SICS Software-Intensive Cyber-Physical Systems (2019) 34:73–84
https://doi.org/10.1007/s00450-019-00396-8

SPECIAL ISSUE PAPER

Migrating parallel applications to the cloud: assessing cloud readiness based on parallel design decisions

Stefan Kehrer[1] · **Wolfgang Blochinger**[1]

Published online: 6 February 2019

Abstract
Parallel applications are the computational backbone of major industry trends and grand challenges in science. Whereas these applications are typically constructed for dedicated High Performance Computing clusters and supercomputers, the cloud emerges as attractive execution environment, which provides on-demand resource provisioning and a pay-per-use model. However, cloud environments require specific application properties that may restrict parallel application design. As a result, design trade-offs are required to simultaneously maximize parallel performance and benefit from cloud-specific characteristics. In this paper, we present a novel approach to assess the cloud readiness of parallel applications based on the design decisions made. By discovering and understanding the implications of these parallel design decisions on an application's cloud readiness, our approach supports the migration of parallel applications to the cloud. We introduce an assessment procedure, its underlying meta model, and a corresponding instantiation to structure this multi-dimensional design space. For evaluation purposes, we present an extensive case study comprising three parallel applications and discuss their cloud readiness based on our approach.

Keywords Parallel computing · Cloud · High Performance Computing · Cloud readiness · Application properties · Cloud migration · Non-trivial parallelism

1 Introduction

Cloud computing evolved as fundamental paradigm to consume compute resources as utilities, which are paid on a per-use basis [1,2]. So-called *cloud-native applications* are designed to exploit the characteristics of cloud environments (e.g., self-service, elasticity, pay-per-use) effectively [3–5]. To this end, cloud-native applications are *distributed* and *elastic* systems composed of *loosely coupled* components that *isolate state* in a minimum of components [5–7]. These characteristics are summarized by means of the IDEAL properties [5], which have been successfully employed to build web-based multi-tier applications [5,8].

Moreover, major trends in industry as well as grand challenges in science require new, highly compute-intensive applications for domains such as artificial intelligence, engineering and scientific simulations, and financial risk analysis [9]. Designing applications for these domains is a hard task and requires a deep understanding of parallel processing techniques [1,10]. Such parallel applications are the subject matter in High Performance Computing (HPC).

Whereas parallel applications are traditionally designed to deliver maximum performance on dedicated HPC clusters and supercomputers, we can see a growing interest to adopt the cloud as execution environment [9,11–16]. Recently, Amazon Web Services (AWS)[1] and Microsoft Azure[2] introduced new cloud offerings optimized for HPC workloads [17,18]. Thus, former performance limitations of standard cloud offerings, e.g., low throughput and high latency networks [9,12,19], are resolved effectively. Consequently, a large number of parallel applications designed for HPC clusters can be deployed to cloud environments without modifications.

However, copying existing applications to the cloud does not enable them to exploit cloud-specific characteristics [19,20]. Moreover, elasticity and pay-per-use represent fun-

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

Wolfgang Blochinger
wolfgang.blochinger@reutlingen-university.de

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

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

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

damentally new concepts in HPC leading to new research opportunities [9,19,21]. Whereas parallel computing traditionally aims at solely maximizing the performance in terms of *speedup* and *parallel efficiency* [22], the cloud introduces a novel opportunity: *Cost control*. By now, parallel applications are able to directly relate processing time and/or the quality of results to costs, thus allowing versatile optimizations.

Basically, parallel applications have to be designed according to the IDEAL properties to make use of cloud-specific benefits [5]. However, this restricts parallel application design (cf. Sect. 2). Complex design trade-offs are required to build parallel applications with respect to parallel performance and to consider cloud-specific characteristics at the same time. To successfully migrate parallel applications to the cloud, one requires a deep understanding of the implications of parallel design decisions on an application's cloud readiness.

This paper presents a systematic approach to structure the multi-dimensional design space of constructing parallel applications for the cloud. Specifically, we investigate the implications of parallel design decisions on an application's cloud readiness. On this basis, we enable the assessment of parallel applications targeted for cloud migration and support design trade-offs required to optimize these applications with respect to their cloud readiness. Our contribution is threefold:

- We present a novel approach for assessing the cloud readiness of parallel applications based on design decisions. Our assessment is based on a meta model that defines a multi-dimensional scale for measuring the cloud readiness of parallel applications.
- We instantiate a model based on our meta model and investigate essential parallel design decisions with respect to their impact on cloud readiness.
- We describe an extensive case study and discuss the cloud readiness of three parallel applications targeted for cloud migration based on our approach.

