Hochschule Reutlingen
Reutlingen University

Jens Haussmann[*†]
Wolfgang Blochinger[*]
Wolfgang Kuechlin[†]

[*]Parallel and Distributed Computing Group,
Reutlingen University, Germany
E-mail: {jens.haussmann,wolfgang.blochinger}@reutlingen-university.de

[†]Symbolic Computation Group,
University of Tuebingen, Germany
E-mail: {jens.haussmann,wolfgang.kuechlin}@uni-tuebingen.de

Cost-Efficient Parallel Processing of Irregularly Structured Problems in Cloud Computing Environments

Manuscript
Dec. 2018

BibTeX

```
@Article{Haussmann2018,
        author="Haussmann, Jens and Blochinger, Wolfgang and Kuechlin, Wolfgang",
        title="Cost-Efficient Parallel Processing of Irregularly Structured Problems
        in Cloud Computing Environments",
        journal="Cluster Computing",
        publisher="Springer",
        year="2018",
        month="Dec",
        day="06",
        issn="1573-7543",
        doi="10.1007/s10586-018-2879-3",
        url="https://doi.org/10.1007/s10586-018-2879-3"
}
```

The final publication is available at Springer via https://doi.org/10.1007/s10586-018-2879-3

Original Publication Reference:
https://link.springer.com/article/10.1007/s10586-018-2879-3

Cost-Efficient Parallel Processing of Irregularly Structured Problems in Cloud Computing Environments

Jens Haussmann[*†] · **Wolfgang Blochinger**[*] · **Wolfgang Kuechlin**[†]

Abstract In this paper, we deal with optimizing the monetary costs of executing parallel applications in cloud-based environments. Specifically, we investigate on how scalability characteristics of parallel applications impact the total costs of computations. We focus on a specific class of irregularly structured problems, where the scalability typically depends on the input data. Consequently, dynamic optimization methods are required for minimizing the costs of computation. For quantifying the total monetary costs of individual parallel computations, the paper presents a cost model that considers the costs for the parallel infrastructure employed as well as the costs caused by delayed results. We discuss a method for dynamically finding the number of processors for which the total costs based on our cost model are minimal. Our extensive experimental evaluation gives detailed insights into the performance characteristics of our approach.

Keywords High Performance Distributed Computing · Cloud Computing · Parallel Computing · Cost Model · Irregularly Structured Problems

1 Introduction

In recent years cloud computing evolved to a mature computing paradigm, attracting more and more attention from both industry and academia. In particular, cloud computing has turned out to be a very promising technological and organizational path heading the utility computing vision.

Also from the perspective of parallel processing, cloud computing can be of considerable interest. For example, one can easily employ typical IaaS (Infrastructure-as-a-Service) cloud offerings to construct (virtual) parallel environments that share many characteristics with on-site compute clusters built of commodity hardware. While the cloud-based solution permits pay-per-use and elasticity of resources, traditional cluster computing requires considerable upfront investment and allows only static scaling by manually adding extra cluster nodes. Thus, cloud computing enables highly scalable parallel execution environments (e.g., regarding the number of utilized processors) that permit immediate, fine-grained cost control.

However, overall scalability of a parallel system is often limited by the characteristics of the considered parallel algorithm. With an increasing number of processors also the parallel overhead (i.e., processor idling, excess computation, and communication) increases, resulting in decreasing parallel efficiency. Ultimately, this means that at some scale we are not able to transform money spent for additional computing resources into an adequate benefit, i.e., increased speedup of computation or a higher processing rate. Cloud-based parallel computing environments allow us to better cope with this situation by optimizing the total costs of individual computations. In this context, we consider the total costs of a computation to be comprised of the money spent on the parallel computing resources and the costs caused by delayed or less accurate results of computations.

