is discussed 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.

The fundamental difference between sequential and parallel tree search is that parallel processing additionally introduces non-essential computational steps and idling steps (cf. Section 3.1). 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 (in form of non-essential computational steps and idling steps).

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 Equation 3.1. The costs per problem size ratio can be formalized as follows:

$$\frac{C_{par}(I, p(t))}{W(I)} = \frac{C_{par}(I, p(t))}{T_{seq}(I)} = \frac{T_{par}(I, p(t)) \cdot \bar{p} \cdot c_{\pi}}{T_{seq}(I)} \tag{5.2}$$

From Equation 3.3 and Equation 5.2 follows:

$$\frac{C_{par}(I, p(t))}{W(I)} = \frac{c_{\pi}}{E_{elastic}(I, p(t))} \tag{5.3}$$

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 one is 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. Equation 5.3). 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.

5.6.1.3 Key Findings & 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 significant advantage 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. Situations, in which one spends money for inefficiently used compute resources, can thus be avoided.

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.

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).

5.6.2 Design of a Reactive Elasticity Controller

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

The core idea of the proposed 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, the workload efficiency as defined in Section 3.1 (cf. Definition 3.4) is employed. To measure the workload efficiency at runtime, one can make use of the following observation: With respect to the task pool execution model (cf. Section 5.4.2), all three sources of overhead (idle time, communication, and excess computation) finally affect the percentage of time a processing unit allocates the CPU to do useful work. Therefore, the workload efficiency WE can be monitored as 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.

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, the corresponding scaling strategy is discussed.

A well-known problem of elasticity control mechanisms are oscillating effects [BBD+14], i.e., compute resources are continuously provisioned and decommissioned leading to high overhead and, as a consequence, to low cost efficiency. This problem can be avoided by stopping to add processing units after the target efficiency level has been reached. Typically, 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 (coarse-gained) 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).

To make use of this knowledge, the scaling strategy can be described as follows (cf. Algorithm 5.3). The computation is started with one processing unit. At runtime, the elasticity controller scales out by adding a configurable number of processing units, which is called the *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 is called 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, 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, is reached. At this point, the