This paper is structured as follows. Section 2 describes the problem statement and motivates our work. We present our approach, the underlying meta model, and the assessment procedure in Sect. 3. Sections 4, 5, and 6 comprise an instantiation of our meta model to enable the assessment procedure. We show how to assess several parallel applications in an extensive case study in Sect. 7. Related work is presented in Sect. 8. Finally, Sect. 9 concludes this article.

2 Problem statement

Cloud environments demand specific application characteristics to benefit from rapid elasticity and the pay-per-use model [6,23]. Therefore, cloud-native applications are designed according to the IDEAL properties: isolated state, distribution, elasticity, automated management, and loose coupling [5]. Existing multi-tier applications commonly achieve these properties by decoupling compute from storage [24,25]. Pushing the complexity of application data and state management to storage, enables a stateless compute tier that scales elastically and renders failure recovery a straightforward management task [6].

From a parallel computing perspective, cloud-native applications are typically classified as *trivial parallel* (also called *embarrassingly parallel* or *pleasantly parallel*) [10]. This means that most cloud-native applications process a large amount of independent tasks (e.g., user request), which can be easily distributed among a set of compute components for parallel processing. Since these tasks are not dependent on each other, cloud-native applications tend to be *perfectly scalable*. There is almost no overhead stemming from idle time or communication among components, thus leading to high *parallel efficiency*.

Whereas this architectural approach has been proven to be a good solution for the efficient execution of user requests or independent tasks in general, cloud environments have become more and more attractive for applications from the HPC domain. These applications focus on more tightly coupled parallel tasks and are denoted as *non-trivial parallel*.

In this context, trivial and non-trivial parallelism are best explained by the structure of the *task interaction graph*. The task interaction graph is a graph where each node represents a computational task processed by the application and each edge represents an interaction. Interactions may either result from task dependencies or the mapping of tasks to compute nodes. An application is considered as *trivial parallel* if the task interaction graph does not contain edges. On the other hand, a *non-trivial parallel* application's task interaction graph contains edges. These edges lead to (application-specific and most often complex) interaction patterns among compute nodes.

Since task interactions lead to communication and synchronization among compute nodes, they limit the scalability of non-trivial parallel applications by introducing overhead such as idle time, communication, and excess computation [22]. Due to strict performance requirements, the application state has to be held locally and cannot be pushed to a separate storage tier. Consequently, failure recovery and elasticity management become complex tasks.

Cloud-native [5–8,20] and parallel application design [10,22,26,27] follow conflicting goals. Whereas cloud-native applications are strictly designed according to specific characteristics of cloud environments, parallel application design aims at uncompromising performance maximization in terms of *speedup* and *parallel efficiency*. Trade-off handling is required to consider parallel performance and the character-

istics of cloud environments at the same time. This trade-off handling is key to successfully migrate a parallel application to the cloud.

3 Meta model and cloud readiness assessment

Cloud-native design guidelines [8,20] and cloud pattern languages [5,7] are driven by prescriptive *cloud application properties* (such as the IDEAL properties) 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 designed for HPC workloads 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. Our 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, we apply prescriptive cloud application properties to assess the cloud readiness of parallel applications that are targeted for cloud migration. In the following, we describe our assessment procedure based on the underlying meta model depicted in Fig. 1.

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 [10,27]. Making these design decisions leads to descriptive application characteristics, which we call *parallel application properties*. Explicit documentation of parallel application properties leads to an abstract profile of the application. We call the resulting artifact a *parallel application profile*.