In this paper, we apply these principles to so-called *irregularly structured problems*. This class of parallel ap-

[*]Parallel and Distributed Computing Group,
Reutlingen University, Germany
E-mail: {jens.haussmann,wolfgang.blochinger}@reutlingen-university.de

[†]Symbolic Computation Group,
University of Tuebingen, Germany
E-mail: wolfgang.kuechlin@uni-tuebingen.de

plications is characterized by varying and unknown task sizes as well as highly unstructured communication patterns. Moreover, the overall scalability of a computation highly depends on the input data and also cannot be determined by a (semi-) static analysis. As a consequence, we need an entirely dynamic method for finding the number of processors for which the total costs are minimal.

In this paper we make the following contributions:

- We introduce a cost model for parallel computations that considers the costs of the cloud infrastructure employed as well as the application-specific opportunity costs for delayed results.
- Based on the cost model, we discuss an optimization method for minimizing the costs of computing irregularly structured problems, which relies on adaptive parallelism.

The remainder of the paper is organized as follows: In Section 2, we give a brief characterization of irregularly structured problems along with a discussion of suitable parallelization techniques. Section 3 discusses our cost model for parallel computations in cloud environments. In Section 4, we present our optimization method for dynamically minimizing the costs of computation. An overview of the cloud-based architecture and implementation of our method is the topic of Section 5. Section 6 reports on the results of our performance evaluation. Section 7 discusses related work.

2 Parallel Execution of Irregularly Structured Problems

Irregularly structured problems can be found in various application domains like astrophysics, fluid dynamics, system modeling and simulation, computer graphics, exhibiting a different manifestation and also a different degree of irregularity. They are characterized by highly input-dependent and unstructured computation and communication patterns [48]. Our work focuses on task-parallel irregularly structured problems (in contrast to data-parallel ones, like sparse-matrix computations).

One important class of task-parallel irregularly structured problems are heuristic search applications based on state space search where (feasible) solutions are represented as paths in a tree-shaped state space [17]. Prominent application domains are artificial intelligence (e.g., [11], [26]), electronic design automation (e.g., [36], [35], [10]) as well as discrete optimization (e.g., [23], [16]).

Geared by a heuristic, a state space tree is explored by dynamically expanding tree nodes heading for a (optimal) solution. For this kind of problems, it is in general possible to decompose the whole problem into several tasks for parallel execution (i.e., employing exploratory decomposition techniques). These tasks represent a subtree of the state space. However, one cannot control or even estimate the size of the individual tasks. As a result, the task sizes of a parallel computation can vary considerably (cf. Figure 1). In order to minimize processor idling, dynamic problem decomposition and consequently dynamic load balancing become necessary. Thus, during computation tasks are continuously generated (by splitting the search space of existing tasks) and transferred to idle processors. Typically, this process leads to highly dynamic computation and communication patterns, where the degree of dynamism reflects the degree of irregularity.

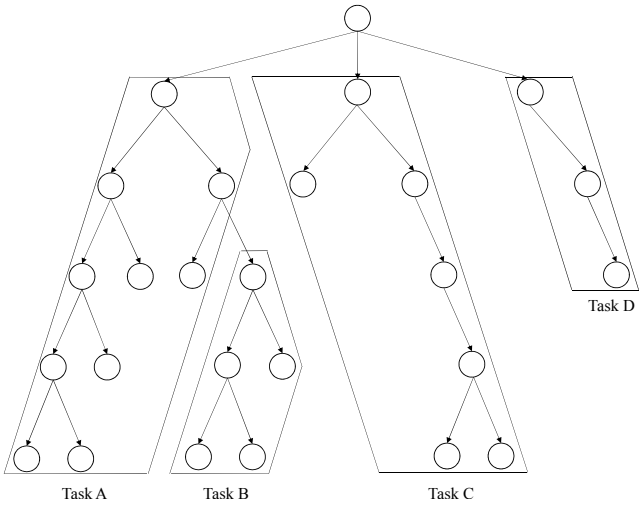

Fig. 1 Exploratory search space decomposition.