Algorithmus 5.3 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
```

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, the initialization phase is considered to be 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, the parallel system is scaled in (by decommissioning processing units) as follows: Whenever the workload efficiency falls below the target efficiency level, a configurable number of processing units, which is called 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 is called the *scale-in latency* sl_{in}. Technically, the elasticity control mechanism works 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.

5.7 Experimental Evaluation

Several experiments have been conducted to evaluate TASKWORK and Equilibrium. First, the system architecture as well as the experimental setup is described. Second, three benchmark applications for parallel tree search are presented, which enable a systematic evaluation. Third, TASKWORK is evaluated with respect to its basic 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. Fourth, a novel experiment is discussed, which shows how elastic scaling operations affect the performance of an application. Finally, Equilibrium is evaluated according to a described evaluation method.

5.7.1 System Architecture and Experimental Setup

In this section, the elastic parallel system architecture as well as the experimental setup are described. 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 the proposed elasticity controller. In the following, all components and their relationships are explained in detail.

OpenStack[1] 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 only the capability to provision and decommission processing units (in form of VMs) on demand is required, also any other cloud environment can be used.

Parallel applications have to be constructed according to cloud-specific design principles to benefit from elasticity. To develop and operate parallel tree search applications, TASKWORK is employed. It also provides a development framework to implement elastic parallel applications (cf. Section 5.5). Based on the framework, application developers only mark potential parallelism in their programs while TASKWORK automatically manages the dynamic adaptation.

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 Section 5.6.1. Technically, the monitoring of required metrics is implemented based on code-level instrumentation. In this respect, 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 the application developer, which fosters the usability of the proposed approach. Collection and aggregation of metrics are performed by distributed moni-

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

toring agents and a global aggregator that supplies the elasticity controller with monitoring data. The monitoring interval can be specified according to application-specific requirements. Scaling operations, i.e., provisioning and decommissioning of VMs, are performed via OpenStack's API.

Compute nodes are operated on CentOS 7 virtual machines with 1 vCPU clocked at 2.6 GHz, 2 GB RAM, and 40 GB disk. All VMs are deployed in an 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).

5.7.2 Benchmark Applications

Evaluating parallel tree search applications is a hard task because for many applications work anomalies occur [GK99; LS84; LW86]. 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. Section 5.1) combined with simultaneous knowledge sharing (e.g., in form of bounds [GC94] or lemmas [SB10]) at runtime. As a result, the expanded search tree differs significantly in its size and shape, which renders a rigorous evaluation (by measuring multiple parallel runs) infeasible. To deal with this issue, three benchmark applications are employed to enable a systematic evaluation. These benchmark applications are described in the following.

Unbalanced Tree Search (UTS): Unbalanced Tree Search [OHL+07] is a commonly employed benchmark to evaluate task pool architectures for parallel tree search (for examples see [Arc18; BF17; PF18]). 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

Table 5.1: Unbalanced Tree Search (UTS) instances

Problem Instance	Random Seed r	Expected Value b	Depth d	Tree Size [# of nodes]
UTS_0	19	4	16	16922208327
UTS_1	19	4	17	67688164184
UTS_2	19	4	18	270751679750
UTS_3	29	5	16	195676745034

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 the measurements, several instances of the geometric tree type are employed, 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 [Arc18; BF17; PF18]. 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 5.1 shows the UTS instances employed for the measurements.

Generic State Space Search Application (GSSSA): GSSSA [HBK19] has been introduced to address the problem that one cannot control the degree of irregularity of UTS instances, 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

Table 5.2: 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

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 5.2 shows the GSSSA instances taken from [HBK19], which are employed for the measurements.

WaitBenchmark: This benchmark was taken from [SB10], 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:

$$Task_{parent}\{T_R\} \xrightarrow{\text{split}} (Task_{parent'}\{T_R - T_{child}\}, Task_{child}\{T_{child}\}). \quad (5.4)$$

Table 5.3: Performance measurements of TSP instances

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

5.7.3 Basic Parallel Performance

The basic parallel performance of TASKWORK is evaluated by measuring speedups and efficiencies for the TSP application implemented with the elastic branch-and-bound development framework. To evaluate the parallel performance, 5 randomly generated instances of the 35-city symmetric TSP were solved. 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 5.3 shows the results of the measurements with three application runs per TSP instance. As one can see, the performance is highly problem-specific.

5.7.4 Scalability

To evaluate the scalability of TASKWORK, two different measurement approaches are employed: First, the speedup is determined with the UTS benchmark, the GSSSA benchmark, and the WaitBenchmark for a fixed problem size and different numbers of processing units. Second, the so-called *scaled speedup* is determined with the WaitBenchmark. The scaled speedup of a parallel system is obtained by increasing the problem size linearly with the number of processing units [GGKK03].

First, the scalability of TASKWORK with a fixed problem size, which is thus independent of the number of processing units, is measured. Figure 5.5

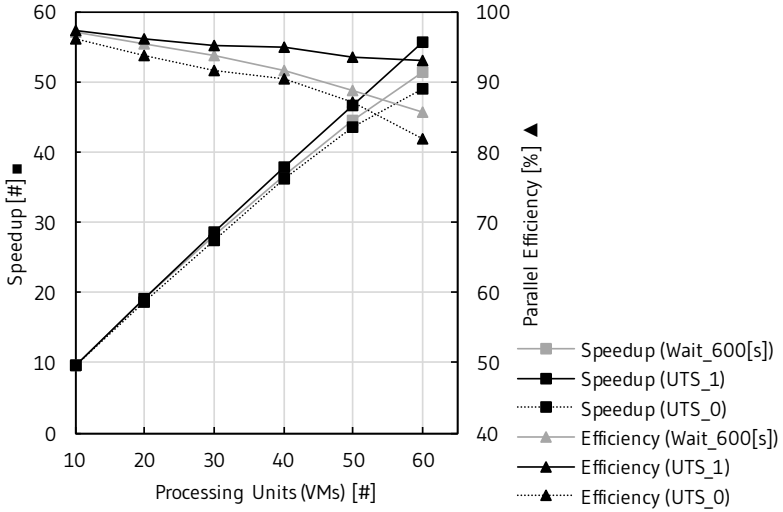

Figure 5.5: The problem instances shown are UTS_0, UTS_1, and the WaitBenchmark with an initial wait time of the root task of $T_{init} = 600$ [s]. Speedups and efficiencies given are arithmetic means based on 3 application runs.

depicts the results of the measurements for the two UTS instances UTS_0 (r=19, b=4, d=16) and UTS_1 (r=19, b=4, d=17) (shown in Table 5.1) and a problem instance of the WaitBenchmark with a fixed initial wait time of the root task of $T_{init} = 600$ [s]. As one can see, the figure shows close to linear speedups for all three problem instances. As expected, better speedups are obtained with larger problem instances. Figure 5.6 depicts the results of the measurements for the three GSSSA instances shown in Table 5.2. As described in Section 5.7.2, the GSSSA benchmark allows to control the degree of irregularity of generated instances, which influences an application's scaling behavior. As one can see in Figure 5.6, the scaling behavior of all three GSSSA instances is different, with $GSSSA_C$ leading to the worst speedups. This can be explained by the high irregular workload fraction w_i (cf. Table 5.2).

Second, the scalability of TASKWORK is measured with a problem size

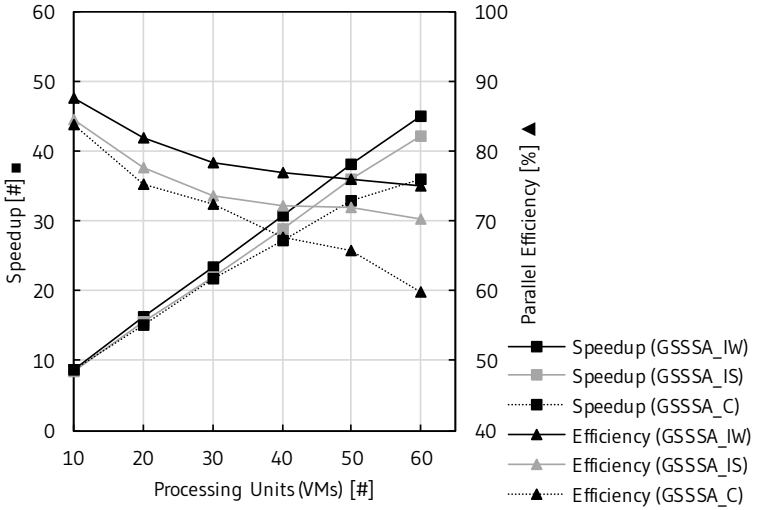

Figure 5.6: The problem instances shown are GSSSA_{IW}, GSSSA_{IS}, GSSSA_{C} as defined in Table 5.2. Speedups and efficiencies given are arithmetic means based on 3 application runs.

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 required 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 the measurements, an initial wait time of the root task of $T_{init}(p) = p \cdot 60$ [s] is defined (thus that $T_{init} = 600$ [s] for $p = 10$). The speedups and efficiencies obtained are depicted in Figure 5.7. The results of the measurements show close to linear speedups. As expected, the (scaled) speedup curve is much better compared to the one obtained by the scalability measurements with $T_{init} = 600$ [s], which are again depicted in Figure 5.7.

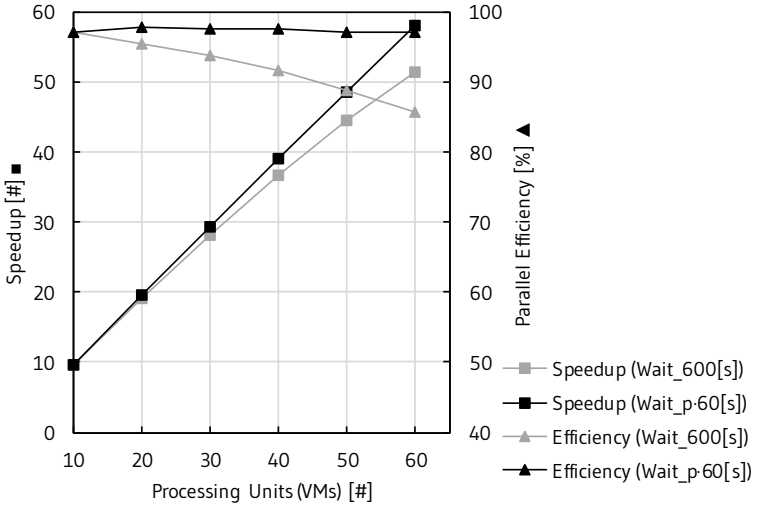

Figure 5.7: The problem instances shown are the WaitBenchmark with an initial wait time of the root task of $T_{init}(p) = p \cdot 60$ [s] and the WaitBenchmark with an initial wait time of the root task of $T_{init} = 600$ [s]. Speedups and efficiencies given are arithmetic means based on 3 application runs.

5.7.5 Overhead of 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. Section 5.3). A fundamental question that arises in this context is: *How fast can resources be effectively employed by the application?* This is a novel perspective on parallel system architectures that also has to be considered for evaluation.

A novel experimental method is proposed 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, the proposed method evaluates the effects of dynamic resource adaptation on performance in terms of the (elastic)

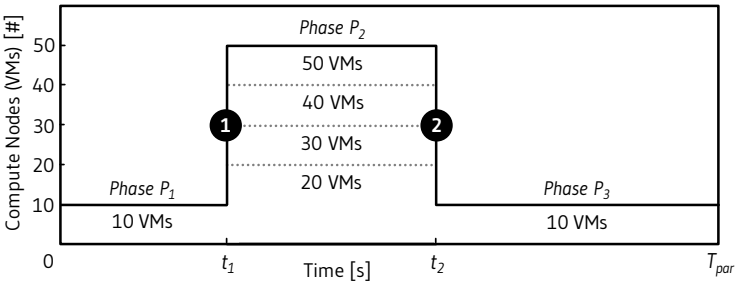

Figure 5.8: The number of compute nodes (physical parallelism) is adapted at t_1 and t_2 to measure the effects on elastic speedup $S_{elastic}$.

speedup metric. The experiment is described in Figure 5.8 and comprises three phases. The application is started with 10 compute nodes (VMs) in Phase P_1. At time t_1, the parallel system is scaled out by adding more VMs to the computation. To evaluate the elastic behavior without platform-specific VM startup times, VMs that are already running are employed. At time t_2, the VMs added at t_1 are decommissioned. 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. Section 5.4.6). One can easily see if newly added compute resources contribute to the computation by comparing the elastic speedup $S_{elastic}$ (speedup with elastic scaling) with the *baseline speedup* $S_{baseline}$ that is determined for a static setting with 10 VMs. To see how effectively new compute resources are employed by TASKWORK, several durations for Phase P_2 as well as different numbers of added VMs (cf. Figure 5.8) are tested, for each of which the percentage change in speedup S_{change} can be calculated as follows:

$$S_{change} = \frac{(S_{elastic} - S_{baseline})}{S_{baseline}} \cdot 100 \tag{5.5}$$

S_{change} allows us to quantify the relative speedup improvements based

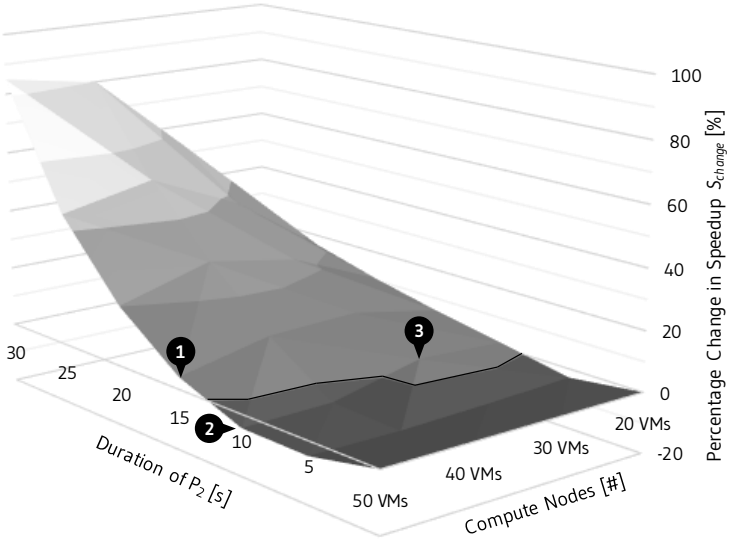

Figure 5.9: The percentage change in speedup S_{change} is calculated based on different durations of Phase P_2 and different numbers of compute nodes (VMs) added to the parallel computation at runtime. The number of VMs shown is the total number of VMs employed in Phase P_2.

on elastic scaling. Both $S_{elastic}$ and $S_{baseline}$ are arithmetic means calculated based on three application runs.

For the measurements, the TSP application implemented based on the elastic branch-and-bound development framework is employed. To avoid work anomalies, pruning is disabled to evaluate elastic scaling. All measurements are based on a TSP instance with 14 cities. The results of the measurements are depicted in Figure 5.9, 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 seconds (cf. Figure 5.9, ❶). Higher speedup improvements can be achieved by increasing the duration of Phase P_2. One can also see that for

a duration of only 10 seconds, adding 40 VMs even leads to a decrease in speedup (cf. Figure 5.9, ❷) whereas adding 20 VMs leads to an increase in speedup for the same duration (cf. Figure 5.9, ❸). 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 platform-specific provisioning times also affect the percentage change in speedup. For instance, containers can be provisioned faster than VMs. However, in this section elastic scaling of the runtime system (and an exemplary application) is evaluated in isolation. A holistic experiment that also includes VM provisioning overhead is discussed in Section 5.7.6.

5.7.6 Elasticity Control

Up to this point, TASKWORK has been evaluated in isolation to analyze its performance, scalability, and the overhead of elastic scaling operations. In this section, the whole elastic parallel system architecture, as initially discussed in Section 5.3 and technically described in Section 5.7.1, is evaluated. The evaluation method used is described in Section 5.7.6.1. The results are presented and discussed in Section 5.7.6.2.

5.7.6.1 Evaluation Method

The 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, the performance of several example problems with different degrees of irregularity (leading to a different scaling behavior) is determined and the results are discussed. 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. On this basis, three important metrics to evaluate the

elasticity controller can be calculated: (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:

$$\delta_{overall} = \frac{|E_{elastic} - E_{target}|}{E_{target}} \cdot 100 \tag{5.6}$$

(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:

$$\delta_{scale} = \frac{|WE_{total} - E_{target}|}{E_{target}} \cdot 100 \tag{5.7}$$

(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 the proposed code instrumentation and monitoring approach:

$$\delta_{approx} = \frac{|WE_{total} - E_{elastic}|}{E_{elastic}} \cdot 100 \tag{5.8}$$

To evaluate the reliability of the implemented elasticity controller, different application runs for the same input problem are compared in terms of the scaling actions executed. Finally, the performance of the elastic parallel system is compared to 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, the performance of the elastic parallel system is measured 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 Section 5.4. 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 affect the provisioning time.

5.7.6.2 Measurements & Results

To show that meeting a specific workload efficiency effectively approximates the elastic efficiency, the performance of three application runs is determined for each UTS instance depicted in Table 5.1. For the experimental evaluation, the elasticity controller is configured with a target efficiency of 95.0%. The results of the measurements (in line with the evaluation method described in Section 5.7.6.1) are shown in Table 5.4 and discussed in the following.

As Table 5.4 reveals, the 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, one 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, the GSSSA instances shown in Table 5.2 are employed. As described in Section 5.7.2, 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. The results of the elasticity measurements are discussed in the following.

Table 5.4: 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 5.5: 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	

Table 5.6: 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	

Table 5.5 shows the elasticity measurements for the three GSSSA instances shown in Table 5.2. The structure of the table is identical to the one of Table 5.4. Note that, due to the limited scaling behavior of the GSSSA instances, a target efficiency of 95.0% cannot be reached with more than one processing unit. To enable the evaluation, the elasticity controller is configured with a target efficiency of 80.0% for processing the GSSSA instances. As one can see in Table 5.5, 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}. This matches the results shown in Figure 5.6 and discussed in Section 5.7.4.

To evaluate the reliability of the elasticity controller, three different application runs for the same input problem are compared with respect to the scaling actions executed. For this purpose, the GSSSA_{IS} instance is selected because it requires the highest number of dynamic adaptations. Figure 5.10 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 depicted in the figure, towards 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 (due to the decreasing task granularity).

The dynamic adaptation of the number of processing units results in additional runtime overhead, which is assessed by comparing the performance

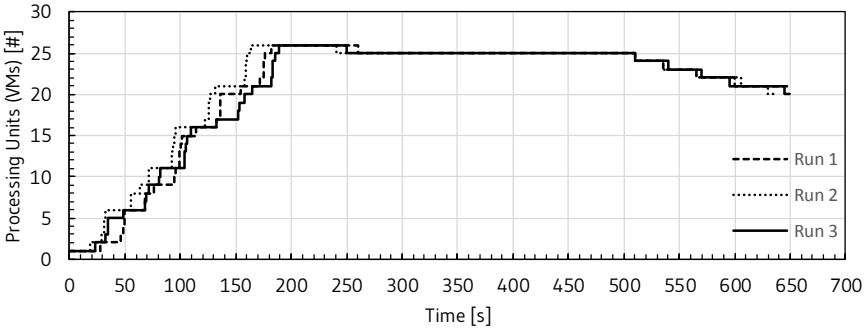

Figure 5.10: The reliability of the elasticity controller is evaluated by comparing the three application runs with the GSSSA_{IS} problem instance shown in Table 5.5 with respect to the processing units employed over time. The functions shown visualize $p(t)$ for each application run.

measurements of the elasticity experiments with the GSSSA instances (cf. Table 5.5) with the performance determined for a static setting with a fixed number of processing units p close to $\bar{p}$. The results obtained are depicted in Figure 5.11 and discussed in the following. Whereas an elastic efficiency of 78.59% on average is achieved for GSSSA_{IW} with ~ 27 processing units in the elastic setting, a parallel efficiency of 80.00% can be determined for a static setting with $p = 27$. For GSSSA_{IS}, an elastic efficiency of 73.42% on average is achieved with ~ 21 processing units in the elastic setting and a parallel efficiency of 74.50% is determined for a static setting with $p = 21$. For GSSSA_{C}, an elastic efficiency of 77.27% on average is achieved with ~ 17 processing units in the elastic setting and a parallel efficiency of 77.62% is determined 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.

To evaluate the elasticity control mechanism in isolation, the overhead of VM provisioning and decommissioning has not been included in the

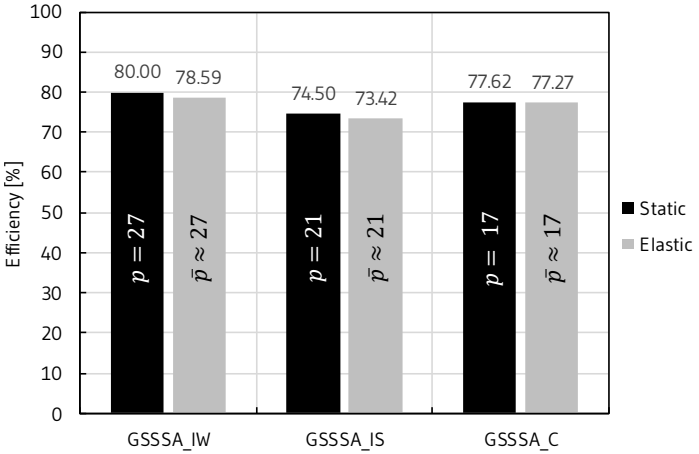

Figure 5.11: To assess the overhead related to the dynamic adaptation of processing units, the performance determined in the elasticity experiments (cf. Table 5.5) is compared to the performance of the same parallel system employing a static number of processing units.

elasticity measurements so far. For the measurements shown in Table 5.4 and Table 5.5 running VMs have been employed 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, the elasticity experiments have also been executed including the VM provisioning overhead. Table 5.6 shows the corresponding measurements for the three GSSSA instances shown in Table 5.2. The results obtained are similar to the ones shown in Table 5.5 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 and monitoring is only slightly higher.

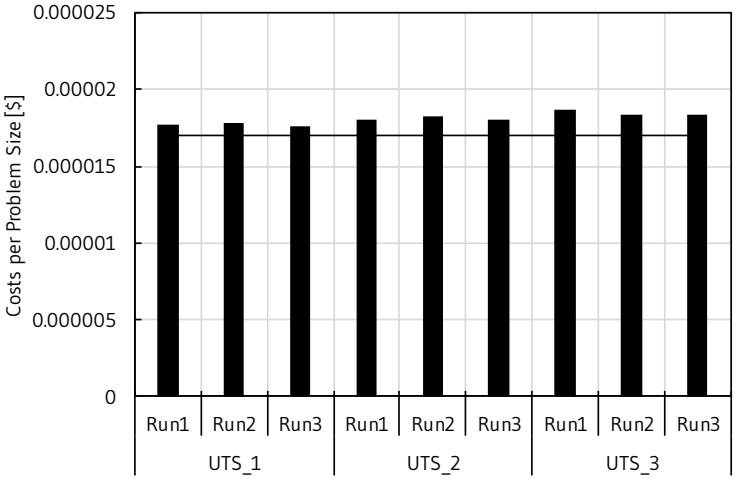

Figure 5.12: Comparison of the costs per problem size of all UTS instances shown in Table 5.1, which are calculated based on Equation 5.2 and the measurements

As shown in Section 5.6.1, 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, it is shown that meeting a target efficiency, as a consequence, results in the same costs per problem size ratio for all processed problem instances. Therefore, the costs per problem size are calculated for all UTS and GSSSA instances based on Equation 5.2 and the measurements. A price for one processing unit per time unit $c_\pi = \$0.000017/s$ is assumed. This value was taken from Google Compute Engine[1], where, at the time of writing, a VM with 1 vCPU costs $\$0.0612/h = \$0.000017/s$[2] (region *europe-west3*).

The costs per problem size ratios are shown in Figure 5.12 for all UTS instances and in Figure 5.13 for all GSSSA instances. As one 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

[1] https://cloud.google.com/compute.
[2] Google Compute Engine provides a per-second billing model.

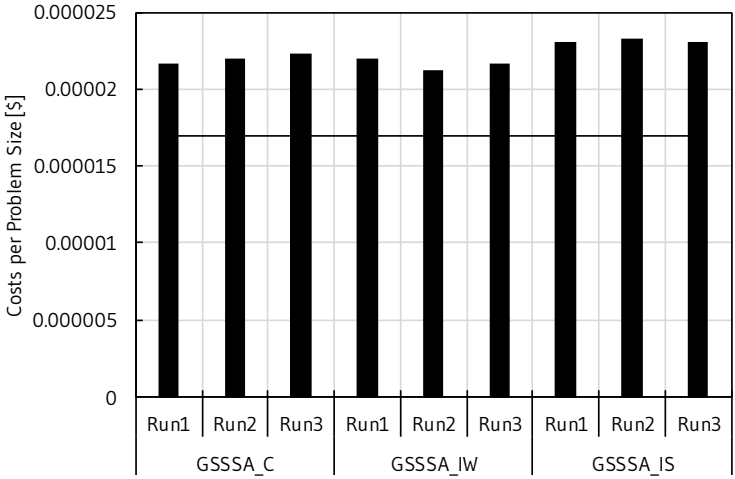

Figure 5.13: Comparison of the costs per problem size of all GSSSA instances shown in Table 5.2, which are calculated based on Equation 5.2 and the measurements

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 Section 5.6.1. 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.

5.8 Related Work

Whereas task-based parallelism has been originally designed to exploit shared memory architectures by systems such as Cilk [BJK+96], it also provides attractive benefits beyond shared memory architectures. In this chapter, specifically parallel tree search applications are addressed. These applications have extensively been considered by related work, discussing their exe-

cution with respect to different environments including clusters [AMM+18; AMS+17; BKLW99; BMW98; Ral03] and grids [ABGL02]. 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 [PK08a], a skeleton for branch-and-bound applications is presented, which supports parallel execution based on MPI. The authors of [BF17] 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 [SB10] discuss the challenges related to parallel tree search in the context of a distributed parallel satisfiability solver. The authors of [AMM+18] discuss the problem of replicable parallel performance of branch-and-bound applications and propose a skeleton that preserves the search order by distributing work in an ordered manner. COHESION is a microkernel-based platform for desktop grid computing [BDS06; SBHD08] 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 [VD14]. Processing units reachable with a lower network latency are preferred for stealing operations. In this chapter, cloud environments are addressed and it is shown how elasticity can be beneficially employed in the context of parallel tree search. More specifically, an approach to control the number of processing units at runtime according to user-defined cost and efficiency thresholds is presented.

In [HBK19], 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 function, thus allowing cost-based elasticity control. Note that one has to adopt the

concept of opportunity costs to make use of this optimization technique. On the contrary, this chapter presents a more general approach to adapt the number of processing units according to a user-defined target efficiency or costs per problem size ratio.

5.9 Chapter Summary and Discussion

This chapter tackles the challenge of developing and operating elastic parallel tree search applications. Related system-level challenges are discussed and it is shown how to enable elastic parallel computations as well as cloud-aware coordination of distributed compute resources. On this basis, a novel runtime system is presented 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. Additionally, a reactive elasticity controller is proposed that handles the cost/efficiency-time trade-off (cf. Section 3.1) by adapting the number of processing units according to user-defined configurations.

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 is 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 execution time limit 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 chapter 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 execution time limit, 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. When the time limit is exceeded, the currently known best solution of the optimization problem is returned to the user[1], 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. 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 [GGH12]. 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.

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

Chapter 6

Serverless Parallel Processing

Serverless computing is an emerging cloud computing paradigm that frees users from resource management issues. Therefore, serverless computing platforms enable the execution of user code in form of stateless FaaS functions (cf. Section 2.2.1). Compute resources are provisioned on-demand and scaled in an automated manner leading to two fundamental benefits: Elasticity by design and per-function resource accounting (and billing). Whereas FaaS functions are stateless, serverless computing platforms also provide backend services to store data [JSS+19]. Exemplary applications of serverless computing include data filtering and transformation, log file analysis, or object recognition in images [JSS+19]. In all these cases, computations are triggered by an event (e.g., a user request) and can be executed independently of each other. This enables these applications to benefit from elastic auto-scaling in a straightforward manner.

More recently, serverless computing platforms have become of interest for parallel applications, which comprise most often complex coordination, com-

munication, and synchronization patterns [JPV+17; SKP+18; WKK+18]. Several challenges related to the development and operation of parallel applications arise from the specific characteristics of serverless computing platforms (e.g., how to implement communication based on shared backend services). Additionally, pay-per-use and elasticity are fundamentally new concepts that have to be considered: As discussed in Chapter 3, parallel processing leads to a cost/efficiency-time trade-off that has to be dealt with.