Some parallel application properties correlate with the aforementioned cloud application properties. By systematically correlating both sets of application properties with *property mappings*, we are 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:

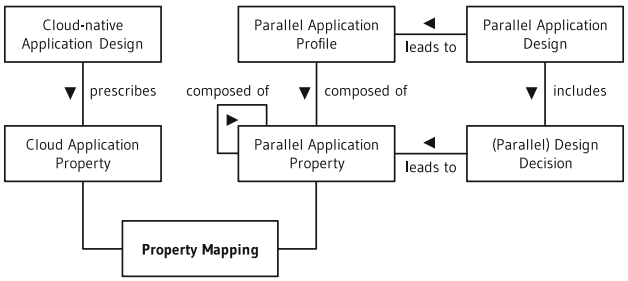

Fig. 1 Meta model that enables the assessment of a parallel application's cloud readiness

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:
$\lambda : M \rightarrow \Psi$, 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 our 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 argument P_{app} and evaluates its cloud readiness based on λ. We define the result of the cloud readiness assessment as the *cloud readiness set* $R \subseteq P_{app} \times C \times \Psi$. Algorithm 1 defines the *cloud readiness assessment* procedure that calculates R based on a given P_{app}.

Our 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. In Sect. 7, we discuss the results of three exemplary cloud readiness assessments based on archetypal parallel applications.

In the following, we construct an instantiation of our meta model that captures the set of cloud application properties C (cf. Sect. 4) and the set of parallel application properties P resulting from parallel design decisions (cf. Sect. 5). Further, we derive the set of property mappings M and specify the graph of the conceptual fitness function λ (cf. Sect. 6).