The discussed parallelization techniques require a (distributed) task pool execution model [24]. A task pool manages the tasks created during computation and is responsible for serving tasks to idle processors. Basically, the task pool model can be implemented in a centralized or distributed fashion. A central task pool is located at a single node, accessible by all processors. This approach is only feasible if the pool is rarely accessed because otherwise it becomes a bottleneck. In contrast, in the distributed task pool approach each processor maintains a local pool accessible by all other processors (cf. Figure 2), resulting in a balanced access distribution and increased scalability. For transferring tasks between the distributed pools, sender-initiated schemes (i.e., work-sharing) or receiver-initiated schemes (i.e., work-stealing) triggered by threshold values on the pool size can be employed. We decided to employ work-

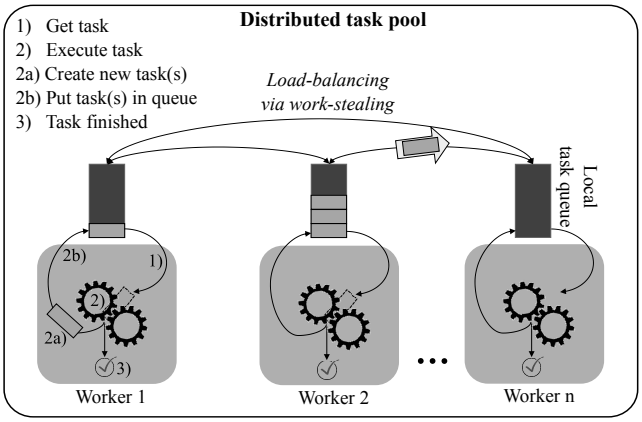

Fig. 2 Load-balancing and task processing in a distributed task pool.

stealing since it has been proven that the system load does not grow unbounded with time, making work-stealing a stable load balancing algorithm [7].

For the considered application class a high degree of irregularity results in high overhead (in the form of processor idling, excess computation, or communication), leading to poor scalability of the parallel system. Moreover, the computations degree of irregularity is input dependent such that the same holds for the scalability behavior.

3 Cost Model

In this section, we deal with the question of quantifying the monetary costs of computations, performed in cloud-based parallel environments featuring pay-per-use billing. A fundamental observation underlying our work is that the scalability of a parallel application can influence the total monetary costs to a significant degree. Intuitively one can say: For a highly scalable problem it is more profitable to utilize additional processors than for a poorly scalable one. (Note that in this paper we focus on strong scaling, meaning that the problem size remains constant with an increasing number of processors). To formally define the term *profitable* we introduce a cost model for cloud-based parallel computations.

3.1 Derivation of a Cost Model for Parallel Cloud Computing

A cost model is a simple description of a system, usually represented by a set of mathematical equations that converts a number of resources or defined input parameters into cost data [51]. Hence, a cost model is an abstraction of a real-world system, implying that not every aspect of the underlying system is captured.

Employing our cost model, our overall goal is to find the number of processors for which the total monetary costs of a parallel computation are minimal. As stated in Section 2, we are focusing on problems where the scalability highly depends on the input data. This prevents any upfront optimization, e.g., based on performance data gathered from previous program runs. Hence, dynamic optimization becomes inevitable. To meet this requirement, it is crucial to minimize the runtime overhead, in particular concerning collecting the model's input data and performing the actual optimization. Consequently, the cost model has to abstract from the rather complex pricing schemes of cloud providers, such as the ones of *Amazon AWS* [1] or *Google Cloud Platform* [5]. Keeping also in mind that we are dealing with compute-intensive parallel applications (in contrast to data-intensive ones), our cost model focuses on resources that are fundamental for executing this class of applications. Thus, we explicitly model the costs for processors and communication, while main memory and hard disks are treated implicitly.