In this chapter, the concept of *serverless skeletons* is introduced to enable parallel cloud programming, which separates functional application development from non-functional aspects of the execution. Additionally, a proactive elasticity control mechanism is discussed that is able to adapt the physical parallelism (in form of FaaS functions) per application run according to a user-defined goal, e.g., a specific execution time limit after which the result of the computation needs to be present while minimizing the monetary costs of the computation. Serverless skeletons are based on the concept of algorithmic skeletons [Col91; GC11], which has been introduced to structure parallel computations as a set of higher-level functions that abstract from complex coordination patterns inherent to parallel processing. It is shown how to use serverless skeletons to transparently ensure parallel coordination and communication based on serverless computing platforms while developers are able to implement functional code without considering parallelism and resource management. To validate this approach, the design and implementation of a serverless farm skeleton as well as a prototypical development and runtime framework based on Apache OpenWhisk is described. The serverless farm skeleton is evaluated experimentally with two example applications: Numerical integration and hyperparameter optimization, which is a commonly applied technique in machine learning. It is reported on performance measurements for both applications and shown how to consider a user-defined execution time limit by adapting the number of processing units in a proactive manner.

This chapter is structured as follows. Section 6.1 presents the novel approach to parallel cloud programming with serverless skeletons. A development and runtime framework for the serverless farm skeleton as well as

a corresponding prototypical implementation is described in Section 6.2. In Section 6.3, the implementation of two example applications, which are also employed in the experimental evaluation, is described. Section 6.4 describes how to construct a proactive elasticity controller for serverless parallel applications. Performance measurements are given in Section 6.5. Related work is discussed in Section 6.6. Finally, Section 6.7 summarizes and concludes this chapter.

6.1 Parallel Cloud Programming with Serverless Skeletons

Skeletons (cf. Section 2.1.1) capture common parallelism patterns and provide abstract implementations of these patterns for a parallel execution environment. In this chapter, specifically serverless computing platforms are addressed - a novel parallel execution environment with benefits such as per-function resource accounting. However, in contrast to other parallel execution environments, the specific characteristics of serverless computing platforms make the implementation of parallel coordination as well as the communication across parallel processing units challenging. In the following, several concepts and design principles to enable parallel cloud programming with serverless skeletons are discussed.

6.1.1 Skeleton-based Development

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

By following the serverless computing paradigm, parallel coordination has to be implemented based on backend services. This requires particular attention because the required consistency guarantees might not be provided by all backend services. Several examples of user and framework functions are discussed in more detail for a serverless farm skeleton (cf. Section 6.2.1).

6.1.2 FaaS Function Topology Mapping

For parallel execution, each skeleton instance requires a specific number of FaaS functions that have to be deployed to the target serverless computing platform. To create these FaaS functions, one has to map user and framework functions to FaaS functions, i.e., functions as seen by the serverless computing platform. As shown in Figure 6.1, it is distinguished between a *skeleton function topology* and a *FaaS function topology*. Each skeleton has a specific skeleton function topology consisting of connected user and framework functions, which is mapped to a FaaS function topology for deployment. Mapping several user / framework functions to the same FaaS function means that they are executed sequentially by a single FaaS function. This separation into two topologies enables flexibility with respect to the deployment of a skeleton instance: On the one hand, by mapping each user / framework function to an independent FaaS function, developers have the ability to fine-tune the resource requirements of each individual user and framework function, which can be scaled-out independently. Additionally, because each FaaS function is operated in a separate container, this enables the ability

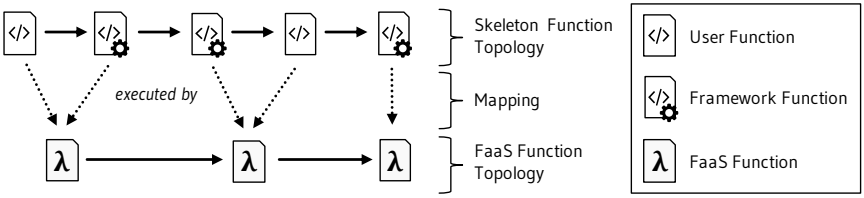

Figure 6.1: For deployment, the skeleton function topology of a skeleton instance has to be mapped to a FaaS function topology.

to isolate individual functions by means of container virtualization. On the other hand, by grouping user / framework functions and mapping them to the same FaaS function, developers can minimize various sources of overhead. Two major sources of overhead are (1) system overhead because, technically, a container is started for each FaaS function and (2) communication overhead because FaaS functions communicate via backend services as described in the following. Several mappings to FaaS function topologies and their characteristics are discussed in the context of the serverless farm skeleton in Section 6.2.2.

6.1.3 Communication via Backend Services

Whereas user and framework functions that have been mapped to the same FaaS function can communicate via shared memory, communication across FaaS functions requires additional effort. Because point-to-point communication is not supported on serverless computing platforms, communication has to be implemented based on shared backend services (cf. Section 2.2.1). To relieve developers of the burden of implementing and adapting code for communication via backend services, the required wrapper code can be automatically generated per FaaS function. By following this approach, the interaction with backend services as well as the serialization and deserialization of data is transparent to developers and provided by the generated wrapper code. The internal structure of a generated FaaS function is depicted in Figure 6.2. To support different backend services, a backend service access layer is introduced, which employs the adapter pattern [GHJV94]. Backend services can thus be selected based on application-specific requirements and easily replaced. The selection of backend services largely depends on the type and size of data structures stored by a serverless skeleton instance as well as their access frequency. In general, frequently accessed, small data structures benefit from in-memory data stores with low access latency, whereas for huge communication volumes object storage services are a good choice.

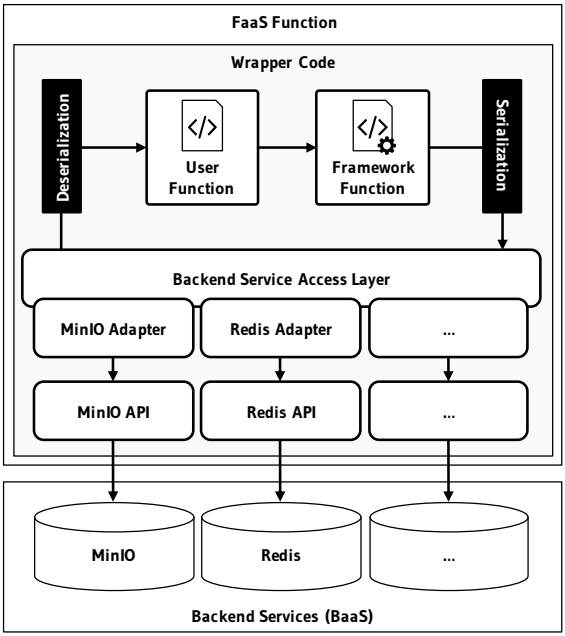

Figure 6.2: FaaS functions can be automatically generated by combining user and framework functions according to the selected FaaS function topology. Generated wrapper code handles the communication via backend services as well as the serialization and deserialization of data.

6.1.4 Automated Delivery and Deployment

Delivery and deployment automation are integral concepts related to cloud programming and have been shown to effectively shorten software release cycles. Figure 6.3 summarizes the integration of the aforementioned concepts to create a continuous delivery pipeline for parallel cloud programming with serverless skeletons. Whereas developers have to implement the user functions required by a particular skeleton, all other steps shown in Figure 6.3 can be automated including the compilation of a serverless skeleton instance (which comprises the generation of wrapper code) and the deployment to

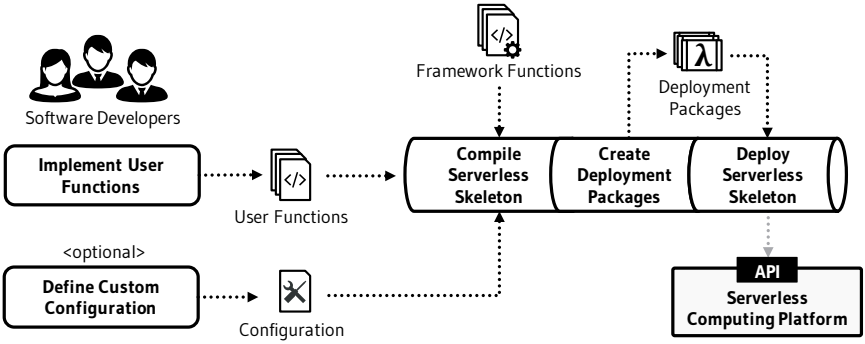

Figure 6.3: A continuous delivery pipeline complements the proposed approach of parallel cloud programming with serverless skeletons.

a serverless computing platform by means of deployment packages. The specification of a skeleton-specific custom configuration is optional (zero-configuration approach).

6.2 Constructing a Serverless Farm Skeleton Framework

In this section, a serverless version of the well-known farm skeleton is presented. Many parallel applications can be implemented based on this skeleton. Prominent examples include frame rendering in computer graphics, brute-force search in cryptography, and Monte Carlo simulation. To validate the concepts proposed in Section 6.1, a Java-based development and runtime framework for the serverless farm skeleton is presented. The remainder of this section is structured as follows. First, the serverless computing platform, upon which the prototypical implementation is built, is described. Subsequently, (1) the user and framework functions of the serverless farm skeleton as well as their implementations, (2) potential mappings of the skeleton function topology to FaaS function topologies, (3) the communication via shared backend services, as well as (4) debugging, (5) delivery, and deployment aspects are discussed.

Serverless Computing Platform: The serverless computing platform ad-

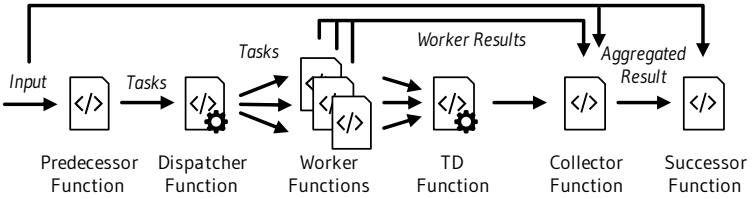

Figure 6.4: User and framework functions of the serverless farm skeleton

dressed is Apache OpenWhisk - an open source serverless computing platform that executes FaaS functions based on events from external sources or API calls. Technically, functions are deployed as Docker containers. The functional logic implemented by developers is called *action* in OpenWhisk jargon and can be written in one of the following programming languages: Node.js, Swift, Java, Go, Scala, Python, PHP, Ruby, or Ballerina. In addition, two backend services are employed: MinIO and Redis. MinIO[1] is an open source object storage that provides an Amazon S3[2] compatible API for data access. Redis[3] is an in-memory data store that can be used as database, cache, or message broker.

6.2.1 User and Framework Functions

In this section, the user and framework functions of a serverless farm skeleton (depicted in Figure 6.4) are described. Related design considerations are discussed accordingly. Function naming is inspired by [PK08b]. The signatures of user functions are declared by Java interfaces, which have to be implemented by developers. Relevant Java methods are shown in Figure 6.5. Note that framework functions are transparent to developers.

Predecessor (User) Function: The predecessor function receives a set of input key-value pairs and initiates the farm skeleton by creating a set of tasks. Each task is described by a set of key-value pairs with the key being a

[1] https://min.io.
[2] https://aws.amazon.com/s3.
[3] https://redis.io.

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

Figure 6.5: Signatures of the serverless farm skeleton user functions declared by Java interfaces

`String` and the value being an `Object`. Finally, the predecessor function returns the tasks that should be processed in parallel.

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

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

Termination Detection (Framework) Function: To detect the termination [GGKK03] of all worker functions, the termination detection function is invoked by each worker function when its computation has been completed. Because point-to-point communication is not supported by serverless computing platforms and FaaS functions are stateless, termination detection has to be implemented based on a shared backend service. As termination is a persistent property of the global system state, which means that once detected it should never be changed again, the implementation of termination detection based on a backend service requires particular attention. False positive or false negative termination detection signals can compromise the execution by detecting termination more than once or never. The implementation is based on Redis. Redis' atomic increment operations are employed to implement a counter, which is incremented atomically once per completed worker function. Termination is detected when the counter has reached the total number of worker functions (which is initially known). In this case, the collector function is invoked.

Collector (User) Function: The collector function receives a set of result values, where each result value has been computed by one worker function. Additionally, the set of input key-value pairs, originally received by the predecessor function, is passed. This is required by applications for which the implementation of the collector function depends on the original input (for an example see Section 6.3.1). Developers implement any application-specific aggregation logic that merges together these result values, e.g., summing up all values. The aggregated result value computed by the collector function is an arbitrary `Object`.

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

6.2.2 FaaS Function Topologies

Figure 6.6 shows potential mappings of user and framework functions of the serverless farm skeleton to FaaS functions. The first mapping offers the highest flexibility, whereas the last mapping is designed to minimize overhead (cf. Section 6.1). Mapping ❹ (cf. Figure 6.6) has the least achievable overhead. Further reduction of FaaS functions is not possible because worker functions have to be scaled out to make use of parallel processing. The framework provides a `YAML`-based configuration file that exposes different FaaS function topology mappings as configuration options. Mapping ❹ is selected by default. Other mappings should only be selected if more flexibility is actually required.

6.2.3 Communication via Backend Services

Based on the communication concept described in Section 6.1 and depicted in Figure 6.2, the framework transparently ensures the communication across FaaS functions. The prototypical implementation of the backend service

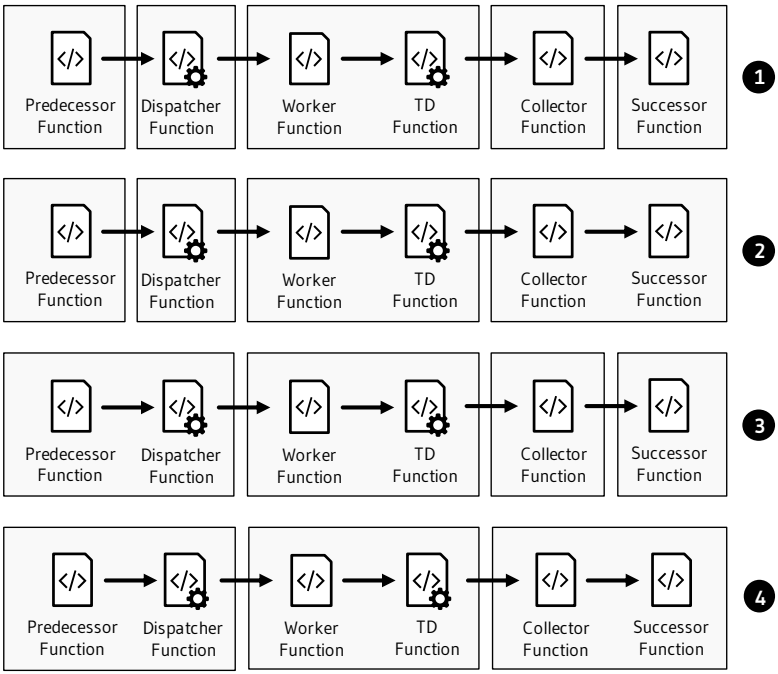

Figure 6.6: Four alternative mappings of the serverless farm skeleton function topology to FaaS function topologies for deployment

access layer supports two different backend services, namely MinIO and Redis. To implement the MinIO adapter, the MinIO client Java SDK[1] version 5.0.6 is used. To implement the Redis adapter, the Redis Java client jedis[2] version 3.0.1 is used. More adapters can be easily added.

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

[1] https://github.com/minio/minio-java.
[2] https://github.com/xetorthio/jedis.

billed per time unit.

6.2.4 Debugging Serverless Skeletons

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

6.2.5 Delivery and Deployment

To deliver and deploy a serverless farm skeleton instance, a delivery pipeline has been implemented as depicted in Figure 6.3:

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

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

isolation is ensured by means of container virtualization because each FaaS function is operated in a separate container. This is automatically ensured by the serverless computing platform.

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

6.2.6 Design Trade-offs

In this section, the design trade-offs made are discussed based on the assessment approach described in Chapter 4.

Parallel applications based on the serverless farm skeleton programming abstraction statically generate tasks (e^-). In general, this negatively affects the cloud readiness because the number of tasks as well as task sizes cannot be changed at runtime (cf. Section 4.4). However, in the context of serverless task farming, an overdecomposition strategy can be employed to support different numbers of processing units. This means that many fine-grained tasks are created by the predecessor function, which can be grouped to create larger tasks. This approach is discussed in Section 6.4.3 to integrate the proactive elasticity control mechanism. Note that tasks are also statically mapped to worker functions (e^-, i^-). However, as the number of worker functions can be determined by means of a proactive elasticity control mechanism (cf. Section 6.4), there is no need to adapt the number of processing units and thus dynamic task mapping is not required to control the cost/efficiency-time trade-off. Additionally, a worker function that completed all tasks simply stops its execution and thus does not lead to idle time. The *task interaction* can be described as *structured* (l^+). The *communication model* is *asynchronous* (l^+). The size of *associated data*, which is stored in external backend services (cf. Section 6.2.3), is *small* (d^+, e^+, i^+).

6.3 Case Studies

In this section, two prototypical applications are discussed that can be easily developed and deployed with the presented serverless farm skeleton framework: Numerical integration and hyperparameter optimization of an artificial neural network. The implementation of each application based on the serverless farm skeleton framework is described in detail.

6.3.1 Numerical Integration

The numerical integration application computes the numerical value of a definite integral of a user-defined function $f(x)$. Therefore, a commonly used technique for approximating the definite integral is employed: The *trapezoidal rule* from the closed Newton-Cotes formulas [Atk89]. The region under the graph $f(x)$ is approximated as a trapezoid of which the area can be easily calculated:

$$\int_a^b f(x)dx \approx (b-a) \cdot \frac{f(a)+f(b)}{2} \tag{6.1}$$

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

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2} \cdot \left(f(x_0) + f(x_N) + 2 \cdot \sum_{k=1}^{N-1} f(x_k) \right), \tag{6.2}$$

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

Implementation: A developer has to implement the partitioning of the integration integral as part of the predecessor function, which is automatically dispatched by the dispatcher (framework) function. Each subinterval is calculated independently by a worker function. Termination is transparently detected by the termination detection (framework) function. Thereafter, the collector function calculates the final value of a definite integral based on Equation 6.2, which is relayed to the successor function accordingly.

6.3.2 Hyperparameter Optimization

Many machine learning techniques are configured by means of parameters that have to be determined. These parameters are called hyperparameters. A prime example are artificial neural networks, which can be configured by a multitude of hyperparameters that influence their network architecture (number of layers, layer size) or the learning process (learning rate). The optimal configuration has to be selected from a (most often) highly multi-dimensional hyperparameter space. Finding the optimal configuration is a non-trivial process referred to as hyperparameter optimization [BYC13].

A commonly used approach for hyperparameter optimization is grid search, which is employed in this case study. However, note that also other approaches such as random search [BB12] can be easily implemented based on the serverless farm skeleton framework because hyperparameter configurations can be evaluated independently of each other and can thus be farmed out for distributed computation. Random and grid search are discussed more thoroughly in [BB12].

The hyperparameter optimization application considers a simple artificial neural network following the multilayer perceptron (MLP) architecture and is designed to optimize the layer size of a hidden layer. The goal is to find the layer size with the highest prediction accuracy (for the data set employed). The network architecture comprises three layers: An input layer, a hidden layer, and an output layer. To train the network, the well-known MNIST[1]

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

data set is used. MNIST is a large collection of handwritten digits that is commonly used to benchmark classification techniques. The input layer of the network has a fixed size of 784, which corresponds to the number of pixels of MNIST images ($28 \cdot 28 = 784$). The fully connected hidden layer uses Rectified Linear Units (ReLU) [NH10] activation functions. The output layer has a fixed size of 10 (representing the 10 possible numbers of the MNIST data set). The learning algorithm employed is stochastic gradient descent.

Implementation: A developer has to implement (1) the generation of hyperparameter configurations as part of the predecessor function, (2) the training and evaluation of an artificial neural network based on a hyperparameter configuration for the worker function, and (3) the aggregation of results for the collector function. In this case, the collector function selects the hyperparameter configuration that produced the best accuracy. The successor function writes the output to the console. Task distribution and termination detection are transparently handled by framework functions. The implementation of hyperparameter optimization is based on Deeplearning4j[1] - a deep learning framework for the Java Virtual Machine (JVM). The ND4J[2] scientific library is employed for linear algebra operations.

6.4 Proactive Elasticity Control for Serverless Skeletons

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

[1] https://github.com/deeplearning4j/deeplearning4j.
[2] https://github.com/deeplearning4j/nd4j.

6.4.1 Automating the Cost/Efficiency-Time Trade-off

A user that employs the serverless task farming framework to implement and execute an application has to select the number of processing units (i.e., the number of worker FaaS functions) per application run. This is implicitly done by the predecessor function that generates a specific number of tasks, each of which is processed by an independent worker FaaS function (cf. Section 6.2.1). As an additional feature, a proactive elasticity control mechanism is presented that predicts the number of processing units required to meet a user-defined execution time limit after which the result of the computation needs to be present while minimizing the associated monetary costs. This is an important scenario, e.g., when parallel processing is embedded in existing workflows. To minimize the monetary costs of the computation, one has to select the minimum number of processing units with which the computation still finishes within the user-defined execution time limit. With this approach, a user does not have to understand the scaling behavior of the application. Rather, the number of processing units is selected by the elasticity controller according to the user-defined execution time limit and thus the cost/efficiency-time trade-off is handled in an automated manner. To this end, the elasticity controller requires a prediction model that captures the application-specific scaling behavior and is able to estimate the required number of processing units upfront. It is discussed how to generate such a model by following a supervised learning approach that considers monitoring data obtained from previous runs of the application.

Finding the required number of processing units can be considered a regression problem. As opposed to classification problems, which require predicting a discrete class label output, regression problems require predicting a continuous quantity output. The output in this case is the number of processing units, which should be provisioned for the computation. Note that, to employ regression techniques, the number of processing units is modeled as continuous quantity, but it can easily be rounded to a (non-negative) integer value for provisioning. Regression techniques can be used to fit a model that explains a response variable based on one (or more) explanatory

variables by estimating the model parameters from data. Therefore, often the method of least squares is used to minimize the sum of the squares of the differences between the observed response variable in a given data set and the predicted response variable. The resulting model can be used to make a prediction of the response variable if values of the explanatory variable(s) are known.

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

6.4.2 Constructing a Prediction Model

To meet a user-defined execution time limit T_{limit}, the elasticity controller has to select the required number of processing units p that speeds up the processing such that the parallel execution time $T_{par} \leq T_{limit}$. The speedup S is defined as $S(I,p) = \frac{T_{seq}(I)}{T_{par}(I,p)}$ (cf. Section 2.1.2). However, due to the non-linear scaling behavior, the speedup does not increase linearly with respect to the number of processing units p. This behavior can be explained based on Amdahl's law [Amd67] which says $T_{seq}(I) = t_s(I) + t_p(I)$ and

$$T_{par}(I,p) = t_s(I) + \frac{t_p(I)}{p}, \tag{6.3}$$

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

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

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

sequential (cf. Figure 6.4).