Algorithm 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
```

4 Cloud application properties

Cloud-native applications are characterized by specific properties that enable them to benefit from cloud environments effectively [5,20]. In the following, we describe the set of *cloud application properties* C 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 [5].

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 increase its performance by adding resources based on horizontal scaling [5,23]. Dynamic adaptation of resources has a huge impact on the cost metric since resources are billed on a pay-per-use basis. The better resources are adapted to the changing workload the more cost efficiency is achievable.

Isolated state (i) Cloud environments typically provide an availability guarantee for the whole platform or infrastructure and not for the components provisioned [5]. 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. We 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 [5]. Both, session state and application state should be managed outside of an application component to enable horizontal scaling. This is typically achieved by decoupling compute from storage [24,25].

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) [5]. 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 [5]. 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 cf. Sect. 7).

5 Parallel application design and properties

Parallel application design has been described in several forms including top-down design processes [22] and pattern languages [10,26,27]. In the following, we summarize the essence of parallel application design in form of parallel design decisions. This section is organized according to these parallel design decisions, each comprising several *parallel application properties*.

Task decomposition To make use of parallelism, typically a single, large problem has to be split into tasks. This process is called task decomposition and represents a fundamental design decision in parallel application design that guides all following design decisions [22]. There are several strategies to decompose a problem into tasks. In *data decomposition*, tasks are partitioned by splitting the data structures they operate on. This is useful for processing large data structures [10]. Depending on the requirements of the application, input, output, or intermediate data can be considered to determine the decomposition [22]. *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 [27]. 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 [22].

Task generation Another important design decision is how task generation is performed. Task generation might either be *static* or *dynamic*. In 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 [10].

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 [22].

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 [10], 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) [26]. 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.

6 Property mappings

In the following, we discuss the set of *property mappings M* and specify the graph of the conceptual fitness function λ. This section is organized according to the cloud application properties (cf. Sect. 4). For each parallel application property within a subsection, the conceptual fitness is discussed in a short text paragraph. Figure 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. This abbreviation ensures better text comprehension.

6.1 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.

6.2 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 [19].

Static task generation (e^-) When static task generation is applied, the amount of tasks is usually matched to available

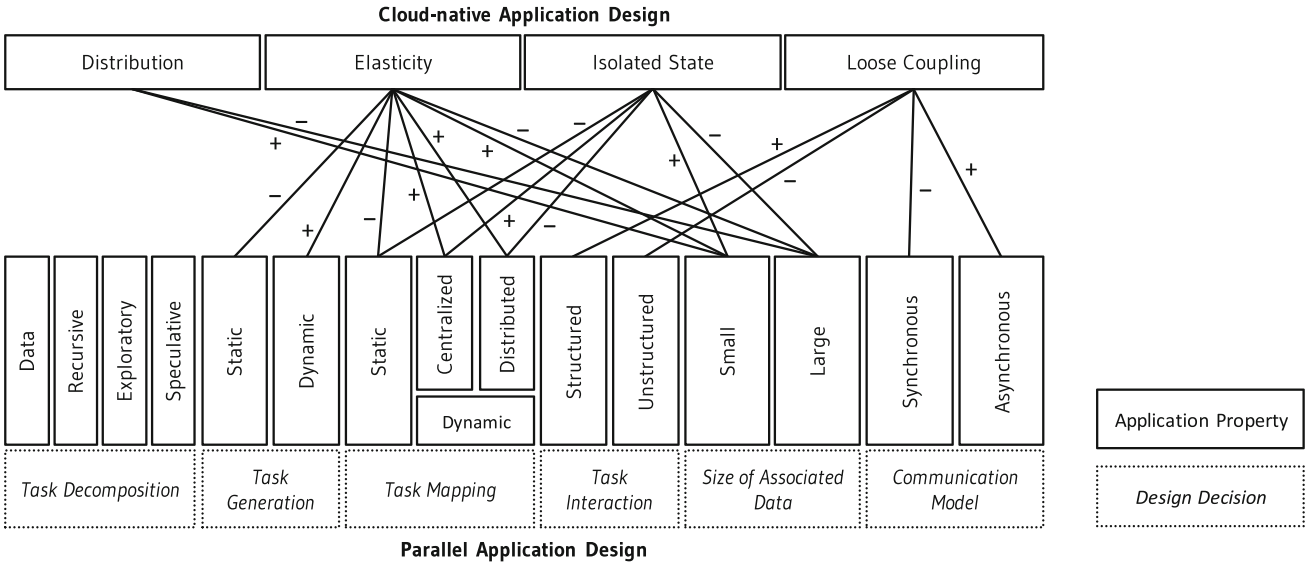

Fig. 2 Property mappings $m \in M$ and their conceptual fitness $\psi \in \Psi$ assigned by λ

compute nodes. Whereas this maximizes parallel efficiency by reducing overhead, it limits elastic scaling of the application as the amount 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 amount 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 does not support elastic scaling. As tasks are inherently bound to components, they cannot be rescheduled across the distributed system. Thus, we 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.

6.3 Isolated state

In HPC, 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. Additionally, HPC environments are not expected to be fault-prone. Thus, parallel applications for these environments have not been developed with resilience and fault tolerance in mind.

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* [22].

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 or external of an application 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.

6.4 Loose coupling

Due to virtualization (leading to CPU timesharing and memory overcommitment), differences with respect to processing times of individual components are the common case in cloud environments. Whereas tight coupling might be more efficient, 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 we can isolate complex interaction patterns between tasks, we 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 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.

Table 1 Parallel application properties of exemplary applications and their impact on cloud readiness