In the context of parallel computing the costs C are usually defined as $C = T_p * p$, where T_p is the parallel runtime using p processors [24]. Following this common definition, costs are expressed in terms of time. In contrast, the costs of executing cloud applications are typically expressed in terms of monetary units. A simple cloud cost model can be found in [25], where the model from parallel computing is extended towards pay-per-use, therefore $C = T_p * p * c_\pi$, where c_π is the cloud provider's price for a single processor per time unit. More sophisticated cost models for cloud computing also take into account the costs for communication as well as storage [40], [49], [15].

To derive a cost model that meets our requirements, it seems natural to integrate established cost models of cloud and parallel computing in a straightforward manner. However, this would result in a multi-objective optimization problem, encompassing the two conflicting objectives *fast processing* versus *low monetary costs*. Multi-objective optimization leads to a set of *Pareto optimal* solutions, imposing the burden of selecting an appropriate solution to the user. This is conflicting with our requirement stated above that optimization must be accomplished automatically while the computation is in progress. Converting all objective functions of the model into a single *aggregated objective function* is one way to overcome this situation. An aggregated objective function implies that all objective goals are converted and aggregated into the same objective by using suitable methods [38].

To derive an aggregated objective function, we apply the concept of *opportunity costs*. This general concept of business economics describes the lost profits that could be made by an alternative investment of a scarce resource [27]. Opportunity costs belong to the implicit costs, modeled as an addend of the total costs. For example, consider a $1M budget investment decision: Leave the money on the bank account or invest it into a new production hall. By taking the first option, one will miss the opportunity of profits from the sale of produced goods. On the other hand, by taking the second option, one will miss the opportunity of returns on interest from the bank. In the same sense opportunity costs can be adapted to a computing infrastructure, treating *time* as a scarce resource. An intuitive example might be an engineer who has to wait for a computation to finish and cannot proceed with his work in an optimal manner until the results of the computation are available. In this situation, the opportunity costs could be expressed regarding the salary of the engineer, for the time he is waiting for the results. Also, consider the production processes of a manufacturer. Production workflow scheduling might be done in a naive manner, resulting in non-optimal schedules. Here, the opportunity costs can be expressed as overhead, compared to the optimal solution, which would, however, be more time consuming to compute. Both examples have shown that opportunity costs are highly application-specific. Moreover, for a given application they can also vary depending on the situation or the point in time. Consider the given example of an engineer for two distinct situations: 1) The engineer has additional lower priority tasks to do while waiting for the computation to finish. 2) The engineer has no more tasks left while waiting for the computation to finish. In the first situation, the opportunity costs are much smaller since the simultaneously working of the engineer generates additional incomes.

Motivated by our observations, we propose the *cost function* given in Equation 1 to formalize our cost model for parallel cloud computing. Subsequently, we discuss all constituents of the cost function in detail.

$$C(p) = C_{cp}(p) + C_{com}(p) + C_{op}(p) \tag{1}$$

The parameter p represents the number of (virtual) processors allocated from the cloud provider. The number of processors is our only controlling lever since generally all scaling operations of a parallel computing infrastructure are supposed to change the level of parallelism while keeping the relative capacity of all other resource parameters (such as memory) constant.

The *computing costs* $C_{cp}(p)$ are modeled in an analogous manner to [25].

$$C_{cp}(p) = T_p * p * c_\pi \tag{2}$$

This cost component expresses the costs that are charged for the virtual computing infrastructure. We assume that each virtual processor is coupled with an appropriate size of main memory and hard disk. The constant c_π expresses the costs for a single virtual processor per time unit, including the costs for main memory and hard disk. Note that T_p and p correlate in a specific way, representing the scalability behavior of the problem.