Based on Equation 6.3, the number of processing units can be expressed as

$$p = \frac{t_p(I)}{T_{par}(I,p) - t_s(I)}. \tag{6.4}$$

Because $T_{seq}(I) = t_s(I) + t_p(I)$ according to Amdahl's law, the execution time of the parallelizable program part $t_p(I) = T_{seq}(I) - t_s(I)$. Consequently, the number of processing units p can also be described as

$$p = \frac{T_{seq}(I) - t_s(I)}{T_{par}(I,p) - t_s(I)}. \tag{6.5}$$

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

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

To enable predictions of the number of processing units, it is required that the sequential execution time T_{seq} and the execution time of the sequential

program part t_s can be described as a function of the input size I_{size}, which can be fitted based on labeled performance measurement data. Here, the sequential execution time T_{seq} is modeled as nth degree polynomial of I_{size}, i.e.,

$$T_{seq}(I) = \sum_{i=0}^{n} \alpha_i \cdot (I_{size}(I))^i, \tag{6.6}$$

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

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

$$t_s(I) = \sum_{j=0}^{m} \beta_j \cdot (I_{size}(I))^j, \tag{6.7}$$

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

This modeling also covers linear and quadratic relations depending on how n and m are selected. In Section 6.5.4, it is shown that this modeling is sufficient to enable accurate predictions for task farming applications. However, in the context of other application classes, other models might be used for T_{seq} and t_s (e.g., logarithmic or exponential models).

Based on Equation 6.5, Equation 6.6, and Equation 6.7, a non-linear regression model that enables the prediction of the required number of processing units can be constructed as

$$\hat{p}(I_{size}(I), T_{limit}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{\sum_{i=0}^{n} \alpha_i \cdot (I_{size}(I))^i - \sum_{j=0}^{m} \beta_j \cdot (I_{size}(I))^j}{T_{limit} - \sum_{j=0}^{m} \beta_j \cdot (I_{size}(I))^j}, \tag{6.8}$$

where $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_n)$ and $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_m)$ are the parameter

vectors of the model and n and m control the sizes of these vectors.

To instantiate a concrete prediction model by means of supervised learning, n and m have to be defined and the parameter vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ have to be estimated from data. Note that increasing n and m increases the number of parameters of the model. However, a general design goal of regression models is to keep the number of model parameters small. This has several reasons: simple models are easier to understand, avoid the curse of dimensionality, and reduce the risk of overfitting [Agg15; Ber16]. Therefore, the prediction model is generated for different, increasing values of n and m (cf. Equation 6.8) until the accuracy of the resulting model in terms of the root-mean-squared-error (RMSE) cannot be significantly increased anymore. Technically, this can be evaluated by comparing the increase in accuracy to a defined threshold.

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

6.4.3 Serverless Elastic Parallel System Architecture

To enable proactive elasticity control for serverless skeletons, several additional functions have to be introduced, which are described in the following. Figure 6.7 shows the resulting serverless elastic parallel system architecture in the context of the serverless farm skeleton. Whereas this architecture is independent of the technologies and programming languages used, a Java-based prototypical implementation that employs Redis as monitoring and model backend service is described accordingly.

Monitoring (Framework) Function: The monitoring function extracts relevant monitoring data of previous application runs from the serverless computing platform and stores them in the monitoring backend service.

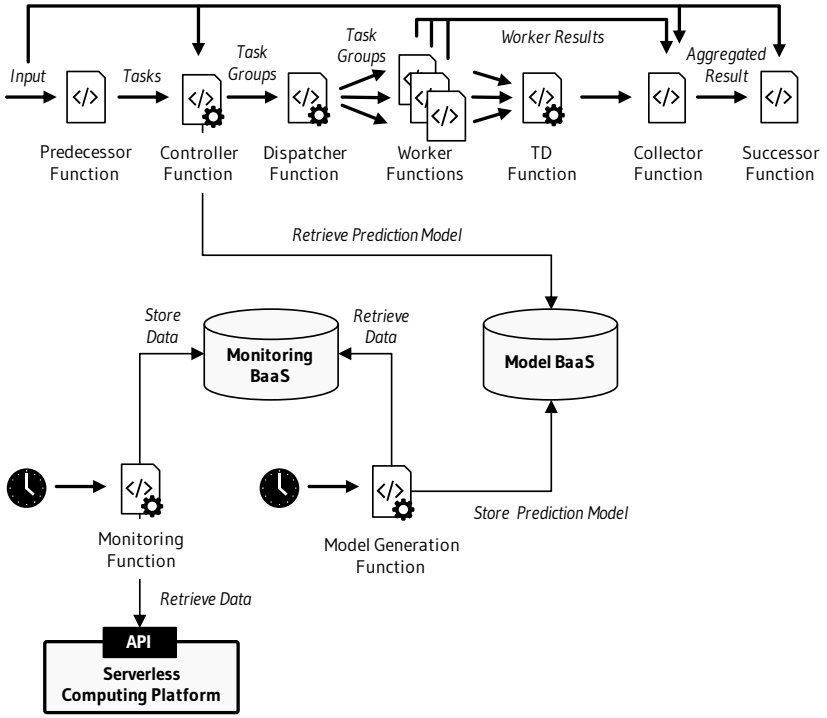

Figure 6.7: Serverless elastic parallel system architecture

Alternatively, monitoring data can also be generated by the skeleton framework, e.g., by storing custom time stamps in the monitoring backend service. This can be easily accomplished by adding instrumentation code to the wrapper code of a skeleton instance's FaaS functions (cf. Section 6.2.3) and enables the consideration of custom metrics. The monitoring (framework) function is triggered periodically.

Model Generation (Framework) Function: The model generation function generates a prediction model by using the monitoring data stored in the monitoring backend service. Also preprocessing steps can be integrated, e.g., to only consider recent data. The model generation process itself is performed according to the concepts discussed in Section 6.4.2. Therefore,

the model is fitted to the data records by adjusting the model parameters. The underlying non-linear least squares problem is solved by employing the Levenberg-Marquardt algorithm [Lev44; Mar63], or more specifically a Java-based implementation[1], which uses JAMA[2] version 1.0.3 for basic linear algebra operations. As discussed in Section 6.4.2, the number of parameters is increased until the accuracy of the resulting model in terms of the RMSE cannot be significantly increased anymore. This is accomplished by comparing the increase in accuracy to a defined threshold. The produced prediction model is stored in the model backend service. Note that the model backend service and monitoring backend service can also be the same entity. The model generation (framework) function is triggered periodically.

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

All these functions are managed by the serverless skeleton framework and have not to be dealt with by the developer. The controller function is mapped to the FaaS function that hosts the dispatcher function and thus does not introduce additional FaaS function topology mappings, which minimizes overhead. The monitoring function and the model generation function can be mapped to the same FaaS function for deployment. In this case, the corresponding FaaS function is triggered periodically to obtain new monitoring data and to generate new models. If a single FaaS function is

[1] https://github.com/odinsbane/least-squares-in-java.
[2] https://math.nist.gov/javanumerics/jama.

employed, monitoring data has not to be stored separately.

6.5 Experimental Evaluation

The skeleton framework is evaluated as follows. First, different FaaS function topologies (cf. Figure 6.6) are compared with respect to their execution time to assess the performance overhead of these deployment options. Second, it is measured how the backend service used affects the execution time by comparing the two different backend services supported by the prototypical implementation (namely MinIO and Redis). Third, the scalability of both example applications described in Section 6.3 is evaluated. Finally, the proposed proactive elasticity control mechanism is evaluated by assessing the accuracy of potential prediction models, which have been generated based on performance measurement data.

Setup. All measurements were executed based on an Apache OpenWhisk installation hosted in an OpenStack-based private cloud environment. The OpenWhisk cluster is operated on two Ubuntu 16.04 virtual machines with 14 vCPUs clocked at 2.6 GHz, 20 GB RAM, and 40 GB disk each. MinIO and Redis are operated on a single Ubuntu 16.04 VM with 2 vCPUs clocked at 2.6 GHz, 8 GB RAM, and 40 GB disk. The hardware underlying the OpenStack-based cloud environment consists of identical servers, each equipped with two Intel Xeon E5-2650v2 CPUs and 128 GB RAM. The virtual network connecting tenant VMs is operated on a 10 Gbit/s physical ethernet network.

6.5.1 FaaS Function Topologies

First, the different FaaS function topologies depicted in Figure 6.6 are compared. Therefore, the execution time of a numerical integration application instance with a sequential execution time T_{seq} of 89.28 seconds is measured. This leads to an execution time of 99.21 seconds for topology ❶, 96.41 seconds for topology ❷, 96.35 seconds for topology ❸, and 93.78 seconds for topology ❹ with Redis as backend service and one worker FaaS function. As expected, topology ❶ has the highest and topology ❹ has the lowest

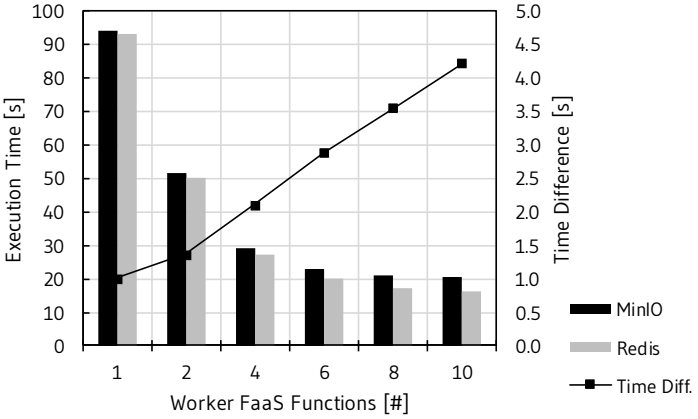

Figure 6.8: Measured execution time of numerical integration application instance with FaaS function topology 4 based on MinIO / Redis backend service

overhead. This is related to the number of FaaS functions employed: Topology ❶ employs 5 FaaS functions whereas topology ❹ employs only 3 FaaS functions. Topology ❷ and ❸ both employ 4 FaaS functions thus leading to similar execution times. Technically, every FaaS function is executed in a Docker container that has to be started. However, note that topology ❶ offers the highest flexibility (cf. Section 6.1).

6.5.2 Backend Services

To compare the performance of the two backend services supported by the prototypical implementation, namely MinIO and Redis, the aforementioned instance of the numerical integration application (T_{seq} = 89.28 seconds) is executed with different degrees of parallelism. Figure 6.8 compares the measured execution times based on MinIO and Redis and shows how the difference of both execution times evolves for an increasing degree of parallelism. The execution based on Redis is faster because it stores all data in memory.

6.5.3 Parallel Performance

The parallel execution time for four instances of the numerical integration application (with different sequential execution times) is measured with respect to different degrees of parallelism. The instances of the numerical integration application have a sequential execution time T_{seq} of (1) 1.02, (2) 9.78, (3) 89.28, and (4) 879.27 seconds. Figure 6.9 shows the achieved elastic speedups. For larger workloads, close to linear elastic speedups were achieved. For small workloads, the overhead outweighs the utility of parallel execution. Elastic speedups determined for the hyperparameter optimization application are shown in Figure 6.10. The application instances depicted have a sequential execution time T_{seq} of (1) 158.68, (2) 516.05, (3) 895.08, and (4) 2071.59 seconds. All parallel performance measurements were executed with FaaS function topology ❹ and Redis backend service. To ensure parallel execution, less worker FaaS functions have been executed in parallel then vCPUs available. Moreover, it is ensured that FaaS functions executed simultaneously are distributed across the OpenWhisk cluster by limiting the invoker user memory.

6.5.4 Proactive Elasticity Control

The controller function groups tasks generated by the predecessor function according to the predicted number of processing units $\hat{p}$ (cf. Section 6.4.3 and Figure 6.7). Consequently, the success of this approach heavily relies on the accuracy of the underlying prediction model. In the following, the model described in Section 6.4.2 is evaluated with respect to its accuracy. Therefore, the model parameters are estimated from monitoring data generated during the performance measurements of the hyperparameter optimization application (cf. Section 6.5.3). The prediction model (cf. Equation 6.8) is fitted to a set of data records, with each record containing the input size and the measured execution time. With respect to the hyperparameter optimization application, the input size is defined as the number of hyperparameter configurations that have to be evaluated. The execution time is given in

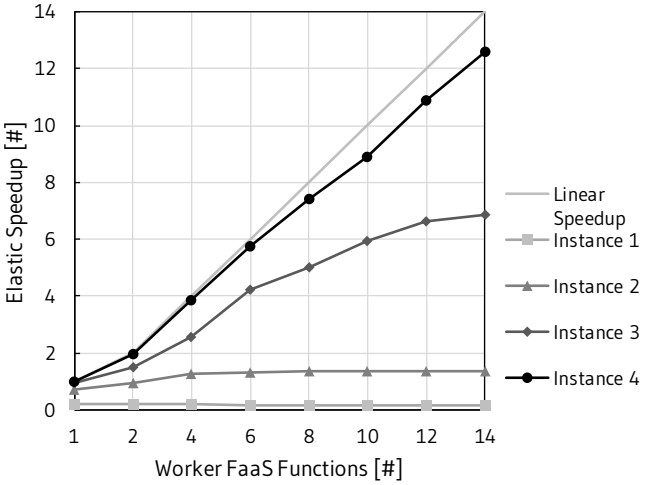

Figure 6.9: Elastic speedups of the numerical integration application with FaaS function topology 4 and Redis backend service

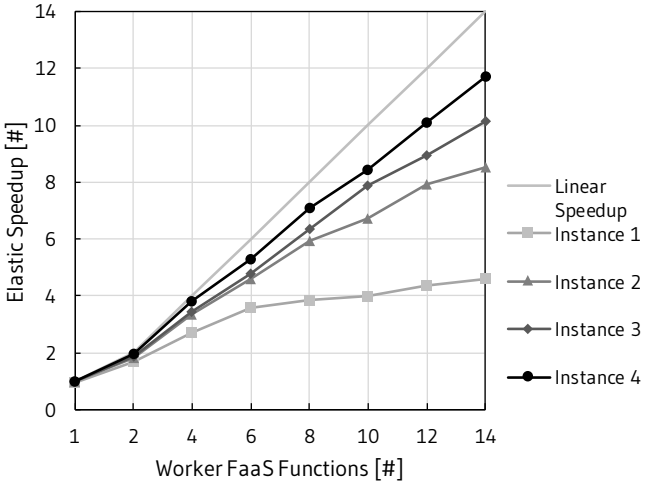

Figure 6.10: Elastic speedups of the hyperparameter optimization application with FaaS function topology 4 and Redis backend service

seconds.

Note that the number of model parameters is automatically selected by the model generation function by defining n and m (as discussed in Section 6.4.2 and 6.4.3). In this regard, several instances of the prediction model $\mathcal{P}_i, i \in \{1, 2, ..., 7\}$ with different sizes of the parameter vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are discussed in the following:

$$\mathcal{P}_i = \begin{cases} n = 0 \text{ and } m = 0 & \text{for} \quad i = 1 \\ n = 0 \text{ and } m = 1 & \text{for} \quad i = 2 \\ n = 1 \text{ and } m = 0 & \text{for} \quad i = 3 \\ n = 1 \text{ and } m = 1 & \text{for} \quad i = 4 \\ n = 1 \text{ and } m = 2 & \text{for} \quad i = 5 \\ n = 2 \text{ and } m = 1 & \text{for} \quad i = 6 \\ n = 2 \text{ and } m = 2 & \text{for} \quad i = 7 \end{cases} \tag{6.9}$$

To evaluate the accuracy of a prediction model instance, the measured number of processing units p is compared with the predicted number of processing units $\hat{p}$. Therefore, the model generation function randomly splits the data records to create a training data set, which contains 75% of the data records used to fit the model instances, and a test data set, which contains 25% of the data records used to assess the prediction accuracy. For the hyperparameter optimization application, 32 data records have been created by means of performance measurements. Thus, 24 data records are employed to train the model instances and 8 data records are employed to evaluate the accuracy. The calculated out-of-sample metrics mean-squared-error *MSE* and root-mean-squared-error *RMSE* are given in Table 6.1. In particular, the prediction model instances $\mathcal{P}_5$, $\mathcal{P}_6$, and $\mathcal{P}_7$ show a good accuracy with respect to the test data set. Prediction model instance $\mathcal{P}_5$ is selected by the model generation function because the accuracy of the model (in terms of the *RMSE*) is better than the accuracy of $\mathcal{P}_6$ and cannot be increased by adding another parameter (cf. Table 6.1).

Model instance $\mathcal{P}_5$ shows the best accuracy with respect to the test data

Table 6.1: Out-of-sample metrics calculated for the generated prediction model instances

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

set. The accuracy of model instance $\mathcal{P}_5$ is also visualized in Figure 6.11, which compares the predicted number of processing units $\hat{p}$ and the measured number of processing units p for the test data set according to the corresponding parallel execution time or execution time limit, respectively. Note that, because the prediction model does not provide integer values, the predicted number of processing units has to be rounded by the controller function. For model instance $\mathcal{P}_5$, the rounded predicted number of processing units is identical to the measured number of processing units for all data records of the test data set (cf. Figure 6.11), which confirms the utility of the presented approach.

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

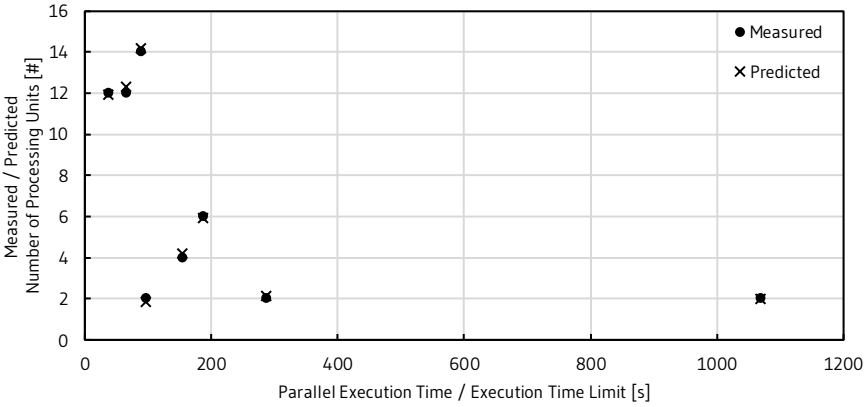

Figure 6.11: Comparison of the measured and predicted numbers of processing units for prediction model instance $\mathcal{P}_5$, which has been fitted to 24 data records.

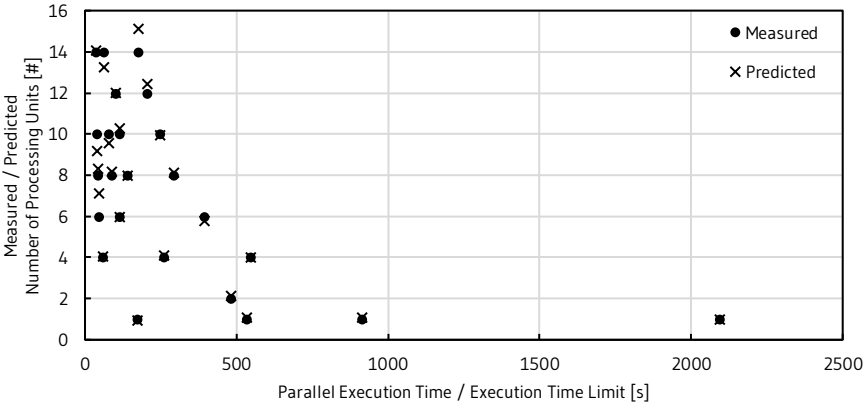

Figure 6.12: Comparison of the measured and predicted numbers of processing units for prediction model instance $\mathcal{P}_5$, which has been fitted to only 8 data records.

6.6 Related Work

Serverless computing promises integrated auto-scaling and transparent resource management, but has only been employed to operate interactive and event-driven applications. More recently, serverless computing platforms have become of interest for parallel processing. The authors of [JPV+17] show that many parallel applications (e.g., applications based on MapReduce) are able to exploit serverless cloud offerings with high bandwidth and high latency object storage as a substitute for distributed memory. The authors present a prototype called PyWren that enables developers to make use of AWS Lambda for parallel execution of locally developed code segments. The authors of [SKP+18] describe how to execute linear algebra algorithms on AWS Lambda. In [WKK+18], serverless computing platforms are evaluated for big data processing use cases based on a matrix multiplication application.

The authors of [ADK09] discuss how to integrate autonomic management of non-functional concerns into algorithmic skeletons. An abstract control loop is described that allows system programmers to compose a given set of monitoring and actuation actions into management rules. Moreover, the adaptation of a skeleton instance at runtime is discussed. The behavioral skeleton concept [ACD+08] is utilized to combine skeletons as higher-level programming abstractions with autonomic managers technically based on a rule engine. The authors of [DZ08] employ the behavioral skeleton concept in the context of the farm skeleton. Their approach considers a so-called *workpool service* for which the service time is optimized based on specified rules. However, a prediction model is not used.

Related work considering performance prediction in the context of parallel applications employs different techniques such as linear regression, support vector machines (SVMs), decision trees, and artificial neural networks [MF10]. In [SİM+07], multilayer neural networks are employed to predict the execution time of parallel applications deployed to different execution environments. The authors of [BRL+08; WM11] use prediction models to extrapolate the performance of an application that solves a problem larger

than the problems used for the measurements (to generate training data). The authors of [WRL15] generate an application model from log data, which is then used to predict an application's performance for different execution environments. In [HWT+13], performance and costs of parallel applications in cloud environments are predicted per application run based on a given application specification. The presented approach automatically selects the optimal resource configuration among a set of defined combinations. The authors of [MAJD17; MAJD18] propose prediction models to find the best resource configuration for a specific application, which can be offered by cloud providers. Their approach is implemented based on random forests, which automatically select the predictors during model construction. Cloud users can employ the prediction model to choose a resource configuration for their application. In contrast to this approach, in this chapter, prediction models are employed to construct a proactive elasticity controller that automatically selects the optimal number of processing units according to user-defined goals.

6.7 Chapter Summary and Discussion

In this chapter, a novel approach to parallel cloud programming that enables elastic parallel processing without considering parallelism or resource management issues is discussed. Based on the well-known concept of algorithmic skeletons, parallel applications, which require coordination, communication, and synchronization, can benefit from serverless computing platforms. Serverless skeletons do not only ease parallel cloud programming, they can also reduce the monetary costs by employing compute resources only when they are efficiently used. The prototypical development and runtime framework shows how to apply the presented concepts to construct a serverless farm skeleton. A proactive elasticity control mechanism handles the cost/-efficiency-time trade-off in an automated manner by predicting the required number of processing units to meet a user-defined execution time limit after which the result of the computation needs to be present while minimizing

the associated monetary costs. The underlying prediction model is obtained and refined in an automated manner by employing supervised learning to infer the scaling behavior from labeled performance measurement data of previous application runs.

Whereas the experimental evaluation shows very promising results for applications implemented based on the farm skeleton, also note that many other parallel execution models (and corresponding skeletons) heavily rely on the consideration of data locality to efficiently exploit compute resources, which is not supported by current serverless computing platforms. This issue should be further investigated in future work. For instance, to retain the strict separation of stateless FaaS functions and backend services, locality-aware backend services can be offered by cloud providers, which store data in close physical proximity to FaaS functions (e.g., on the same rack). In this context, also additional skeleton types and corresponding execution models are required to benefit from serverless computing platforms. Moreover, the presented regression model is based on Amdahl's law, which explains the non-linear scaling behavior of parallel applications by modeling the execution time of a sequential program part and the execution time of a parallelizable program part. As discussed in Section 6.4.2, Amdahl's model is well-suited for serverless task farming applications. However, whereas the experimental results show that this simple model enables accurate predictions for task farming applications, other models might be required for other classes of parallel applications, e.g., to explicitly consider communication overhead. Finally, current serverless computing platforms do not support the use of specialized hardware accelerators. However, it is expected that serverless computing platforms will support these in the future. For instance, hyperparameter optimization would substantially benefit from GPU-enabled training of large artificial neural networks.

Chapter 7

Container-based Module Isolation

Modular software development has become the dominating paradigm to ensure flexibility, adaptability, and maintainability. By means of modern development frameworks (e.g., OSGi[1], JPMS[2]), applications can be designed and developed as a set of small and loosely coupled modules, which can be replaced fast, shared, and reused in another context. Thus, modular software development enables the rapid construction of applications by composing existing, newly developed and / or publicly available modules.

However, whereas composing newly developed and publicly available modules enables rapid software development, satisfying non-functional requirements such as reliability, efficiency, and security in this context is a hard task. Newly added modules might be unstable, resource-intensive, or untrustworthy. In fact, one would like to establish an *isolated context* for these modules to benefit from improved security, fault containment, and

[1] https://www.osgi.org.
[2] http://openjdk.java.net/projects/jigsaw/spec.

fine-grained resource control.

In this chapter, it is discussed how to employ container virtualization to isolate application modules from each other according to a specification of isolation constraints. The resulting concept called *container-based module isolation* enables module-level isolation as an alternative to service-level isolation and is particularly suited for applications with high performance requirements. To this end, non-functional requirements are satisfied by automatically transforming an application composed of modules into a container-based system, which can be deployed to an existing container runtime environment. To deal with the increased overhead that results from isolating modules by means of containers, the minimum set of containers required to satisfy the isolation constraints specified is calculated. The architecture of a transformation pipeline for transforming a modular application into a container-based system is presented and a prototypical implementation that automatically transforms applications developed with the Java Platform Module System (JPMS) into container-based systems is discussed and evaluated.

The remainder of this chapter is structured as follows. Section 7.1 describes several scenarios in which container-based module isolation can be beneficially employed. Section 7.2 introduces the concept of container-based module isolation. In Section 7.3, the architecture of a transformation pipeline that automatically transforms a modular application into a container-based system is presented. A prototypical implementation is discussed in Section 7.4 and evaluated in Section 7.5. Section 7.6 reviews related work. Section 7.7 summarizes the approach and concludes the chapter.

7.1 Problem Statement and Motivation

Modules are well-suited to establish code-level structure and enable code reuse and sharing [Par72]. Today, modern frameworks support modular software development (e.g., OSGi, JPMS), which introduced novel features such as service abstraction. This enables loosely coupled modules and thus

fosters flexibility, adaptability, and maintainability [Par72]. Modules can easily be added, replaced, refactored, or removed depending on the current context and application-specific requirements. Moreover, modules can be shared across teams and with the public by employing private / public code repositories. By following this approach, developers typically enhance their applications with existing modules rather than developing required functionality from scratch. Making use of public code repositories and open source software dramatically increases the speed of development.