Design decision	SPMD-based MPI app.	MPMD-based MPI app.	Work Queue-based app.
Task decomposition	Data	Data	Data
Task generation	Static (e^-)	Dynamic (e^+)	Dynamic (e^+)
Task mapping	Static (e^-, i^-)	Cent. dynamic (e^+, i^+)	Cent. dynamic (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^+)

7 Case study

In this section, we study several parallel applications for protein folding, a problem widely recognized as grand challenge. 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 model and rapid experimentation.

First, we describe a common method to solve the protein folding problem. Thereafter, we show how to assess the cloud readiness of three archetypal parallel applications employing this method. Table 1 summarizes the corresponding parallel application properties.

7.1 Replica exchange molecular dynamics

Replica exchange molecular dynamics (REMD) is a common approach for the protein folding problem [28,29]. 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.

Numerous simulations are based on ergodic sampling of such energy landscapes. Therefore, Monte Carlo simulations are applied to the configuration space of a protein. Each simulation requires a specific temperature as input. The corresponding energy of a configuration can be evaluated based on classical mechanics. 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 [30]. 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 Monte Carlo simulations, which enhance sampling of the potential energy surface by random walks in energy space, are run in parallel [28]. 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, Monte Carlo simulations are applied to multiple replicas in parallel. Further, the simulation proceeds in a stepwise approach by executing parallel Monte Carlo simulations followed by a replica exchange procedure between neighboring replicas, in which temperature values are exchanged.

Whereas a simulation of n replicas also requires n-times more compute resources, it has been found out that, in some cases, the quality of the simulation increases more than $\frac{1}{n}$-times [30]. This makes REMD an ideal candidate for cloud adoption as added computational resources increase the quality of the simulation superlinearly in such a case.

In the following, we analyze three parallel applications for REMD and assess their cloud readiness.

7.2 SPMD-based MPI application

This application for REMD is based on the Message Passing Interface (MPI) [31]. MPI-1 is still the dominant approach to implement HPC applications. Such programs typically employ the Single Program Multiple Data (SPMD) technique [26,27]. SPMD is a parallel programming style that allows the definition of a single program that is executed by each MPI process.[3] However, at specific regions different branches of instructions are defined and executed only by a subset of the processes. These branches are typically distin-

[3] In MPI jargon, a process is a processing unit that can be distributed across a set of compute nodes.

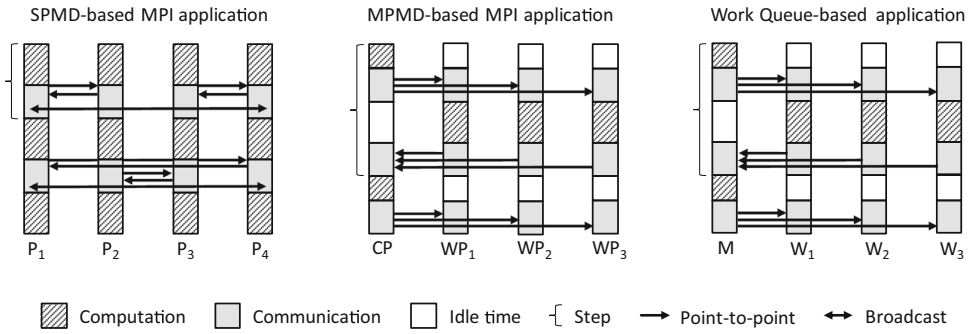

Fig. 3 Computational steps and interaction patterns of exemplary parallel applications

guished by the process id (called rank), which is assigned to each process.

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. Fig. 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.

7.2.1 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*.

7.2.2 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. 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. Fig. 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.

7.3 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 [32]. Moreover, it employs the Multiple Program Multiple Data (MPMD) parallel programming style [33]. 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 Monte Carlo simulations 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. Fig. 3).

7.3.1 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 Monte Carlo 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*.

7.3.2 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 2.0 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 [15] provide such an approach that transforms iterative MPI-based applications into automatically managed, elastic applications [14,15].

7.4 Work Queue-based application

Our third application is a parallel application based on the Work Queue framework [11,34]. 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 Monte Carlo 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. Fig. 3). The master thus coordinates the simulation of multiple workers that can be distributed across compute nodes.

7.4.1 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*.

7.4.2 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 tasks 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.

7.5 Discussion

In this case study, we analyzed 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. Referring to Table 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. Sect. 6).

The results of our 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 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 (e.g., MPI-2). Such an application might be operated in 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 to enable 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, Fig. 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, we accept a decreased parallel performance to ensure an application's cloud readiness.