The *communication costs* $C_{com}(p)$ are modeled as follows:

$$C_{com}(p) = T_p * B_p * c_\theta \tag{3}$$

Generally, the degree of communication, and therefore the communication cost of a parallel application, depends on the number of processors p. However, the exact relationship is highly application-specific. We model this relationship using the *communication rate* B_p (in terms of average bytes received per time unit), as an application-specific dependency of p. Accordingly, the overall *communication volume* of a parallel application is $T_p * B_p$. The costs for a single byte received are denoted as c_θ. (Note that currently for some cloud providers $c_\theta = 0$ holds true.) Communication costs are an example for illustrating a typical difference between cost models for cloud and parallel computing. In traditional parallel computing, communication is treated as constituent part of the overhead of a computation that increases the parallel runtime and thus contributes to the (computing) costs. However, in cloud computing communication causes additional costs billed by the cloud provider for transferring data over communication links between the nodes. Thus, communication contributes to several cost components.

Finally, the last cost component models the *opportunity costs* $C_{op}(p)$ for a parallel cloud computation.

$$C_{op}(p) = T_p * c_\omega \tag{4}$$

As discussed above, the opportunity costs of the computation are expressed regarding the parallel runtime and the constant c_ω denoting the lost opportunities of a simultaneous missed alternative per time unit.

Based on the cost functions defined in Equations 1 to 4 our overall goal is to find the number of processors for which the total costs $C(p)$ are minimal.

3.2 Exemplary Application of the Cost Model

In principle, our cost model is applicable for any compute-intensive parallel cloud application. However, in our work, we employ this model in the context of task-parallel, irregularly structured problems. Thus, we take a closer look at our cost model by examining Equations 1 to 4 in the light of the distributed task pool execution model. To better understand the characteristics of our cost model, we discuss three illustrative exemplary problems with different scalability: A perfectly parallelizable problem, a non-parallelizable problem, and a realistic problem, that falls in between the two extremes. For each problem, we set the price for a processor to $c_\pi = \$0.2/h$ and for communication to $c_\theta = \$0.1/GB$. These prices are in accordance with typical offerings of public cloud providers, like Amazon AWS [1] and Google Cloud Platform [5]. As discussed above, opportunity costs are highly application-specific. For the purpose of demonstration, we assume costs of $c_\omega = \$1/h$. Moreover, for all three problems we assume a sequential runtime of $T_1 = 12h$ and model the parallel runtime T_p by Amdahl's law: $T_p = \beta * T_1 + \frac{(1-\beta)*T_1}{p}$. The three example problems are characterized by a different sequential fraction $0 \leq \beta \leq 1$. Furthermore, we assume a distributed task pool execution environment that employs a round-robin work-stealing load balancing mechanism. In our model, the load balancing messages constitute the communication rate B_p. We assume that a worker sends one message every 5 ms on average when running idle and each message has a size of 1 KB. According to Amdahl's law, parallel overhead origins solely from sequential fractions of a computation, resulting in idle time. Overall idle time can be calculated by the difference of the total time spent in solving a problem in parallel and the time spent performing useful work: $T_p * p - T_1 = \beta * T_1 * (p-1)$. Thus, we can determine the communication volume $T_p * B_p$ by $\frac{\beta * T_1 * (p-1)}{5ms} * 1KB$, and therefore $B_p \simeq \frac{\beta * T_1 * (p-1)}{5ms} * 1KB * \frac{1}{T_p}$.

We define the *perfectly parallelizable* problem by $\beta = 0$, hence $T_p = \frac{T_1}{p}, \forall p$. Consequently, every processor is exclusively performing essential computations with no manifestation of parallel overhead. Thus, parallel efficiency remains constant at 1.0 for every number of processors p. For the task pool execution model, this holds true for a computation where each processor computes a single task of equal size. This also means that no load balancing takes place. Thus, the number of bytes received is zero at any time. Figure 3 shows the cost constituents of Equation 1, as well as the resulting total costs $C(p)$ of the perfectly parallelizable problem.