Besides, container runtime environments are employed to operate and manage cloud applications [KQ17; STH18]. These environments make use of container virtualization to ease the transition from development to production and substantially reduce the risk of faults due to environmental changes. To make use of these benefits, developers are required to provide their application in form of one or more container images [BO16].

As modules can be shared and reused in another context, they might be unstable, resource-intensive, or untrustworthy. Thus, operating all modules in a single container might impair non-functional requirements such as reliability, efficiency, and security. To deal with this problem, *container virtualization can be employed to isolate selected modules from one or more other modules*. Containers create an isolated context with benefits such as improved security, fault containment, and fine-grained resource control based on established Linux kernel features (such as namespaces and control groups) [PBSJ17; Tur14].

In the following, several scenarios are discussed, in which isolating modules, i.e., limiting their interference, by means of container virtualization and thus deploying an application as a set of containers can be beneficially employed.

Security: Modules imported from third parties contain potentially untrustworthy or malicious code. Whereas known vulnerabilities can be detected by vulnerability scanners in an automated manner, some (yet unknown) vulnerabilities cannot be detected by such scanners. Finding these vulnerabilities is a manual and time-consuming task resulting in delayed software releases. Moreover, developers might want to use third-party modules even

if they are untrustworthy, e.g., because there is no alternative available. Implementing the provided functionality from scratch is often no option, e.g., in the context of rapid prototyping. Well-known examples of security threats are memory safety issues including buffer overflow attacks and dangling pointer bugs as well as filesystem manipulations. Whenever timely releases are an important requirement, operating third-party modules in isolation from security-sensitive modules avoids these threats.

Fault containment: Newly developed, prototypical, and potentially unstable software deteriorates the reliability of an application. Isolating potentially unstable modules by means of container virtualization enables fault containment [Sar03] and thus successfully avoids consequential errors that affect the core functionality of an application [BGO+16].

Resource allocation and control: Isolating a module allows the fine-grained specification of resource requirements for the corresponding container [STH18]. Thus, modules can be restricted with respect to their CPU cycles or memory requirements by simply leveraging resource management features of the underlying operating system.

Testing and monitoring: Testing and monitoring multiple modules operated in a single container requires instrumentation at the programming level. Isolating a module in a container, however, eases monitoring. By following this approach, modules can be monitored by logging the resource consumption and runtime behavior of the corresponding container. Newly added modules can be isolated this way, monitored easily, and might become part of the core functionality over time, e.g., when they have been implemented more efficiently with respect to their resource consumption.

Whereas existing container runtime environments facilitate the deployment and operation of multi-container applications [BO16], there is a lack of concepts and tools that allow developers to employ container virtualization to isolate modules from each other. On the other hand, manually constructing a set of containers is a frustrating and error-prone task as well as subject to frequent changes as an application evolves.

7.2 Concepts of Container-based Module Isolation

In this section, the concept of container-based module isolation is introduced. First, the requirements identified are described. Moreover, it is discussed how the proposed approach satisfies these requirements. Then, it is shown that the concepts can be mapped to the operational principles of existing container runtime environments.

7.2.1 Requirements

The following requirements have been identified:

R1 Software developers require a simple means to specify isolation constraints, which ensure that a module runs in isolation from one or more other modules.

R2 An automated transformation of a modular application to a container-based system has to be ensured. Similarly, automated deployment must be facilitated by generating the deployment artifacts required.

R3 Isolating modules by means of containers leads to various sources of overhead, which should be minimized. Thus, it is essential to find the minimum set of containers required for deployment while satisfying all isolation constraints specified.

R4 To access isolated modules, inter-container communication is required. Access and location transparency have to be achieved to avoid code changes (by developers).

7.2.2 Container-based Module Isolation

To isolate modules by means of container virtualization, simple yet expressive constructs have to be provided in order to specify isolation constraints among one or more modules (*R1*).

In this context, a modular application is represented by a set of modules $\mathcal{M} = \{m_1, ..., m_n\}$. A set of isolation constraints is defined as a binary relation

$\perp \subseteq \mathcal{M} \times \mathcal{M}$, where an isolation constraint $(m_k, m_l) \in \perp$ states that the modules m_k and m_l have to be operated in different containers. This simple formalization also allows the construction of higher-level isolation constraints, which can be automatically transformed to a binary relation.

The generation of a container-based system considering the isolation constraints specified has to be automated (*R2*). Therefore, an automated transformation pipeline is proposed that transforms a given application and a set of isolation constraints to a container-based system (cf. Figure 7.1). Technically, the transformation pipeline generates deployment artifacts that can be used for automated deployment. Details of the transformation pipeline are described in Section 7.3. In the following, the core concepts of the transformation are described by means of a formal specification. The Container-based Module Isolation Problem (CMIP) and the Minimum Container-based Module Isolation Problem (Minimum-CMIP) are defined based on [SB11].

Let $C = \{c_1, ..., c_m\}$ be a set of containers in each of which one or more modules can be operated.

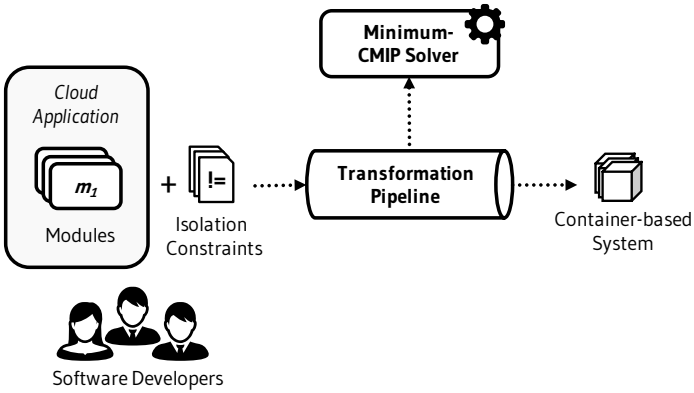

Figure 7.1: Container-based module isolation considers isolation constraints specified by software developers and automatically transforms a given modular application into a container-based system by solving the underlying optimization problem.

Definition 7.1 (Container-based Module Isolation Problem (k-CMIP))
Given an application with a set of modules $\mathcal{M}$ and a set of isolation constraints $\perp$ over $\mathcal{M}$, k-CMIP is the decision problem whether there is a set C of containers with $|C| \leq k$ and a mapping $f : \mathcal{M} \rightarrow C$ such that all isolation constraints in $\perp$ are satisfied.

k-CMIP is equivalent to the graph (vertex) coloring problem [CLRS09] and thus NP-complete. This can be shown by mapping the modules of an application to the vertices of a graph and the isolation constraints specified to the graph's edges. In this context, a valid coloring of the graph represents a mapping of modules to containers.[1]

To minimize the overhead resulting from container-based module isolation, one has to find the minimum set of containers required to satisfy the isolation constraints specified (*R3*). This can be accomplished by solving the corresponding optimization problem.

Definition 7.2 (Minimum Container-based Module Isolation Problem (Minimum-CMIP))
Given an application with a set of modules $\mathcal{M}$ and a set of isolation constraints $\perp$ over $\mathcal{M}$, Minimum-CMIP is the optimization problem of finding a minimum set of containers C, i.e., finding a minimum k, such that a solution to k-CMIP exists for $\mathcal{M}$ and $\perp$.

A Minimum-CMIP solver thus computes a valid and optimized container assignment for the modules of a given application. Based on the calculated container assignment, the proposed transformation pipeline builds a container-based system (cf. Figure 7.1). However, isolating modules by means of containers leads to inter-container communication. To ensure access and location transparency (*R4*), additional proxy mechanisms are required that serve local method / function calls by routing them to isolated modules. Figure 7.2 shows an exemplary container-based system with four

[1] Note that a graph only has a valid coloring if the edge relation is irreflexive. Analogously, an isolation constraint $(m_i, m_i) \in \perp$ cannot be satisfied because a module cannot be isolated from itself. Therefore, it is assumed that $\nexists (m_i, m_i) \in \perp$.

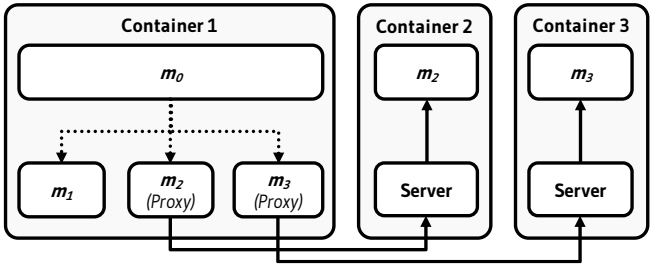

Figure 7.2: Exemplary container-based system that satisfies a set of isolation constraints specified

modules. As one can see, m_2 and m_3 have been isolated from module m_0 (and m_1) and each other. Nevertheless, m_0 accesses the isolated modules m_2 and m_3 in the same manner as the local module m_1. The required proxy mechanisms have to be generated automatically. This is accomplished by the transformation pipeline, which is described in detail in Section 7.3.

7.2.3 Deployment to Container Runtime Environments

Existing container runtime environments support the deployment and management of multi-container applications [BO16]. In Kubernetes jargon a so-called *pod*[1] is defined as a group of one or more containers with shared resources such as storage and network. The containers of a pod are guaranteed to be co-located and are given a shared context. In particular, containers of a pod share an IP address and port space, which eases the generation of proxy mechanisms for inter-container communication. The pod concept has been adopted by other container runtime environments. For example, Marathon, a container orchestrator for Apache Mesos, supports pods with the version 1.4[2]. Nomad supports so-called groups[3] with similar semantics. The prototypical implementation (cf. Section 7.4) generates deployment artifacts for Kubernetes.

[1] https://kubernetes.io/docs/concepts/workloads/pods.
[2] https://mesosphere.github.io/marathon/docs/pods.html.
[3] https://www.nomadproject.io/docs/job-specification/group.html.

These container runtime environments can be hosted in a public or private cloud setting. Furthermore, they are also offered as a service by cloud providers. For instance, Amazon Web Services offers Kubernetes as a service with Amazon EKS[1]. By following this approach, customers benefit from an automated setup with multiple availability zones and cluster auto-scaling.

7.3 Transformation Pipeline

In this section, the architecture of a transformation pipeline for transforming a given application represented by a set of modules and a set of isolation constraints into a container-based system is described. Different technologies can be used to implement this transformation pipeline (for an example see Section 7.4). The transformation pipeline is essentially comprised of five steps, which are depicted in Figure 7.3 and explained in the following.

Prepare transformation: The transformation pipeline performs code-level transformations to isolate application modules based on the isolation constraints specified. Source code files have to be provided by developers and are retrieved from a filesystem or a code repository (version control system) such as Subversion[2] or Git[3]. Additional validity checks can be applied to ensure that the source code provided and the specification of isolation constraints can be processed by the transformation pipeline.

Create container assignment: To create a valid container assignment, i.e., an assignment of modules to containers, the underlying Minimum-CMIP has to be solved. Based on the given application, a corresponding Minimum-CMIP instance is derived. Therefore, the provided source code files have to be scanned to identify modules. As a result, an undirected graph with the modules as vertices and the isolation constraints as edges is generated (cf. Section 7.2). Consequently, algorithms that are able to compute the graph coloring problem can be applied to solve the constructed Minimum-CMIP. However, note that solving such an NP-optimization problem requires an

[1] https://aws.amazon.com/de/eks.
[2] https://subversion.apache.org.
[3] https://git-scm.com.

approximation algorithm because large problems cannot be solved exactly with acceptable runtimes. In Section 7.4, a prototypical implementation of the Minimum-CMIP solver is described. The solution of this optimization problem provides the number of containers required as well as the assignment of modules to containers. The container assignment is the construction plan according to which the following steps build a container-based system.

Isolate modules: In this step, several code-level transformations are applied to isolate modules according to the container assignment provided. To isolate modules by means of containers, inter-container communication is required. Inter-container dependencies are resolved (1) by generating proxy modules, each of which replaces an original module with a proxy and serves local calls by routing them to the isolated module and (2) by generating a corresponding server per container that dispatches inter-container calls to the modules hosted by the container (cf. Figure 7.2). Inter-container calls can be implemented based on any remote procedure call (RPC) technology. However, special emphasis has to be put on ensuring access and location transparency (related to requirement *R4*).

Create container images: Container images are the artifacts from which containers can be instantiated. To specify the construction of a container image, a build specification (e.g., a Dockerfile) is employed. In this step, a build specification is generated per container listed by the container assignment. A build specification contains the modules assigned to the container, the generated proxy mechanisms, as well as dependencies required for execution (e.g., interpreter, runtime system). In this context, it might be required to compile developer-supplied and generated modules. The build specifications are employed to build container images. Finally, these container images are pushed to a container registry from which they can be retrieved for deployment (cf. Figure 7.3).

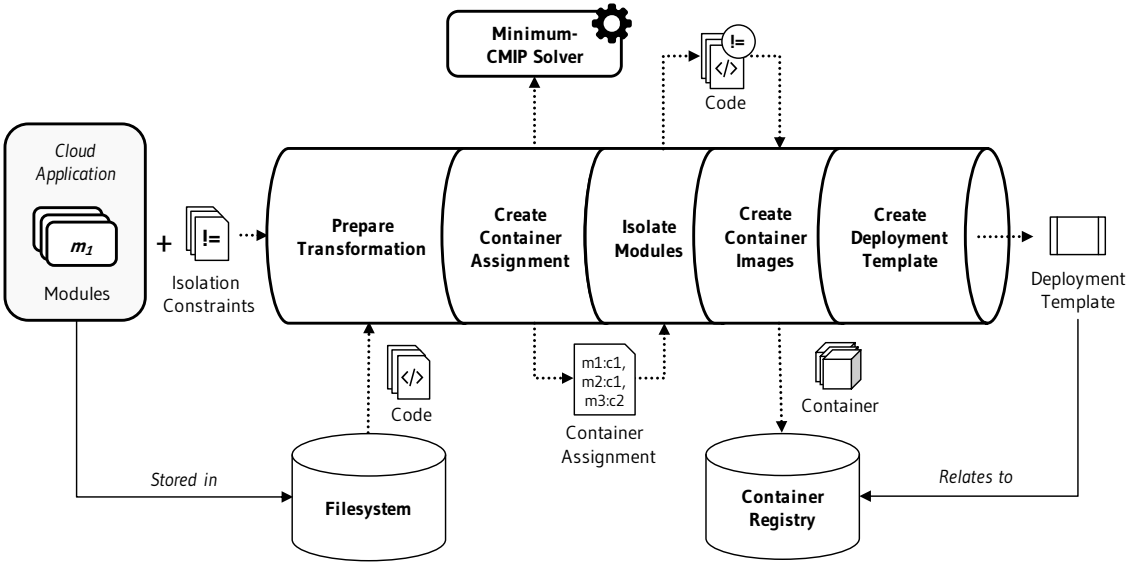

Figure 7.3: The transformation pipeline is comprised of five steps and finally produces a container-based system according to the isolation constraints specified.

Create deployment template: For deployment, a deployment template is required that specifies the container images related to the application. A deployment template can be processed by a container runtime environment in an automated manner. The syntax of the deployment template depends on the target runtime environment. For example, the prototype described in Section 7.4 creates a deployment template for Kubernetes.

7.4 Prototypical Implementation

To validate the concept of container-based module isolation, a prototype of the proposed transformation pipeline has been implemented in Java. The *Java Platform Module System* has been chosen as modular development framework, Docker to build containers, and Kubernetes as target runtime environment.

7.4.1 Java Platform Module System (JPMS)

The novel JPMS, introduced with Java 9, provides the basic constructs for modular software development of Java-based applications and aims to solve several well-known dependency management problems [Jec17]. Therefore, it proposes modules for encapsulating code, which explicitly declare their dependencies to other modules, exported packages, and provided services (implemented interfaces) in the module descriptor file (`module-info.java`). By design, dependencies are static and thus can be verified at compile time. Modules can be distributed in form of modular JAR (Java Archive) files.

The JPMS does not only enable application-level modularity, but has been used to modularize the Java Development Kit (JDK) itself. This is a major advantage compared to monolithic Java Runtime Environments (JRE): By resolving explicit dependencies, only required platform modules are packaged into a so-called *modular runtime image* [Bla18]. A novel JDK-tool called `jlink`[1] can be used to create these runtime images. This is

[1] https://openjdk.java.net/jeps/282.

particularly beneficial in the context of container virtualization because the size of container images is significantly reduced.

7.4.2 Implementation

In the following, it is described how each step of the transformation pipeline depicted in Figure 7.3 is implemented. The expected input is a Java-based application developed with the JPMS. Each module contains a `JSON` file that specifies its isolation constraints.

Prepare transformation: The source code files provided by developers are retrieved from the local filesystem. Each module contains a simple text file describing a list of modules from which it should be isolated. It is checked that the specification of isolation constraints complies with the `JSON` format.

Create container assignment: Modules are identified by means of the Java Module API. The corresponding `JSON` files are parsed to identify isolation constraints. Based on this information, a Minimum-CMIP is constructed. An undirected graph with the identified modules as vertices and the identified isolation constraints as edges is generated. It is checked if $\nexists(m_i, m_i) \in \perp$ holds (cf. Section 7.2). To solve the Minimum-CMIP, the Welsh-Powell algorithm [WP67] is used, which is a (heuristic) greedy algorithm for graph coloring. The Minimum-CMIP solver produces a valid container assignment according to which modules are isolated as described in the following.

Isolate modules: To isolate modules according to the container assignment, several code-level transformations are applied. First, modules that are required locally but hosted in another container are replaced with generated proxy modules. Such dependencies can be derived from the module descriptor file (cf. Section 7.4.1). Proxy modules provide the same interface as the replaced module (via the module descriptor) and internally perform a remote procedure call to the original module (via localhost). Next, a server module is created per container that dispatches inter-container calls to the modules hosted by the container. Technically, remote procedure calls are

implemented based on the Java bindings[1] of ZeroMQ[2], a high-performance messaging library. Access and location transparency are thus ensured by encapsulating the inter-container communication required inside local proxy modules. Proxy and server modules are generated with JavaPoet[3], a source code generation library. Note that it is assumed that all containers are co-located by means of the pod concept (cf. Section 7.2.3). Thus, discovery mechanisms are not required.

Create container images: A temporary folder is created per container on the local filesystem, which contains the build specification of and all artifacts required by the corresponding container. First, the developer-supplied and generated modules are compiled resulting in a bunch of JAR files. Then, a Dockerfile per container has to be created. The generation of Dockerfiles is implemented based on Apache FreeMarker[4], which is an open-source template engine. The Dockerfiles generated are based on the Alpine Linux base image[5]. To build container images, a Docker Engine is assumed to be running on the host. For connecting to the Docker Engine, Docker-Client[6] is used. Docker-Client connects to the Docker Engine via the default UNIX domain socket provided to control Docker-specific functionality. The container images described by the generated Dockerfiles are built and pushed to a private Docker Registry[7] from which they can be retrieved for deployment (cf. Figure 7.3).

Create deployment template: In Kubernetes, deployment templates are specified in `YAML`. As mentioned in Section 7.2.3, a pod is created per application. Thus, the generated `YAML` file defines a single pod and lists all the container images related to this pod that have been built before. The generation of deployment templates is also implemented based on Apache FreeMarker.

[1] https://github.com/zeromq/jeromq.
[2] http://zeromq.org.
[3] https://github.com/square/javapoet.
[4] http://freemarker.org.
[5] https://hub.docker.com/_/alpine.
[6] https://github.com/spotify/docker-client.
[7] https://docs.docker.com/registry.

7.5 Experimental Evaluation

In the following, the results of an experimental evaluation of the prototypical implementation are discussed. It is also shown that solving the Minimum-CMIP does not significantly increase the overall transformation time.

Testbed. The testbed consists of a Kubernetes cluster hosted in an OpenStack[1]-based cloud environment. The Kubernetes master is operated on a CentOS 7 virtual machine with 2 vCPUs clocked at 2.6 GHz, 4 GB RAM, and 40 GB disk. Kubernetes nodes are operated on a CentOS 7 VM with 8 vCPUs clocked at 2.6 GHz, 8 GB RAM, and 40 GB disk.

7.5.1 Communication Overhead

As the implementation transforms method calls to isolated modules into inter-container calls (crossing container boundaries), communication latency is expected to be higher compared to intra-container calls. For evaluation, the latency of an intra-container and an inter-container method call has been measured. The method employed for the measurements requires a `String` object as argument and simply returns this `String` object. The serialized size of this object is only 8 bytes. It thus establishes a baseline for more complex, arbitrarily implemented methods. 150 measurements have been executed for both the intra- and the inter-container method call. All measurements were performed in the Kubernetes testbed described above. Based on the measurements, an average intra-container latency T_{intra} of 60.21 ± 20.4 microseconds and an average inter-container latency T_{inter} of 8405.15 ± 4977.5 microseconds can be calculated. Thus, the average inter-container latency T_{inter} is about 140 times higher. Depending on the actual execution time of the method itself and how frequently it is called, the performance penalty might be negligible. For instance, in the context of compute-intensive methods, the inter-container latency does not significantly increase the total execution time. Also note that the measurements evaluate a worst-case scenario. The execution time of methods is usually much

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

higher. Nonetheless, trade-offs are required with respect to conflicting non-functional properties. Isolation constraints provide a means to trade security for response time.

7.5.2 Deployment Overhead

To compare the deployment time of a single-container application with the deployment time of several multi-container applications, the following experiment has been conducted. The deployment time T_{deploy} has been measured for a single-container application and three multi-container applications with 5, 10, and 20 containers per application. The container image employed for the measurements is based on the Alpine Linux 3.7 base image[1] and OpenJDK 11. It has a total image size of 253 MB. All measurements were performed in the Kubernetes testbed. An average deployment time T_{deploy} of $13.0 \pm 0.7\,\text{s}$ (1 container), $20.7 \pm 1.1\,\text{s}$ (5 containers), $32.0 \pm 1.9\,\text{s}$ (10 containers), and $59.0 \pm 6.5\,\text{s}$ (20 containers) has been calculated based on three deployment runs for each application. Note that the deployment of multiple containers requires sublinear time when compared to a single-container application. For instance, the deployment time of an application with 20 containers only increases by a factor of roughly 4.5. As a result, employing multiple containers to isolate modules from each other does not significantly increase an application's deployment time.

7.5.3 Minimum-CMIP Solver Overhead

It has to be ensured that solving the Minimum-CMIP (which is part of the presented transformation pipeline) does not significantly increase the overall transformation time. To evaluate the performance of the Minimum-CMIP solver, the execution time T_{solve} that is required to create a valid container assignment can be measured. In the following, an experiment is described that shows how the implementation performs for a wide range of different

[1] https://hub.docker.com/_/alpine.

Algorithmus 7.1 Generation of Isolation Constraints