8 Related work

It exists a growing body of research on how to utilize cloud environments for HPC workloads [2,3,9,11–16,18,19,35–37]. Related work typically measures the performance of existing applications executed in cloud environments. Network virtualization and hardware heterogeneity are identified as main causes of performance degradation [9,12]. Further, the absence of networking hardware optimized for high throughput and low latency as well as hardware accelerators have been described as limiting factors.

With the availability of more and more HPC cloud offerings, at least some of these problems can be resolved effectively [17,18]. Whereas cloud environments have been compared to HPC clusters and supercomputers [38], a comparison of characteristics of applications developed for these environments is missing. We present such an approach that supports the assessment of parallel applications with respect to their cloud readiness. Our 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 non-trivial 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[4] pattern language, which has been created by merging two existing approaches [26,27], 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 and leads to a multi-dimensional design space involving parallel design decisions and cloud-specific considerations.

Several directives how to design parallel cloud applications are given in [11]. Whereas the authors apply these directives to an exemplary HPC application, they do not provide a systematic approach to handle design trade-offs in parallel application design. Similarly, the authors of [21] 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. In this work, we aim at guiding trade-off handling for parallel application design in the context of cloud migration.

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

We thus enable these alternate formulations that are required to benefit from cloud-specific properties.

9 Conclusion and future work

In this paper, we address the problem of migrating a parallel application to the cloud. We identified the understanding of implications on an application's cloud readiness resulting from parallel design decisions as key requirement for successful cloud migration. To support this understanding, we present a systematic approach to assess the cloud readiness of parallel applications. Our approach structures the multi-dimensional design space of constructing parallel applications for cloud environments based on application properties. As a result, we enable the evaluation of parallel design decisions with respect to cloud readiness, which constitutes the basis for cloud migration and provides hints towards optimization of an application's cloud readiness.

Successfully migrating parallel applications to the cloud enables a new breed of cloud services, which address some of the most challenging computing problems, satisfying computational demands in both research and industry. Specifically, parallel applications with unpredictable resource requirements may benefit from elasticity and on-demand resource provisioning. In this context, we plan to investigate container virtualization for deploying parallel applications to the cloud [39,40]. Future research on this topic may transform the cloud into a state-of-the-art execution environment for many applications of significant scientific and practical interest.

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

References

1. Asanovic K, Bodik R, Demmel J, Keaveny T, Keutzer K, Kubiatowicz J, Morgan N, Patterson D, Sen K, Wawrzynek J et al (2009) A view of the parallel computing landscape. Commun ACM 52(10):56–67
2. Varghese B, Buyya R (2018) Next generation cloud computing: new trends and research directions. Future Gener Comput Syst 79:849–861
3. Zhang Q, Cheng L, Boutaba R (2010) Cloud computing: state-of-the-art and research challenges. J Internet Serv Appl 1(1):7–18
4. Mell P, Grance T (2011) The NIST definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg
5. Fehling C, Leymann F, Retter R, Schupeck W, Arbitter P (2014) Cloud computing patterns: fundamentals to design, build, and manage cloud applications. Springer, Berlin
6. Kratzke N, Quint PC (2017) Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. J Syst Softw 126:1–16
7. Fehling C, Leymann F, Retter R, Schumm D, Schupeck W (2011) An architectural pattern language of cloud-based applications. In: Proceedings of the 18th conference on pattern languages of programs, ACM, New York, PLoP '11, pp 2:1–2:11
8. Andrikopoulos V, Binz T, Leymann F, Strauch S (2013) How to adapt applications for the cloud environment. Computing 95(6):493–535
9. Netto MAS, Calheiros RN, Rodrigues ER, Cunha RLF, Buyya R (2018) Hpc cloud for scientific and business applications: taxonomy, vision, and research challenges. ACM Comput Surv (CSUR) 51(1):8:1–8:29
10. Massingill BL, Mattson TG, Sanders BA (2001) Parallel programming with a pattern language. Int J Softw Tools Technol Transf (STTT) 3(2):217–234
11. Rajan D, Canino A, Izaguirre JA, Thain D (2011) Converting a high performance application to an elastic cloud application. In: cloud computing technology and science (CloudCom), 2011 IEEE third international conference on, IEEE, pp 383–390