```
1: procedure GenerateConstraints(M_i, w)
2:     ⊥_i ← {}
3:     ξ ← |M_i| · w
4:     for each m_k ∈ M_i do
5:         x ← a random integer in {0, ..., ξ}
6:         k ← GetIndex(m_k)
7:         for j = 0, j < x, j++ do
8:             l ← k
9:             while l = k do
10:                l ← a random integer in {1, ..., |M_i|}
11:            end while
12:            m_l ← GetElement(M_i, l)
13:            if (m_l, m_k) ∉ ⊥_i and (m_k, m_l) ∉ ⊥_i then
14:                ⊥_i ← ⊥_i ∪ {(m_k, m_l)}
15:            end if
16:        end for
17:    end for
18:    return ⊥_i
19: end procedure
```

problems with different characteristics (such as number of modules and isolation constraints as well as constraint densities).

1400 prototypical Minimum-CMIP instances have been generated and evaluated with respect to the corresponding execution time T_{solve}. Minimum-CMIP instances were constructed for 28 different sets of modules $\mathcal{M}_i, i \in \{1, 2, ..., 28\}$, with

$$|\mathcal{M}_i| = \begin{cases} 10 & \text{for} \quad i = 1 \\ |\mathcal{M}_{i-1}| + 10 & \text{for} \quad 1 < i < 10 \\ 100 & \text{for} \quad i = 10 \\ |\mathcal{M}_{i-1}| + 100 & \text{for} \quad 10 < i < 19 \\ 1000 & \text{for} \quad i = 19 \\ |\mathcal{M}_{i-1}| + 1000 & \text{for} \quad 19 < i < 28 \\ 10000 & \text{for} \quad i = 28 \end{cases} \tag{7.1}$$

Table 7.1: Summary of Minimum-CMIP instances with $|\mathcal{M}_i| \leq 100$

Instance	$\lvert\mathcal{M}_i\rvert$	AVG($\lvert\perp_i\rvert$)	AVG(T_{solve})	AVG($\lvert C_i\rvert$)
$\mathcal{M}_1$	10	17.08	23.02 ms	3.76
$\mathcal{M}_2$	20	109.34	23.76 ms	5.56
$\mathcal{M}_3$	30	243.26	24.06 ms	7.04
$\mathcal{M}_4$	40	445	24.69 ms	8.70
$\mathcal{M}_5$	50	694.76	24.51 ms	10.06
$\mathcal{M}_6$	60	998.89	25.01 ms	11.24
$\mathcal{M}_7$	70	1336.6	25.68 ms	12.44
$\mathcal{M}_8$	80	1788.76	25.68 ms	13.74
$\mathcal{M}_9$	90	2227.5	26.14 ms	14.94
$\mathcal{M}_{10}$	100	2754.4	27.07 ms	15.88

The GenerateConstraints procedure shown in Algorithm 7.1 describes the generation of isolation constraints. For each module $m_k \in \mathcal{M}_i$, the procedure adds a random number of isolation constraints to $\perp_i$. GenerateConstraints can be parameterized by means of the weighting factor w to control the constraint density. For each $\mathcal{M}_i$, 10 different weighting factors $w \in \{0.1, 0.2, ..., 1.0\}$ are employed for each of which 5 random configurations of isolation constraints are generated leading to $28 \cdot 10 \cdot 5 = 1400$ Minimum-CMIP instances in total.

Table 7.1 summarizes the average number of isolation constraints, the average execution time, and the average number of containers required to satisfy the isolation constraints specified of all tested Minimum-CMIP instances with $|\mathcal{M}_i| \leq 100$. Figure 7.4 shows the measured execution time T_{solve} for all 1400 Minimum-CMIP instances generated. As one can easily see, all container assignments have been created in less than 23 seconds, even in a scenario with 10000 modules and more than 50 million isolation constraints. Consequently, these measurements show that the Minimum-CMIP solver does not significantly increase the overall transformation time.

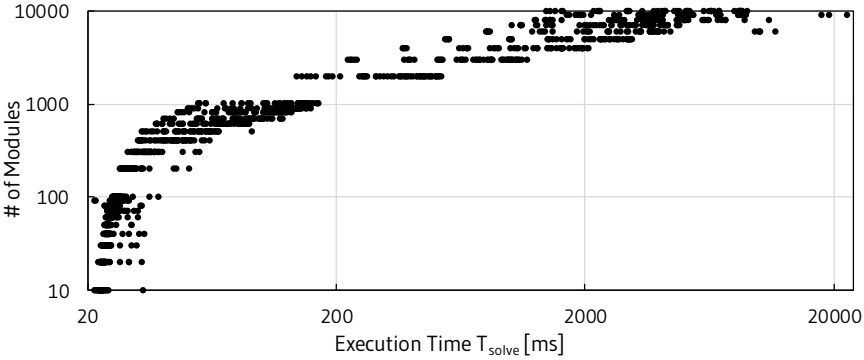

Figure 7.4: The execution time T_{solve} of 1400 Minimum-CMIP instances has been measured by running the Minimum-CMIP solver implemented in Java on a CentOS 7 VM with 8 vCPUs clocked at 2.6 GHz, 8 GB RAM, and 40 GB disk in an OpenStack-based cloud environment. Note that the implementation of the Minimum-CMIP solver is a sequential one, i.e., it does not make use of parallel processing. Both axes of the plot are scaled logarithmically to base 10.

7.6 Related Work

OSGi is another well-known framework for developing Java-based modular software. Modules are called bundles in OSGi jargon. An additional service layer provides a publish-find-bind model for Java interfaces, which can be dynamically bound by means of a service registry mechanism. Whereas both JPMS and OSGi offer code-level encapsulation, they fail to offer isolation in terms of resource allocation and security [GD09; GTFC08; Weg09]. In [Weg09], process-level isolation is employed to isolate modules based on the OSGi framework. The main motivations given are improved security and stability. The authors mainly consider use cases in the field of home automation, whereas here container-based module isolation is employed for applications deployed to state-of-the-art container runtime environments. The authors of [GTM+09] describe a customized JVM that provides isolation and resource accounting for OSGi-based applications. The authors of [SB11]

provide a formal definition of the so-called Module Isolation Problem (MIP) and the corresponding Minimum Module Isolation Problem (Minimum-MIP). Whereas the original concept has been described in the context of distributed computing infrastructures, here the idea of module isolation is applied to cloud applications. The formulations of the CMIP and the Minimum-CMIP represent adaptations of the ones presented in [SB11], which are tailored to the problem setting (cf. Section 7.1). Fundamental differences of both approaches are that the authors of [SB11] do not consider container virtualization, support hot-deployment of modules, and employ an online algorithm for solving the underlying optimization problem. In this work, hot-deployment of modules is not supported. Consequently, Minimum-CMIP has to be solved only once per software release, which is ensured by the Minimum-CMIP solver in the transformation pipeline.

Service-level isolation is a well-known principle in the design and development of cloud applications [LPD+13]. In this chapter, it is shown that containers can be beneficially employed not only for service-level isolation but to isolate application modules from each other, which is beneficial for a variety of use cases (cf. Section 7.1). Current container runtime environments (cf. Section 7.2.3) establish a shared context for all containers of a single pod by means of the same kernel features that are employed to isolate each container. Thus, they support both container-based service isolation and container-based module isolation (as proposed here) out of the box.

In [BO16], the authors argue that containers are the fundamental objects from which distributed systems should be built and discuss several design patterns for container-based systems. Specifically, the *single-node, multi-container application patterns* described are relevant to this work. In this context, three patterns namely the sidecar, ambassador, and adapter pattern have been described. All these patterns describe a specific composition of containers, which can be reused in other contexts. An example for the ambassador pattern is a container that acts as a proxy for communication from / to a main container. This proxy container can be shared across teams and / or with the public, thus other applications can easily reuse the proxy mechanism provided. In contrast to this approach, here, containers

are employed to enforce isolation among modules. Whereas the proposed approach is restricted to a specific programming language, it enables fine-grained control over module composition and non-functional requirements while minimizing the resulting overhead.

7.7 Chapter Summary and Discussion

In this chapter, the concept of container-based module isolation is introduced, which enables software developers to deal with non-functional requirements in the context of applications with high performance requirements. An automated transformation pipeline is presented, which has been validated and evaluated by means of a prototypical implementation. As a result, it is shown how to combine the benefits of modular software development and container virtualization to deal with several non-functional requirements in the context of rapidly changing applications. Technically, security benefits of container-based module isolation (as described in Section 7.1) depend on the container virtualization technique employed. Furthermore, resource allocation and control as well as monitoring depend on the container runtime environment. However, note that the Kubernetes ecosystem, for instance, provides many tools that can be used in conjunction with the proposed concepts.

The concepts presented in this chapter pave the way for future research on new development and engineering methods. For instance, automatic deduction or recommendation of isolation constraints based on testing results can be investigated in the future. Generally speaking, one might think of whole test suites for testing applications based on open source software or third-party components with respect to their non-functional requirements.

Chapter 8

Conclusion and Outlook

Cloud Computing introduces an attractive execution environment for parallel applications with benefits such as on-demand access to compute resources, a pay-per-use billing model, and elasticity. Based on these specific properties, cloud environments offer new capabilities to speed up the time-to-result for both scientists and industrial companies. Cloud users have the freedom to select the required compute resources per application run. By means of elastic scaling, the resource configuration can even be adapted at runtime, which allows fine-grained control of an application's performance as well as the related monetary costs due to the pay-per-use billing model. Whereas this enables even small organizations, which cannot afford and operate a large compute cluster, to benefit from cloud environments, also many challenges result from these new opportunities. As discussed initially in Chapter 1, in this thesis, specifically new opportunities and challenges related to elasticity are addressed. A deep understanding of elasticity-related opportunities and corresponding control mechanisms will be fundamentally required in the future. This is not only backed by HPC-aware cloud offerings, which have been introduced recently. One can also see the adoption of cloud-specific properties and operational principles in supercomputing, which seems to be

a reasonable evolution of traditional environments.

The contributions presented in this thesis are summarized in Section 8.1. Section 8.2 outlines future research opportunities.

8.1 Summary of Contributions

To discuss the opportunities and challenges related to elasticity, Chapter 3 provides a definition of the term elastic parallel system based on existing parallel computing theory. Novel metrics for elasticity control as well as ex-post performance evaluation are presented. On this basis, the opportunities and challenges that result from elasticity are discussed in detail and related to the contributions of this thesis. Specifically, adding an elasticity controller to a parallel system leads to new architectural challenges, which have to be considered during parallel application design. Chapter 4 shows how to assess the cloud readiness of existing parallel applications based on the parallel design decisions made. This systematic approach supports the cloud migration of parallel applications by discovering and understanding the implications of parallel design decisions on the cloud readiness.

Based on the principles of elastic parallel systems, two exemplary elastic parallel system architectures are presented in Chapter 5 and 6. Both architectures include (1) an elasticity controller to handle the cost/efficiency-time trade-off, (2) a cloud-aware parallel execution model that supports a dynamic number of processing units based on cloud resources, and (3) a programming abstraction to ease the implementation of applications. Chapter 5 shows how to employ a reactive elasticity controller to handle the cost/efficiency-time trade-off in the context of state space search. Chapter 6 describes how to enable serverless parallel computations using proactive elasticity control. Whereas the reactive elasticity controller employs user-defined thresholds to dynamically adapt the number of processing units, the proactive elasticity controller described in Chapter 6 employs supervised learning to infer the scaling behavior from labeled performance measurement data of previous application runs. Moreover, the reactive elasticity

controller adapts the physical parallelism by provisioning and decommissioning VMs, whereas the proactive elasticity controller is implemented based on a serverless computing platform and controls the number of FaaS functions. The presented elastic parallel system architectures are designed according to the specific characteristics of the classes of applications considered. To ease the development of elastic parallel applications, both approaches employ higher-level programming abstractions that separate development concerns from parallel execution concerns. As a result, developers are concerned with the functional aspects of the application only, whereas the elasticity controller automatically handles the cost/efficiency-time trade-off according to user-defined goals.

Whereas, delivery and deployment automation in the context of serverless parallel processing has been discussed in Chapter 6, Chapter 7 introduces an approach based on container virtualization, which can be used to either deploy applications to IaaS cloud environments or existing container runtime environments. The presented approach allows for non-functional trade-offs, which are automatically handled by the proposed transformation pipeline.

8.2 Research Opportunities and Future Work

Many principles discussed in this thesis can be applied to other application classes to build elasticity control mechanisms, to construct new programming abstractions, and to design elastic parallel system architectures. On the one hand, proactive mechanisms might also be employed to select the number of processing units for existing parallel applications that have been designed for a static number of processing units (e.g., MPI-based applications). In this case, architectural refactoring is not necessarily required while the cost/efficiency-time trade-off can still be considered. On the other hand, a user might recognize that she requires the results of a computation earlier than expected even though the application run has already been started. By considering this novel context and adapting the number of processing units at runtime, elasticity control can help to improve the flexibility of HPC users

Figure 8.1: The quality of results adds a third dimension to the cost/-efficiency-time trade-off, which enables new Pareto optimal solutions. New research opportunities arise in the context of an elasticity controller that handles this trade-off.

while also providing cost-awareness.

Besides the discussed optimization goals low cost / high efficiency and short execution time also a third goal can be considered (as discussed in Section 5.9): High quality of results. The quality of results adds a third dimension to the cost/efficiency-time trade-off as depicted in Figure 8.1. However, just like one cannot minimize the execution time and the monetary costs at the same time, one cannot maximize the quality of results while minimizing the execution time or the monetary costs. Novel opportunities in form of Pareto optimal solutions arise from this new dimension. For instance, one can enforce a fixed monetary budget or execution time limit by accepting a lower quality of results. To additionally consider the quality of results as part of the trade-off that is handled by an elasticity controller, elastic parallel applications have to produce a sequence of (approximate) intermediate results thus that the computation produces a usable result even if it is terminated [GGH12]. Research on how to control the quality of results dimension and the resulting trade-off (cf. Figure 8.1) in the context of different application classes is required.

Note that scalability theory is also affected by adding the quality of results dimension to the trade-off. There are two different notions of scalability: Strong and weak scalability. By investigating the strong scaling behavior,

one analyzes how the parallel execution time varies with the number of processing units for a fixed problem size, whereas by investigating the weak scaling behavior, one analyzes how the parallel execution time varies with the number of processing units for a fixed problem size per processor. The cost/efficiency-time trade-off, as described in this thesis, only focuses on strong scaling, i.e., it is assumed that the problem size is not affected by the number of processing units employed. Future work that also considers the quality of results dimension as part of the trade-off has to address the issue that the problem size can be varied at runtime. It has to be investigated how this affects the notion of scalability on a conceptual basis as well as related experimental evaluations.

In existing research, elastic parallel systems have been compared based on different dimensions such as monetary costs, energy consumption, or performance (cf. Section 2.3.2). On the other hand, existing work in the context of interactive (multi-tier) applications proposes novel elasticity metrics to evaluate elasticity controllers [BHS+19; HKWG15; KHKR11]. These metrics are designed to evaluate and compare the elastic behavior of systems, i.e., how good resource demand and resource supply are mapped with respect to accuracy and timing [HKWG15]. The authors of [HKWG15] show how to aggregate elasticity metrics to compare a baseline platform with a benchmarked platform. Currently, such approaches are not applied to elasticity control mechanisms in the HPC domain, which requires further investigation.

An important aspect that is typically considered in parallel computing but not addressed in this thesis is fault tolerance. However, it has been recognized that especially serverless computing platforms enable simple fault tolerance mechanisms: Because application state is persisted by means of backend services for communication purposes, one can also reuse stored data for snapshotting-based fault tolerance protocols. Future research might investigate on how to integrate fault tolerance mechanisms at the runtime system level, which are thus transparent to developers. Specifically, large-scale and / or long-running applications benefit from fault tolerance.

Finally, the concept of container-based module isolation as described in

Chapter 7 can be applied to all types of modular applications. For instance, combining the microservices architectural style for the external architecture and modular software development for the internal architecture of cloud applications provides a powerful means to tackle both the increasing complexity of software-based systems and the need for rapid adaptation due to frequently changing requirements. Whereas (micro)services have been proven to benefit from independent deployment and management as well as interoperability [STH18], modules are well-suited to establish code-level structure and enable code reuse and sharing. By applying the concept of container-based module isolation in this context, one is able to establish an isolated context for selected modules to benefit from improved security, fault containment, and fine-grained resource control, but without the need to create, deploy, and manage a separate service. Note that building and maintaining too many (too fine-grained) services has been identified as one of the major issues of the microservices architectural style and leads to performance issues [DGL+17; STH18].

Danksagungen

An erster Stelle möchte ich mich ganz herzlich bei Prof. Dr. Wolfgang Blochinger und Prof. Dr. Dr. h. c. Frank Leymann für die wissenschaftliche Betreuung meiner Doktorarbeit bedanken. Besonderer Dank gilt Prof. Dr. Wolfgang Blochinger für die mir übertragene Verantwortung, das Vertrauen und insbesondere die Unterstützung bei wissenschaftlichen und organisatorischen Fragestellungen. Ich danke allen Mitgliedern des Instituts für Architektur von Anwendungssystemen (IAAS) der Universität Stuttgart und der Forschungsgruppe Parallel and Distributed Computing der Hochschule Reutlingen - besonders Jens Haußmann, der mich über die letzten Jahre mit seinem technischen Fachwissen beraten hat. Bei den ehemaligen Studenten Florian Riebandt und Jochen Scheffold bedanke ich mich für die tatkräftige Unterstützung bei meiner Forschungsarbeit. Außerdem danke ich allen Mitgliedern des Kooperativen Promotionskollegs Services Computing für die nützlichen Hinweise und Kommentare während unserer Kolloquien. Bei Prof. Dr. Alfred Zimmermann und Dr. Dierk Jugel möchte ich mich für die Zusammenarbeit während meines Studiums bedanken. Bei meinem ehemaligen Lehrer Dr. Gerhard Bitsch bedanke ich mich für mein Interesse am Fachgebiet der Informatik, welches meinen Werdegang maßgeblich beeinflusst hat. Ebenso danke ich Dominik Zietlow für unsere zahlreichen Diskussionen. Zum Schluss möchte ich mich bei meinen Eltern, meinen

Schwiegereltern und meiner Frau bedanken, ohne deren Engagement die vorliegende Arbeit in dieser Form nicht entstehen hätte können. Besonderer Dank gilt meiner Frau, die mich jederzeit bestärkt und mir die notwendigen Freiräume geschaffen hat.

Bibliography

[ABD+09] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, et al. “A view of the parallel computing landscape.” In: *Communications of the ACM* 52.10 (2009), pp. 56–67 (cit. on pp. 13, 26, 87, 102).

[ABGL02] K. Anstreicher, N. Brixius, J.-P. Goux, J. Linderoth. “Solving large quadratic assignment problems on computational grids.” In: *Mathematical Programming* 91.3 (2002), pp. 563–588 (cit. on p. 137).

[ABLS13] V. Andrikopoulos, T. Binz, F. Leymann, S. Strauch. “How to adapt applications for the cloud environment.” In: *Computing* 95.6 (2013), pp. 493–535 (cit. on pp. 61, 62).

[ACD+08] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, N. Tonellotto. “Behavioural Skeletons in GCM: Autonomic Management of Grid Components.” In: *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. IEEE, 2008, pp. 54–63 (cit. on p. 171).

[ADK09] M. Aldinucci, M. Danelutto, P. Kilpatrick. “Co-design of Distributed Systems Using Skeleton and Autonomic Management Abstractions.” In: *Euro-Par 2008 Workshops - Parallel Processing*. Ed. by E. César, M. Alexander, A. Streit, J. L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, S. Jha. Springer, 2009, pp. 403–414 (cit. on p. 171).

[ADT03] M. Aldinucci, M. Danelutto, P. Teti. “An advanced environment supporting structured parallel programming in Java.” In: *Future Generation Computer Systems* 19.5 (2003), pp. 611–626 (cit. on p. 29).

[AEJ18] R. Aljamal, A. El-Mousa, F. Jubair. “A comparative review of high-performance computing major cloud service providers.” In: *2018 9th International Conference on Information and Communication Systems (ICICS)*. IEEE, 2018, pp. 181–186 (cit. on p. 14).

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia. “A View of Cloud Computing.” In: *Communications of the ACM* 53.4 (2010), pp. 50–58 (cit. on pp. 14, 31).

[Agg15] C. C. Aggarwal. *Data mining: The textbook*. Springer, 2015 (cit. on p. 161).

[AGJ+14] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, L. Kale. “Parallel Programming with Migratable Objects: Charm++ in Practice.” In: *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 647–658 (cit. on p. 39).

[Ama19] Amazon.com, Inc. *Amazon.com Announces Fourth Quarter Sales up 20% to $72.4 Billion*. 2019. URL: https://ir.aboutamazon.com/news-releases/news-release-details/amazoncom-announces-fourth-quarter-sales-20-724-billion (cit. on p. 31).

[Amd67] G. M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.” In: *Proceedings of the Spring Joint Computer Conference*. AFIPS ’67 (Spring). ACM, 1967, pp. 483–485 (cit. on p. 158).

[AMM+18] B. Archibald, P. Maier, C. McCreesh, R. Stewart, P. Trinder. “Replicable parallel branch and bound search.” In: *Journal of Parallel and Distributed Computing* 113 (2018), pp. 92–114 (cit. on pp. 104, 137).

[AMP14] F. Alexandre, R. Marques, H. Paulino. “On the Support of Task-parallel Algorithmic Skeletons for multi-GPU Computing.” In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 880–885 (cit. on p. 29).

[AMS+17] B. Archibald, P. Maier, R. Stewart, P. Trinder, J. De Beule. "Towards Generic Scalable Parallel Combinatorial Search." In: *Proceedings of the International Workshop on Parallel Symbolic Computation*. PASCO 2017. ACM, 2017, 6:1–6:10 (cit. on p. 137).

[APDM18] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle. "Elasticity in Cloud Computing: State of the Art and Research Challenges." In: *IEEE Transactions on Services Computing* 11.2 (2018), pp. 430–447 (cit. on pp. 15, 31, 32).

[Arc18] B. Archibald. "Algorithmic skeletons for exact combinatorial search at scale." PhD thesis. University of Glasgow, 2018 (cit. on pp. 85, 117, 118).

[Atk89] K. E. Atkinson. *An Introduction to Numerical Analysis*. Second Edition. John Wiley and Sons, 1989 (cit. on p. 154).

[BB12] J. Bergstra, Y. Bengio. "Random Search for Hyper-parameter Optimization." In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305 (cit. on p. 155).

[BBCR10] J. Barnat, L. Brim, M. Ceska, P. Rockai. "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*. IEEE, 2010, pp. 4–7 (cit. on p. 85).

[BBD+14] M. M. Bersani, D. Bianculli, S. Dustdar, A. Gambi, C. Ghezzi, S. Krstić. "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*. PESOS 2014. ACM, 2014, pp. 38–47 (cit. on p. 113).

[BBF+18] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann. "A Systematic Review of Cloud Modeling Languages." In: *ACM Computing Surveys (CSUR)* 51.1 (2018), 22:1–22:38 (cit. on p. 35).

[BDO+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, M. Vanneschi. "P3 L: A structured high-level parallel language, and its structured support." In: *Concurrency: Practice and Experience* 7.3 (1995), pp. 225–255 (cit. on p. 29).

[BDS06] W. Blochinger, C. Dangelmayr, S. Schulz. “Aspect-Oriented Parallel Discrete Optimization on the Cohesion Desktop Grid Platform.” In: *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID 06)*. IEEE, 2006, pp. 49–56 (cit. on p. 137).

[Ber16] R. A. Berk. *Statistical learning from a regression perspective*. Second Edition. Springer, 2016 (cit. on p. 161).

[BF17] M. Bungart, C. Fohry. “A Malleable and Fault-Tolerant Task Pool Framework for X10.” In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 749–757 (cit. on pp. 117, 118, 137).

[BGO+16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes. “Borg, Omega, and Kubernetes.” In: *ACM Queue* 14.1 (2016), pp. 70–93 (cit. on pp. 35, 178).

[BHS+19] A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, S. Kounev. “Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field.” In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (2019), pp. 800–813 (cit. on pp. 31, 201).

[BJK+96] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou. “Cilk: An Efficient Multithreaded Runtime System.” In: *Journal of Parallel and Distributed Computing* 37.1 (1996), pp. 55–69 (cit. on pp. 94, 136).

[BKLW99] W. Blochinger, W. Küchlin, C. Ludwig, A. Weber. “An Object-Oriented Platform for Distributed High-Performance Symbolic Computation.” In: *Mathematics and Computers in Simulation* 49.3 (1999), pp. 161–178 (cit. on p. 137).

[BL09] P. Bonami, M. A. Lejeune. “An Exact Solution Approach for Portfolio Optimization Problems Under Stochastic and Integer Constraints.” In: *Operations Research* 57.3 (2009), pp. 650–670 (cit. on p. 85).

[BL99] R. D. Blumofe, C. E. Leiserson. “Scheduling Multithreaded Computations by Work Stealing.” In: *Journal of the ACM* 46.5 (1999), pp. 720–748 (cit. on p. 94).

[Bla18] N. Black. “Nicolai Parlog on Java 9 Modules.” In: *IEEE Software* 35.3 (2018), pp. 101–104 (cit. on p. 186).

[BMW98] W. Blochinger, W. Michlin, A. Weber. “The distributed object-oriented threads system DOTS.” In: *Solving Irregularly Structured Problems in Parallel*. Ed. by A. Ferreira, J. Rolim, H. Simon, S.-H. Teng. Springer, 1998, pp. 206–217 (cit. on p. 137).

[BO16] B. Burns, D. Oppenheimer. “Design Patterns for Container-based Distributed Systems.” In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX, 2016, pp. 108–113 (cit. on pp. 36, 177, 178, 182, 194).

[BRA+11] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, D. Thain. “Work queue + python: A framework for scalable scientific ensemble applications.” In: *Workshop on Python for High-Performance and Scientific Computing*. 2011 (cit. on pp. 40, 77).

[Bra09] I. Brandic. “Towards Self-Manageable Cloud Services.” In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. IEEE, 2009, pp. 128–133 (cit. on p. 33).

[BRL+08] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, M. Schulz. “A Regression-based Approach to Scalability Prediction.” In: *Proceedings of the 22Nd Annual International Conference on Supercomputing*. ICS ’08. ACM, 2008, pp. 368–377 (cit. on p. 171).

[BST89] H. E. Bal, J. G. Steiner, A. S. Tanenbaum. “Programming Languages for Distributed Computing Systems.” In: *ACM Computing Surveys (CSUR)* 21.3 (1989), pp. 261–322 (cit. on p. 27).

[BSVI07] P. Brenner, C. R. Sweet, D. VonHandorf, J. A. Izaguirre. “Accelerating the replica exchange method through an efficient all-pairs exchange.” In: *The Journal of chemical physics* 126.7 (2007), p. 074103 (cit. on p. 72).

[BT89] D. P. Bertsekas, J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1989 (cit. on p. 48).

[But97] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman, 1997 (cit. on p. 29).

[BYC13] J. Bergstra, D. Yamins, D. D. Cox. “Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures.” In: *Proceedings of the 30th International Conference on International Conference on Machine Learning*. JMLR.org, 2013, pp. I-115–I-123 (cit. on p. 155).

[CB14] R. N. Calheiros, R. Buyya. “Meeting Deadlines of Scientific Workflows in Public Clouds with Tasks Replication.” In: *IEEE Transactions on Parallel and Distributed Systems* 25.7 (2014), pp. 1787–1796 (cit. on p. 32).

[CCB+12] D. Chandler, N. Coskun, S. Baset, E. Nahum, S. R. M. Khandker, T. Daly, N. W. I. Paul, L. Barton, M. Wagner, R. Hariharan, et al. *Report on Cloud Computing to the OSG Steering Committee*. Tech. rep. SPEC OSG Cloud Computing Working Group, 2012. URL: https://www.spec.org/osgcloud/docs/osgcloudwgreport20120410.pdf (cit. on p. 56).

[CDE+13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. “Spanner: Google’s globally distributed database.” In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8 (cit. on pp. 33, 65).

[CDK+01] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001 (cit. on p. 29).

[CGR11] C. Cachin, R. Guerraoui, L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Second Edition. Springer, 2011 (cit. on p. 97).

[Che15] L. Chen. “Continuous Delivery: Huge Benefits, but Challenges Too.” In: *IEEE Software* 32.2 (2015), pp. 50–54 (cit. on p. 34).

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. Third Edition. MIT Press, 2009 (cit. on pp. 103, 181).

[Col91] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991 (cit. on pp. 28, 142).

[Des19] Descartes Labs. *Thunder from the Cloud: 40,000 Cores Running in Concert on AWS*. 2019. URL: https://medium.com/descarteslabs-team/thunder-from-the-cloud-40-000-cores-running-in-concert-on-aws-bf1610679978 (cit. on p. 14).

[DFH+93] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, R. L. While. "Parallel programming using skeleton functions." In: *PARLE '93 Parallel Architectures and Languages Europe*. Ed. by A. Bode, M. Reeve, G. Wolf. Springer, 1993, pp. 146–160 (cit. on pp. 28, 29).

[DG08] J. Dean, S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Communications of the ACM* 51.1 (2008), pp. 107–113 (cit. on p. 29).

[DGL+17] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina. "Microservices: Yesterday, Today, and Tomorrow." In: *Present and Ulterior Software Engineering*. Ed. by B. Mazzara Manueland Meyer. Springer, 2017, pp. 195–216 (cit. on p. 202).

[DPP97] M. Danelutto, F. Pasqualetti, S. Pelagatti. "Skeletons for data parallelism in p31." In: *Euro-Par'97 Parallel Processing*. Ed. by C. Lengauer, M. Griebl, S. Gorlatch. Springer, 1997, pp. 619–628 (cit. on p. 29).

[DSSS05] J. Dongarra, T. Sterling, H. Simon, E. Strohmaier. "High-performance computing: clusters, constellations, MPPs, and future directions." In: *Computing in Science Engineering* 7.2 (2005), pp. 51–59 (cit. on p. 27).

[DZ08] M. Danelutto, G. Zoppi. "Behavioural Skeletons Meeting Services." In: *Computational Science – ICCS 2008*. Ed. by M. Bubak, G. D. van Albada, J. Dongarra, P. M. A. Sloot. Springer, 2008, pp. 146–153 (cit. on p. 171).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. "Declarative vs. Imperative: Two Modeling Patterns for theAutomated Deployment of Applications." In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services, 2017, pp. 22–27 (cit. on pp. 34, 35).

[ED05] D. J. Earl, M. W. Deem. “Parallel tempering: Theory, applications, and new perspectives.” In: *Physical Chemistry Chemical Physics* 7.23 (2005), pp. 3910–3916 (cit. on p. 73).

[EH08] C. Evangelinos, C. N. Hill. “Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon’s EC2.” In: *In The 1st Workshop on Cloud Computing and its Applications (CCA)*. ACM, 2008 (cit. on p. 36).

[EZL89] D. L. Eager, J. Zahorjan, E. D. Lazowska. “Speedup versus efficiency in parallel systems.” In: *IEEE Transactions on Computers* 38.3 (1989), pp. 408–423 (cit. on p. 51).

[FBB08] K. B. Ferreira, P. Bridges, R. Brightwell. “Characterizing application sensitivity to OS interference using kernel-level noise injection.” In: *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2008, pp. 1–12 (cit. on p. 38).

[Fei96] L. Feitelson Dror G.and Rudolph. “Toward convergence in job schedulers for parallel supercomputers.” In: *Job Scheduling Strategies for Parallel Processing*. Ed. by L. Feitelson Dror G.and Rudolph. Springer, 1996, pp. 1–26 (cit. on p. 32).

[FLR+11] C. Fehling, F. Leymann, R. Retter, D. Schumm, W. Schupeck. “An Architectural Pattern Language of Cloud-based Applications.” In: *Proceedings of the 18th Conference on Pattern Languages of Programs*. PLoP ’11. ACM, 2011, 2:1–2:11 (cit. on pp. 61, 62).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014 (cit. on pp. 32, 61, 62, 65, 66).

[Fos00] I. Foster. “Languages for Parallel Processing.” In: *Handbook on Parallel and Distributed Processing*. Ed. by J. Błażewicz, K. Ecker, B. Plateau, D. Trystram. Springer, 2000, pp. 92–165 (cit. on p. 27).

[Fos95] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman, 1995 (cit. on pp. 27, 76).

[GB12] G. Galante, L. C. E. d. Bona. “A Survey on Cloud Computing Elasticity.” In: *2012 IEEE Fifth International Conference on Utility and Cloud Computing*. IEEE, 2012, pp. 263–270 (cit. on pp. 15, 31, 32).

[GC11] S. Gorlatch, M. Cole. “Parallel Skeletons.” In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Springer, 2011, pp. 1417–1422 (cit. on pp. 28, 142).

[GC94] B. Gendron, T. G. Crainic. “Parallel Branch-and-Branch Algorithms: Survey and Synthesis.” In: *Operations Research* 42.6 (1994), pp. 1042–1066 (cit. on p. 117).

[GD09] K. Gama, D. Donsez. “Towards Dynamic Component Isolation in a Service Oriented Platform.” In: *Component-Based Software Engineering*. Ed. by G. A. Lewis, I. Poernomo, C. Hofmeister. Springer, 2009, pp. 104–120 (cit. on p. 193).

[GEM+16] G. Galante, L. C. Erpen De Bona, A. R. Mury, B. Schulze, R. da Rosa Righi. “An Analysis of Public Clouds Elasticity in the Execution of Scientific Applications: a Survey.” In: *Journal of Grid Computing* 14.2 (2016), pp. 193–216 (cit. on pp. 13, 14, 37, 69, 88).

[GFG+16] A. Gupta, P. Faraboschi, F. Gioachin, L. V. Kale, R. Kaufmann, B. Lee, V. March, D. Milojicic, C. H. Suen. “Evaluating and Improving the Performance and Scheduling of HPC Applications in Cloud.” In: *IEEE Transactions on Cloud Computing* 4.3 (2016), pp. 307–321 (cit. on pp. 14, 36, 38, 39, 90).

[GGH12] Y. Guo, M. Ghanem, R. Han. “Does the Cloud need new algorithms? An introduction to elastic algorithms.” In: *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. IEEE, 2012, pp. 66–73 (cit. on pp. 139, 200).

[GGKK03] A. Grama, A. Gupta, G. Karypis, V. Kumar. *Introduction to Parallel Computing*. Second Edition. Pearson Education, 2003 (cit. on pp. 15, 26, 29, 30, 48, 49, 61, 66, 67, 70, 92, 96, 120, 149).

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994 (cit. on p. 145).

[GK99] A. Grama, V. Kumar. “State of the art in parallel search techniques for discrete optimization problems.” In: *IEEE Transactions on Knowledge and Data Engineering* 11.1 (1999), pp. 28–35 (cit. on p. 117).

[GKG+13] A. Gupta, L. V. Kale, F. Gioachin, V. March, C. H. Suen, B. S. Lee, P. Faraboschi, R. Kaufmann, D. Milojicic. “The Who, What, Why, and How of High Performance Computing in the Cloud.” In: *IEEE 5th International Conference on Cloud Computing Technology and Science*. IEEE, 2013, pp. 306–314 (cit. on pp. 14, 37, 38).

[GKM+13] A. Gupta, L. V. Kalé, D. Milojicic, P. Faraboschi, S. M. Balle. “HPC-Aware VM Placement in Infrastructure Clouds.” In: *2013 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2013, pp. 11–20 (cit. on p. 37).

[GL10] H. González-Vélez, M. Leyton. “A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers.” In: *Software: Practice and Experience* 40.12 (2010), pp. 1135–1160 (cit. on p. 29).

[GLS14] W. Gropp, E. Lusk, A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Third Edition. MIT Press, 2014 (cit. on pp. 27, 29).

[GM11] A. Gupta, D. Milojicic. “Evaluation of HPC Applications on Cloud.” In: *2011 Sixth Open Cirrus Summit*. IEEE, 2011, pp. 22–26 (cit. on p. 36).

[GR17] G. Galante, R. da Rosa Righi. “Exploring Cloud Elasticity in Scientific Applications.” In: *Cloud Computing*. Springer, 2017, pp. 101–125 (cit. on pp. 39, 40).

[GRV95] T. Gautier, J. L. Roch, G. Villard. “Regular versus irregular problems and algorithms.” In: *Parallel Algorithms for Irregularly Structured Problems*. Ed. by A. Ferreira, J. Rolim. Springer, 1995, pp. 1–25 (cit. on pp. 83, 86).

[GSKM13] A. Gupta, O. Sarood, L. V. Kale, D. Milojicic. “Improving HPC Application Performance in Cloud through Dynamic Load Balancing.” In: *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 402–409 (cit. on p. 39).

[GTFC08] N. Geoffray, G. Thomas, B. Folliot, C. Clément. “Towards a New Isolation Abstraction for OSGi.” In: *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. IIES ’08. ACM, 2008, pp. 41–45 (cit. on p. 193).

[GTL99] W. Gropp, R. Thakur, E. Lusk. *Using MPI-2: Advanced features of the message passing interface*. MIT Press, 1999 (cit. on pp. 16, 27, 29, 41, 76).

[GTM+09] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frenot, B. Folliot. “I-JVM: a Java Virtual Machine for component isolation in OSGi.” In: *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. IEEE, 2009, pp. 544–553 (cit. on p. 193).

[Hay08] B. Hayes. “Cloud computing.” In: *Communications of the ACM* 51.7 (2008), pp. 9–11 (cit. on p. 31).

[HBK19] J. Haussmann, W. Blochinger, W. Kuechlin. “Cost-efficient parallel processing of irregularly structured problems in cloud computing environments.” In: *Cluster Computing* 22.3 (2019), pp. 887–909 (cit. on pp. 39, 43, 44, 51, 118, 119, 137).

[HF10] J. Humble, D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley, 2010 (cit. on pp. 17, 34).

[HFG+19] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, C. Wu. “Serverless Computing: One Step Forward, Two Steps Back.” In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research*. 2019 (cit. on p. 33).

[HK15] T. Hamm, S. Kehrer. “Goal-oriented Decision Support in Collaborative Enterprise Architecture.” In: *Digital Enterprise Computing*. Ed. by A. Zimmermann, A. Rossmann. Gesellschaft für Informatik, 2015, pp. 175–182 (cit. on p. 23).

[HKJR10] P. Hunt, M. Konar, F. P. Junqueira, B. Reed. “ZooKeeper: Wait-free Coordination for Internet-scale Systems.” In: *Proceedings of the 2010 USENIX Annual Technical Conference*. USENIX, 2010 (cit. on pp. 91, 92, 94).

[HKR13] N. R. Herbst, S. Kounev, R. Reussner. “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, 2013, pp. 23–27 (cit. on pp. 32, 49, 53).

[HKWG15] N. R. Herbst, S. Kounev, A. Weber, H. Groenda. “BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments.” In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2015, pp. 46–56 (cit. on p. 201).

[HM11] J. Humble, J. Molesky. “Why enterprises must adopt devops to enable continuous delivery.” In: *Cutter IT Journal* 24.8 (2011), p. 6 (cit. on pp. 17, 34).

[HNA17] D. M. P. Hung, S. M. S. Naidu, M. O. Agyeman. “Architectures for cloud-based HPC in data centers.” In: *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)*. IEEE, 2017, pp. 138–143 (cit. on p. 13).

[HWT+13] H. Huang, L. Wang, B. C. Tak, L. Wang, C. Tang. “CAP3: A Cloud Auto-Provisioning Framework for Parallel Processing Using On-Demand and Spot Instances.” In: *2013 IEEE Sixth International Conference on Cloud Computing*. IEEE, 2013, pp. 228–235 (cit. on p. 172).

[JA02] L. E. Jordan, G. Alaghband. *Fundamentals of Parallel Processing*. Prentice Hall, 2002 (cit. on p. 48).

[Jec17] A. Jecan. *Java 9 Modularity Revealed: Project Jigsaw and Scalable Java Applications*. Apress, 2017 (cit. on p. 186).

[JKSZ15a] D. Jugel, S. Kehrer, C. M. Schweda, A. Zimmermann. “A decision-making case for collaborative enterprise architecture engineering.” In: *INFORMATIK 2015*. Ed. by D. W. Cunningham, P. Hofstedt, K. Meer, I. Schmitt. Gesellschaft für Informatik, 2015, pp. 865–879 (cit. on p. 23).

[JKSZ15b] D. Jugel, S. Kehrer, C. M. Schweda, A. Zimmermann. “Providing EA decision support for stakeholders by automated analyses.” In: *Digital Enterprise Computing*. Ed. by A. Zimmermann, A. Rossmann. Gesellschaft für Informatik, 2015, pp. 151–162 (cit. on p. 23).

[JPV+17] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, B. Recht. “Occupy the Cloud: Distributed Computing for the 99%.” In: *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 445–451 (cit. on pp. 142, 171).

[JR13] F. Junqueira, B. Reed. *ZooKeeper: distributed process coordination*. O’Reilly Media, 2013 (cit. on pp. 92, 93, 98).

[JRM+10] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, N. J. Wright. “Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud.” In: *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE, 2010, pp. 159–168 (cit. on pp. 36, 80).

[JS15] B. Jennings, R. Stadler. “Resource management in clouds: Survey and research challenges.” In: *Journal of Network and Systems Management* 23.3 (2015), pp. 567–619 (cit. on p. 32).

[JSS+19] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, D. A. Patterson. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. 2019. arXiv: 1902.03383 [cs.OS] (cit. on pp. 33, 141).

[KB18a] S. Kehrer, W. Blochinger. “AUTOGENIC: Automated Generation of Self-configuring Microservices.” In: *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. SciTePress, 2018, pp. 35–46 (cit. on p. 22).

[KB18b] S. Kehrer, W. Blochinger. “TOSCA-based container orchestration on Mesos.” In: *Computer Science - Research and Development* 33.3 (2018), pp. 305–316 (cit. on p. 22).

[KB19a] S. Kehrer, W. Blochinger. “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, pp. 57–69 (cit. on p. 21).

[KB19b] S. Kehrer, W. Blochinger. “Elastic Parallel Systems for High Performance Cloud Computing: State-of-the-Art and Future Directions.” In: *Parallel Processing Letters* 29.02 (2019), pp. 1950006-1–1950006-20 (cit. on p. 22).

[KB19c] S. Kehrer, W. Blochinger. “Migrating parallel applications to the cloud: assessing cloud readiness based on parallel design decisions.” In: *SICS Software-Intensive Cyber-Physical Systems* 34.2 (2019), pp. 73–84 (cit. on p. 22).

[KB19d] S. Kehrer, W. Blochinger. “Model-Based Generation of Self-adaptive Cloud Services.” In: *Cloud Computing and Services Science*. Ed. by V. M. Muñoz, D. Ferguson, M. Helfert, C. Pahl. Springer, 2019, pp. 40–63 (cit. on p. 22).