12. Yang X, Wallom D, Waddington S, Wang J, Shaon A, Matthews B, Wilson M, Guo Y, Guo L, Blower JD, Vasilakos AV, Liu K, Kershaw P (2014) Cloud computing in e-science: research challenges and opportunities. J Supercomput 70(1):408–464
13. Galante G, da Rosa Righi R (2017) Exploring cloud elasticity in scientific applications. In: Antonopoulos N, Gillam L (eds) Cloud computing: principles, systems and applications. Springer, Cham, pp 101–125
14. da Rosa Righi R, Rodrigues VF, Rostirolla G, da Costa CA, Roloff E, Navaux POA (2018) A lightweight plug-and-play elasticity service for self-organizing resource provisioning on parallel applications. Future Gener Comput Syst 78:176–190
15. d R Righi R, Rodrigues VF, da Costa CA, Galante G, de Bona LCE, Ferreto T (2016) Autoelastic: automatic resource elasticity for high performance applications in the cloud. IEEE Trans Cloud Comput 4(1):6–19
16. Gupta A, Kale LV, Gioachin F, March V, Suen CH, Lee BS, Faraboschi P, Kaufmann R, Milojicic D (2013) The who, what, why, and how of high performance computing in the cloud. In: 2013 IEEE 5th international conference on cloud computing technology and science 1:306–314
17. Pellerin D, Ballantyne D, Boeglin A (2015) An introduction to high performance computing on aws: scalable, cost-effective solutions for engineering, business, and science. Amazon Whitepaper. https://d1.awsstatic.com/whitepapers/Intro_to_HPC_on_AWS.pdf. Accessed 9 July 2018
18. Zhang J, Lu X, Panda DKD (2017) Designing locality and numa aware mpi runtime for nested virtualization based hpc cloud with sr-iov enabled infiniband. In: Proceedings of the 13th ACM SIGPLAN/SIGOPS international conference on virtual execution environments. ACM, New York, VEE '17, pp 187–200
19. Galante G, Erpen De Bona LC, Mury AR, Schulze B, da Rosa Righi R (2016) An analysis of public clouds elasticity in the execution of scientific applications: a survey. J Grid Comput 14(2):193–216
20. Leymann F, Breitenbücher U, Wagner S, Wettinger J (2017) Native cloud applications: why monolithic virtualization is not their foundation. Springer, Cham, pp 16–40
21. Parashar M, AbdelBaky M, Rodero I, Devarakonda A (2013) Cloud paradigms and practices for computational and data-enabled science and engineering. Comput Sci Eng 15(4):10–18
22. Grama A (2003) Introduction to parallel computing. Pearson Education, London
23. Toffetti G, Brunner S, Blöchlinger M, Spillner J, Bohnert TM (2017) Self-managing cloud-native applications: design, implementation, and experience. Future Gener Comput Syst 72(Supplement C):165–179
24. Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Ghemawat S, Gubarev A, Heiser C, Hochschild P et al (2013) Spanner:

googles globally distributed database. ACM Trans Comput Syst (TOCS) 31(3):8
25. Verbitski A, Gupta A, Saha D, Brahmadesam M, Gupta K, Mittal R, Krishnamurthy S, Maurice S, Kharatishvili T, Bao X (2017) Amazon aurora: design considerations for high throughput cloud-native relational databases. In: Proceedings of the 2017 ACM international conference on management of data. ACM, pp 1041–1052
26. Massingill BL, Mattson TG, Sanders BA (2007) Reengineering for parallelism: an entry point into plpp for legacy applications. Concurr Comput: Pract Exp 19(4):503–529
27. Keutzer K, Massingill BL, Mattson TG, Sanders BA (2010) 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
28. Sugita Y, Okamoto Y (1999) Replica-exchange molecular dynamics method for protein folding. Chem Phys lett 314(1):141–151
29. Brenner P, Sweet CR, VonHandorf D, Izaguirre JA (2007) Accelerating the replica exchange method through an efficient all-pairs exchange. J Chem Phys 126(7):074,103
30. Earl DJ, Deem MW (2005) Parallel tempering: theory, applications, and new perspectives. Phys Chem Chem Phys 7(23):3910–3916
31. Gropp W, Lusk E, Skjellum A (2014) Using MPI: portable parallel programming with the message-passing interface, 3rd edn. MIT press, Cambridge
32. Gropp W, Thakur R, Lusk E (1999) Using MPI-2: advanced features of the message passing interface. MIT press, Cambridge
33. Foster I (1995) Designing and building parallel programs: concepts and tools for parallel software engineering. Addison-Wesley Longman Publishing Co., Inc., Boston
34. Bui P, Rajan D, Abdul-Wahid B, Izaguirre J, Thain D (2011) Work queue+python: A framework for scalable scientific ensemble applications. In: Workshop on python for high-performance and scientific computing
35. Gupta A, Milojicic D (2011) Evaluation of hpc applications on cloud. In: 2011 Sixth open cirrus summit, pp 22–26
36. Vecchiola C, Pandey S, Buyya R (2009) High-performance cloud computing: a view of scientific applications. In: Pervasive systems, algorithms, and networks (ISPAN), 2009 10th international symposium on, IEEE, pp 4–16
37. Hung DMP, Naidu SMS, Agyeman MO (2017) Architectures for cloud-based hpc in data centers. In: Big data analysis (ICBDA), 2017 IEEE 2nd international conference on, IEEE, pp 138–143
38. Jackson KR, Ramakrishnan L, Muriki K, Canon S, Cholia S, Shalf J, Wasserman HJ, Wright NJ (2010) 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, pp 159–168
39. Kehrer S, Blochinger W (2018) Tosca-based container orchestration on mesos. Comput Sci-Res Dev 33:305–316
40. Kehrer S, Blochinger W (2018) Autogenic: automated generation of self-configuring microservices. In: Proceedings of the 8th international conference on cloud computing and services science (CLOSER), SciTePress, pp 35–46