[KB19e] S. Kehrer, W. Blochinger. “TASKWORK: A Cloud-aware Runtime System for Elastic Task-parallel HPC Applications.” In: *Proceedings of the 9th International Conference on Cloud Computing and Services Science*. SciTePress, 2019, pp. 198–209 (cit. on p. 22).

[KB20a] S. Kehrer, W. Blochinger. “Development and Operation of Elastic Parallel Tree Search Applications using TASKWORK.” In: *Cloud Computing and Services Science*. Ed. by D. Ferguson, V. M. Muñoz, C. Pahl, M. Helfert. Springer, 2020, pp. 42–65 (cit. on p. 21).

[KB20b] S. Kehrer, W. Blochinger. “Equilibrium: An Elasticity Controller for Parallel Tree Search in the Cloud.” In: *The Journal of Supercomputing (in press)* (2020) (cit. on p. 21).

[KC03] J. O. Kephart, D. M. Chess. “The vision of autonomic computing.” In: *Computer* 36.1 (2003), pp. 41–50 (cit. on p. 33).

[KHKR11] M. Kuperberg, N. Herbst, J. v. Kistowski, R. Reussner. *Defining and Quantifying Elasticity of Resources in Cloud Computing and Scalable Platforms*. Tech. rep. 16. Karlsruher Institut für Technologie (KIT), 2011 (cit. on p. 201).

[KJZ16a] S. Kehrer, D. Jugel, A. Zimmermann. “Categorizing Requirements for Enterprise Architecture Management in Big Data Literature.” In: *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*. IEEE, 2016, pp. 1–8 (cit. on p. 23).

[KJZ16b] S. Kehrer, D. Jugel, A. Zimmermann. “A systematic literature review of big data literature for EA evolution.” In: *Digital Enterprise Computing*. Ed. by D. Hertweck, C. Decker. Gesellschaft für Informatik, 2016, pp. 209–220 (cit. on p. 23).

[KKB14] A. Kertesz, G. Kecskemeti, I. Brandic. “An interoperable and self-adaptive approach for SLA-based service virtualization in heterogeneous Cloud environments.” In: *Future Generation Computer Systems* 32 (2014), pp. 54–68 (cit. on p. 33).

[KLB+17] S. Kounev, P. Lewis, K. L. Bellman, N. Bencomo, J. Camara, A. Diaconescu, L. Esterle, K. Geihs, H. Giese, S. Götz, P. Inverardi, J. O. Kephart, A. Zisman. “The Notion of Self-aware Computing.” In: *Self-Aware Computing Systems*. Ed. by S. Kounev, J. O. Kephart, A. Milenkoski, X. Zhu. Springer, 2017, pp. 3–16 (cit. on p. 33).

[KMMS10] K. Keutzer, B. L. Massingill, T. G. Mattson, B. A. Sanders. “A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects.” In: *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. ACM, 2010 (cit. on pp. 15, 27, 61, 62, 66, 73, 80).

[KQ17] N. Kratzke, P.-C. Quint. “Understanding cloud-native applications after 10 years of cloud computing-A systematic mapping study.” In: *Journal of Systems and Software* 126 (2017), pp. 1–16 (cit. on pp. 35, 61, 177).

[KRB19] S. Kehrer, F. Riebandt, W. Blochinger. “Container-Based Module Isolation for Cloud Services.” In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 177–186 (cit. on p. 22).

[KS05] H. Kuchen, J. Striegnitz. “Features from functional programming for a C++ skeleton library.” In: *Concurrency and Computation: Practice and Experience* 17.7-8 (2005), pp. 739–756 (cit. on p. 29).

[KS92] H. Kautz, B. Selman. “Planning As Satisfiability.” In: *Proceedings of the 10th European Conference on Artificial Intelligence*. ECAI ’92. John Wiley and Sons, 1992, pp. 359–363 (cit. on p. 85).

[KSB19] S. Kehrer, J. Scheffold, W. Blochinger. “Serverless Skeletons for Elastic Parallel Processing.” In: *2019 IEEE 5th International Conference on Big Data Intelligence and Computing (DATACOM)*. IEEE, 2019, pp. 185–192 (cit. on p. 21).

[Kuc19] H. Kuchen. “Parallel Programming with Algorithmic Skeletons.” In: *The Art of Structuring: Bridging the Gap Between Information Systems Research and Practice*. Ed. by K. Bergener, M. Räckers, A. Stein. Springer, 2019, pp. 527–536 (cit. on p. 29).

[KY17] A. Kanso, A. Youssef. “Serverless: beyond the cloud.” In: *Proceedings of the 2nd International Workshop on Serverless Computing*. ACM, 2017, pp. 6–10 (cit. on p. 33).

[KZSB20] S. Kehrer, D. Zietlow, J. Scheffold, W. Blochinger. “Self-tuning Serverless Task Farming using Proactive Elasticity Control.” In: *Cluster Computing (in press)* (2020) (cit. on p. 21).

[LBWW17] F. Leymann, U. Breitenbücher, S. Wagner, J. Wettinger. “Native Cloud Applications: Why Monolithic Virtualization Is Not Their Foundation.” In: *Cloud Computing and Services Science*. Ed. by M. Helfert, D. Ferguson, V. Méndez Muñoz, J. Cardoso. Springer, 2017, pp. 16–40 (cit. on pp. 61, 62, 65).

[Lev44] K. Levenberg. “A method for the solution of certain non-linear problems in least squares.” In: *Quarterly of applied mathematics* 2.2 (1944), pp. 164–168 (cit. on p. 163).

[Ley09] F. Leymann. “Cloud computing: The next revolution in IT.” In: ed. by D. Fritsch. Wichmann, 2009, pp. 3–12 (cit. on p. 31).

[LKN+09] A. Lenk, M. Klems, J. Nimis, S. Tai, T. Sandholm. “What’s inside the Cloud? An architectural map of the Cloud landscape.” In: *2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*. IEEE, 2009, pp. 23–31 (cit. on p. 31).

[LML14] T. Lorido-Botran, J. Miguel-Alonso, J. A. Lozano. “A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments.” In: *Journal of Grid Computing* 12.4 (2014), pp. 559–592 (cit. on p. 32).

[LMP+15] M. Leppänen, S. Mäkinen, M. Pagels, V. Eloranta, J. Itkonen, M. V. Mäntylä, T. Männistö. “The highways and country roads to continuous deployment.” In: *IEEE Software* 32.2 (2015), pp. 64–72 (cit. on p. 35).

[LPD+13] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, K. Rojas. “Service Isolation vs. Consolidation: Implications for IaaS Cloud Application Deployment.” In: *2013 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2013, pp. 21–30 (cit. on p. 194).

[LS84] T.-H. Lai, S. Sahni. “Anomalies in Parallel Branch-and-bound Algorithms.” In: *Communications of the ACM* 27.6 (1984), pp. 594–602 (cit. on p. 117).

[LW15] F. Liu, J. B. Weissman. “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*. ACM, 2015, 33:1–33:12 (cit. on p. 83).

[LW86] G. Li, B. W. Wah. “Coping with Anomalies in Parallel Branch-and-Bound Algorithms.” In: *IEEE Transactions on Computers* C-35.6 (1986), pp. 568–573 (cit. on p. 117).

[MAJD17] G. Mariani, A. Anghel, R. Jongerius, G. Dittmann. “Predicting Cloud Performance for HPC Applications: A User-Oriented Approach.” In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2017, pp. 524–533 (cit. on p. 172).

[MAJD18] G. Mariani, A. Anghel, R. Jongerius, G. Dittmann. “Predicting cloud performance for HPC applications before deployment.” In: *Future Generation Computer Systems* 87 (2018), pp. 618–628 (cit. on p. 172).

[Mar63] D. W. Marquardt. “An algorithm for least-squares estimation of nonlinear parameters.” In: *SIAM Journal on Applied Mathematics* 11.2 (1963), pp. 431–441 (cit. on p. 163).

[MCTD13] D. Moldovan, G. Copil, H. Truong, S. Dustdar. “MELA: Monitoring and Analyzing Elasticity of Cloud Services.” In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. IEEE, 2013, pp. 80–87 (cit. on p. 32).

[MF10] A. Matsunaga, J. A. B. Fortes. “On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications.” In: *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 495–504 (cit. on p. 171).

[MG+11] P. Mell, T. Grance, et al. *The NIST definition of cloud computing*. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011 (cit. on p. 31).

[MH11] M. Mao, M. Humphrey. "Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows." In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. ACM, 2011, 49:1–49:12 (cit. on p. 32).

[MKH13] V. Mauch, M. Kunze, M. Hillenbrand. "High performance cloud computing." In: *Future Generation Computer Systems* 29.6 (2013), pp. 1408–1416 (cit. on p. 37).

[MMS01] B. L. Massingill, T. G. Mattson, B. A. Sanders. "Parallel programming with a pattern language." In: *International Journal on Software Tools for Technology Transfer (STTT)* 3.2 (2001), pp. 217–234 (cit. on pp. 61, 62, 66, 67).

[MMS07] B. L. Massingill, T. G. Mattson, B. A. Sanders. "Reengineering for Parallelism: an entry point into PLPP for legacy applications." In: *Concurrency and Computation: Practice and Experience* 19.4 (2007), pp. 503–529 (cit. on pp. 15, 27, 61, 66, 67, 73, 80).

[NCR+18] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, R. Buyya. "HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges." In: *ACM Computing Surveys (CSUR)* 51.1 (2018), 8:1–8:29 (cit. on pp. 13, 14, 36).

[New15] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015 (cit. on p. 33).

[NH10] V. Nair, G. E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines." In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. Omnipress, 2010, pp. 807–814 (cit. on p. 156).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Committee Specification 01*. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html (cit. on p. 35).

[OHL+07] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, C.-W. Tseng. "UTS: An Unbalanced Tree Search Benchmark." In: *Languages and Compilers for Parallel Computing*. Ed. by G. Almási, C. Caşcaval, P. Wu. Springer, 2007, pp. 235–250 (cit. on p. 117).

[Par66] D. F. Parkhill. *The Challenge of the Computer Utility*. Addison-Wesley, 1966 (cit. on pp. 14, 31).

[Par72] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules." In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058 (cit. on pp. 176, 177).

[PARD13] M. Parashar, M. AbdelBaky, I. Rodero, A. Devarakonda. "Cloud Paradigms and Practices for Computational and Data-Enabled Science and Engineering." In: *Computing in Science Engineering* 15.4 (2013), pp. 10–18 (cit. on pp. 16, 81).

[PBSJ17] C. Pahl, A. Brogi, J. Soldani, P. Jamshidi. "Cloud Container Technologies: a State-of-the-Art Review." In: *IEEE Transactions on Cloud Computing* 7.3 (2017), pp. 677–692 (cit. on p. 177).

[PF18] J. Posner, C. Fohry. "Hybrid work stealing of locality-flexible and cancelable tasks for the APGAS library." In: *The Journal of Supercomputing* 74.4 (2018), pp. 1435–1448 (cit. on pp. 117, 118).

[PH05] M. Parashar, S. Hariri. "Autonomic Computing: An Overview." In: *Unconventional Programming Paradigms*. Ed. by J.-P. Banâtre, P. Fradet, J.-L. Giavitto, O. Michel. Springer, 2005, pp. 257–269 (cit. on p. 33).

[PHP16] R. Peinl, F. Holzschuher, F. Pfitzer. "Docker Cluster Management for the Cloud - Survey Results and Own Solution." In: *Journal of Grid Computing* 14.2 (2016), pp. 265–282 (cit. on p. 35).

[PJ18] A. Prabhakaran, L. J. "Cost-Benefit Analysis of Public Clouds for Offloading In-House HPC Jobs." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 57–64 (cit. on pp. 26, 84).

[PK08a] M. Poldner, H. Kuchen. "Algorithmic Skeletons for Branch and Bound." In: *Software and Data Technologies*. Ed. by J. Filipe, B. Shishkov, M. Helfert. Springer, 2008, pp. 204–219 (cit. on pp. 92, 93, 137).

[PK08b] M. Poldner, H. Kuchen. “On Implementing the Farm Skeleton.” In: *Parallel Processing Letters* 18.01 (2008), pp. 117–131 (cit. on p. 148).

[Pri57] R. C. Prim. “Shortest connection networks and some generalizations.” In: *The Bell System Technical Journal* 36.6 (1957), pp. 1389–1401 (cit. on p. 104).

[QCB18] C. Qu, R. N. Calheiros, R. Buyya. “Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey.” In: *ACM Computing Surveys (CSUR)* 51.4 (2018), 73:1–73:33 (cit. on p. 32).

[Ral03] T. Ralphs. “Parallel branch and cut for capacitated vehicle routing.” In: *Parallel Computing* 29.5 (2003), pp. 607–629 (cit. on pp. 85, 137).

[RBA11] A. Raveendran, T. Bicer, G. Agrawal. “A Framework for Elastic Execution of Existing MPI Programs.” In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IEEE, 2011, pp. 940–947 (cit. on pp. 39, 42).

[RCIT11] D. Rajan, A. Canino, J. A. Izaguirre, D. Thain. “Converting a high performance application to an elastic cloud application.” In: *IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2011, pp. 383–390 (cit. on pp. 39, 40, 77, 80).

[RCRR16] R. Rosa Righi, C. A. Costa, V. F. Rodrigues, G. Rostirolla. “Joint-analysis of Performance and Energy Consumption when Enabling Cloud Elasticity for Synchronous HPC Applications.” In: *Concurrency and Computation: Practice & Experience* 28.5 (2016), pp. 1548–1571 (cit. on p. 56).

[RLT17] S. Rathnayake, D. Loghin, Y. M. Teo. “CELIA: Cost-Time Performance of Elastic Applications on Cloud.” In: *46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 342–351 (cit. on pp. 39, 42).

[RM19] H. Röger, R. Mayer. “A Comprehensive Survey on Parallelization and Elasticity in Stream Processing.” In: *ACM Computing Surveys (CSUR)* 52.2 (2019), 36:1–36:37 (cit. on p. 32).

[Ron05] D. P. Ronconi. “A Branch-and-Bound Algorithm to Minimize the Makespan in a Flowshop with Blocking.” In: *Annals of Operations Research* 138.1 (2005), pp. 53–65 (cit. on p. 85).

[RRC+15] R. da Rosa Righi, V. F. Rodrigues, C. A. da Costa, D. Kreutz, H.-U. Heiss. “Towards Cloud-based Asynchronous Elasticity for Iterative HPC Applications.” In: *Journal of Physics: Conference Series* 649 (2015), p. 012006 (cit. on pp. 39, 41).

[RRC+16] R. da Rosa Righi, V. F. Rodrigues, C. A. da Costa, G. Galante, L. C. E. de Bona, T. Ferreto. “AutoElastic: Automatic Resource Elasticity for High Performance Applications in the Cloud.” In: *IEEE Transactions on Cloud Computing* 4.1 (2016), pp. 6–19 (cit. on pp. 39, 41, 77).

[RRC+18] V. F. Rodrigues, R. da Rosa Righi, C. A. da Costa, D. Singh, V. M. Munoz, V. Chang. “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, 2018, pp. 261–268 (cit. on pp. 39, 41).

[RRR+18] R. da Rosa Righi, V. F. Rodrigues, G. Rostirolla, C. A. da Costa, E. Roloff, P. O. A. Navaux. “A lightweight plug-and-play elasticity service for self-organizing resource provisioning on parallel applications.” In: *Future Generation Computer Systems* 78 (2018), pp. 176–190 (cit. on pp. 39, 41, 77).

[RT17] D. Rajan, D. Thain. “Designing Self-Tuning Split-Map-Merge Applications for High Cost-Efficiency in the Cloud.” In: *IEEE Transactions on Cloud Computing* 5.2 (2017), pp. 303–316 (cit. on pp. 39, 43, 44, 51).

[RTA+13] D. Rajan, A. Thrasher, B. Abdul-Wahid, J. A. Izaguirre, S. Emrich, D. Thain. “Case Studies in Designing Elastic Applications.” In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 466–473 (cit. on pp. 39, 40).

[Sar03] T. Saridakis. “Design Patterns for Fault Containment.” In: *Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP)*. Universitätsverlag Konstanz, 2003, pp. 493–520 (cit. on p. 178).

[SB10] S. Schulz, W. Blochinger. “Parallel SAT Solving on Peer-to-Peer Desktop Grids.” In: *Journal of Grid Computing* 8.3 (2010), pp. 443–471 (cit. on pp. 92, 117, 119, 137).

[SB11] S. Schulz, W. Blochinger. “Adjustable Module Isolation for Distributed Computing Infrastructures.” In: *2011 IEEE/ACM 12th International Conference on Grid Computing*. IEEE, 2011, pp. 98–105 (cit. on pp. 180, 193, 194).

[SBHD08] S. Schulz, W. Blochinger, M. Held, C. Dangelmayr. “COHESION - A microkernel based Desktop Grid platform for irregular task-parallel applications.” In: *Future Generation Computer Systems* 24.5 (2008), pp. 354–370 (cit. on pp. 92, 137).

[SBS96] P. Stephan, R. K. Brayton, A. L. Sangiovanni-Vincentelli. “Combinational test generation using satisfiability.” In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15.9 (1996), pp. 1167–1176 (cit. on p. 85).

[SCHW17] S. Shudler, A. Calotoiu, T. Hoefler, F. Wolf. “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*. PPoPP ’17. ACM, 2017, pp. 131–143 (cit. on p. 107).

[Sed84] R. Sedgewick. *Algorithms*. Addison-Wesley, 1984 (cit. on p. 104).

[SİM+07] K. Singh, E. İpek, S. A. McKee, B. R. de Supinski, M. Schulz, R. Caruana. “Predicting parallel application performance via machine learning approaches.” In: *Concurrency and Computation: Practice and Experience* 19.17 (2007), pp. 2219–2235 (cit. on p. 171).

[SKK03] C. Sinz, A. Kaiser, W. Küchlin. “Formal methods for the validation of automotive product configuration data.” In: *Ai Edam* 17.1 (2003), pp. 75–97 (cit. on p. 85).

[SKP+18] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, J. Ragan-Kelley. *numpywren: serverless linear algebra*. Tech. rep. UCB/EECS-2018-137. EECS Department, University of California, Berkeley, 2018. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-137.html (cit. on pp. 39, 41, 142, 171).

[SMDS15] E. Strohmaier, H. W. Meuer, J. Dongarra, H. D. Simon. “The TOP500 List and Progress in High-Performance Computing.” In: *Computer* 48.11 (2015), pp. 42–49 (cit. on p. 26).

[SO99] Y. Sugita, Y. Okamoto. “Replica-exchange molecular dynamics method for protein folding.” In: *Chemical physics letters* 314.1 (1999), pp. 141–151 (cit. on pp. 72, 73).

[SSTP09] M. C. Schmidt, N. F. Samatova, K. Thomas, B.-H. Park. “A scalable, parallel algorithm for maximal clique enumeration.” In: *Journal of Parallel and Distributed Computing* 69.4 (2009), pp. 417–428 (cit. on p. 85).

[ST05] M. Salehie, L. Tahvildari. “Autonomic Computing: Emerging Trends and Open Problems.” In: *ACM SIGSOFT Software Engineering Notes* 30.4 (2005), pp. 1–7 (cit. on p. 33).

[ST98] D. B. Skillicorn, D. Talia. “Models and Languages for Parallel Computation.” In: *ACM Computing Surveys (CSUR)* 30.2 (1998), pp. 123–169 (cit. on p. 27).

[STH18] J. Soldani, D. A. Tamburri, W.-J. V. D. Heuvel. “The pains and gains of microservices: A Systematic grey literature review.” In: *Journal of Systems and Software* 146 (2018), pp. 215–232 (cit. on pp. 33, 177, 178, 202).

[SW03] Y. Sun, C.-L. Wang. “Solving Irregularly Structured Problems Based on Distributed Object Model.” In: *Parallel Computing* 29.11-12 (2003), pp. 1539–1562 (cit. on pp. 83, 86).

[TBB+15] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, A. Edmonds. “An Architecture for Self-managing Microservices.” In: *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*. AIMC ’15. ACM, 2015, pp. 19–24 (cit. on p. 33).

[TBB+17] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, T. M. Bohnert. “Self-managing cloud-native applications: Design, implementation, and experience.” In: *Future Generation Computer Systems* 72 (2017), pp. 165–179 (cit. on pp. 33, 65).

[Tur14] J. Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014 (cit. on pp. 35, 36, 177).

[UCL04] G. Utrera, J. Corbalan, J. Labarta. “Implementing malleability on MPI jobs.” In: *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2004, pp. 215–224 (cit. on pp. 26, 32).

[VD03] S. S. Vadhiyar, J. J. Dongarra. “SRS: A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems.” In: *Parallel Processing Letters* 13.02 (2003), pp. 291–312 (cit. on pp. 26, 32).

[VD14] T. T. Vu, B. Derbel. “Link-Heterogeneous Work Stealing.” In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2014, pp. 354–363 (cit. on pp. 39, 40, 137).

[VGS+17] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, X. Bao. “Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases.” In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1041–1052 (cit. on pp. 33, 65).

[VPB09] C. Vecchiola, S. Pandey, R. Buyya. “High-performance cloud computing: A view of scientific applications.” In: *10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN)*. IEEE, 2009, pp. 4–16 (cit. on p. 13).

[vTT+18] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, A. Iosup. “Serverless is More: From PaaS to Present Cloud Computing.” In: *IEEE Internet Computing* 22.5 (2018), pp. 8–17 (cit. on p. 33).

[WALS18] J. Wettinger, V. Andrikopoulos, F. Leymann, S. Strauch. “Middleware-Oriented Deployment Automation for Cloud Applications.” In: *IEEE Transactions on Cloud Computing* 6.4 (2018), pp. 1054–1066 (cit. on pp. 34, 35).

[Weg09] T. Wegner. “A Secure Multi-Provider OSGi Platform Enabling Process-Isolation by Using Distribution.” In: *Proceedings of the 2009 International Conference on Security & Management, SAM 2009, July 13-16, 2009, Las Vegas Nevada, USA, 2 Volumes*. CSREA Press, 2009, pp. 340–345 (cit. on p. 193).

[WKK+18] S. Werner, J. Kuhlenkamp, M. Klems, J. Müller, S. Tai. “Serverless Big Data Processing using Matrix Multiplication as Example.” In: *2018 IEEE International Conference on Big Data*. IEEE, 2018, pp. 358–365 (cit. on pp. 142, 171).

[WLZ+18] L. Wang, M. Li, Y. Zhang, T. Ristenpart, M. Swift. “Peeking Behind the Curtains of Serverless Platforms.” In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX, 2018, pp. 133–145 (cit. on p. 34).

[WM11] X. Wu, F. Mueller. “ScalaExtrap: Trace-based Communication Extrapolation for Spmd Programs.” In: *SIGPLAN Notices* 46.8 (2011), pp. 113–122 (cit. on p. 171).

[WP67] D. J. A. Welsh, M. B. Powell. “An upper bound for the chromatic number of a graph and its application to timetabling problems.” In: *The Computer Journal* 10.1 (1967), pp. 85–86 (cit. on p. 187).

[WRL15] A. Wong, D. Rexachs, E. Luque. “Parallel Application Signature for Performance Analysis and Prediction.” In: *IEEE Transactions on Parallel and Distributed Systems* 26.7 (2015), pp. 2009–2019 (cit. on p. 172).

[YWW+14] X. Yang, D. Wallom, S. Waddington, J. Wang, A. Shaon, B. Matthews, M. Wilson, Y. Guo, L. Guo, J. D. Blower, A. V. Vasilakos, K. Liu, P. Kershaw. “Cloud computing in e-Science: research challenges and opportunities.” In: *The Journal of Supercomputing* 70.1 (2014), pp. 408–464 (cit. on pp. 14, 37).

[ZCB10] Q. Zhang, L. Cheng, R. Boutaba. “Cloud computing: state-of-the-art and research challenges.” In: *Journal of internet services and applications* 1.1 (2010), pp. 7–18 (cit. on p. 31).

[Zim17] O. Zimmermann. “Microservices Tenets.” In: *Computer Science - Research and Development* 32.3-4 (2017), pp. 301–310 (cit. on p. 33).

[ZLP16] J. Zhang, X. Lu, D. K. Panda. “Performance Characterization of Hypervisor- and Container-Based Virtualization for HPC on SR-IOV Enabled InfiniBand Clusters.” In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 1777–1784 (cit. on p. 38).

[ZLP17] J. Zhang, X. Lu, D. K. Panda. “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*. VEE ’17. ACM, 2017, pp. 187–200 (cit. on pp. 14, 37).

[ZLZ+11] Y. Zhai, M. Liu, J. Zhai, X. Ma, W. Chen. “Cloud versus in-house cluster: Evaluating Amazon cluster compute instances for running MPI applications.” In: *SC ’11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, pp. 1–10 (cit. on p. 36).

All links were last followed on August 5, 2020.

List of Figures

List of Tables

List of Definitions