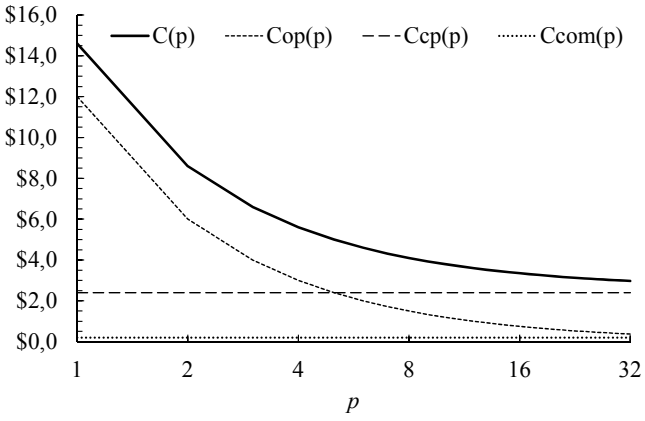

Fig. 3 Costs of a perfectly parallelizable problem for $p = 1 \ldots 32$.

One can see continuously decreasing total costs $C(p)$, with stabilization for an increasing number of processors p. This downward trend is heading towards \$2.40, which can be explained by the reduction of the opportunity costs. The runtime T_p decreases in proportion to number of processors p, therefore $C_{op}(p)$ converges towards zero. Consequently, this means that the computing costs $C_{cp}(p)$ and communication costs $C_{com}(p)$ together are the limit for the minimum total costs $C(p)$ of the problem. Another insight from this graph is that $C_{cp}(p)$ remains constant for perfectly parallelizable problems. This means that the problem can either be calculated with p processors in time t or with t processors in time p for the same costs. Summing up all cost constituents reveals an execution with minimal total cost by using the largest possible number of processors.

The *non-parallelizable* problem is characterized by $\beta = 1$, hence $T_p = T_1, \forall p$. This means that we have a single serial task that is executed in the task pool. Considering the communication overhead due to work-stealing messages from load balancing for such a setting, the number of received bytes per time unit B_p increases for an increasing number of processors p. According to the definition of B_p above and $T_p = T_1, \forall p$ this results in $B_p \simeq \frac{(p-1)}{5ms} * 1KB$. Hence, at any point in time, $p-1$ workers are requesting tasks from other workers, leading to a linear growth of the bytes received from $B_1 = 0GB/h$ to $B_{32} = 21.29GB/h$. Figure 4 shows the cost constituents of Equation 1 as well as the resulting total costs $C(p)$ of the *non-parallelizable* problem.

Figure 4 is in stark contrast to the previously discussed Figure 3. In the case of a non-parallelizable problem, $C_{op}(p)$ remains constant because the parallel runtime T_p cannot be reduced through additional processor capacities. Since T_p remains constant and B_p grows linearly, $C_{com}(p)$ grows linearly in the same way. There-

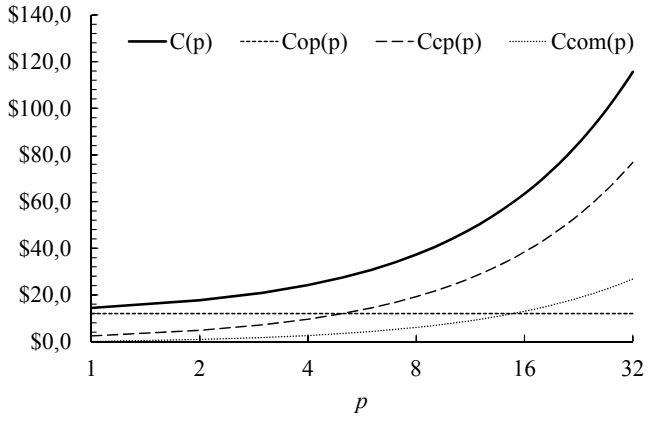

Fig. 4 Costs for a non-parallelizable problem for $p = 1 \ldots 32$.

fore, $C(p)$ will increase continuously due to idling overhead of the surplus processors as well as the communication overhead for load balancing. Since there are no cost benefits at all by using multiple processors, $p = 1$ is the cost minimal computing infrastructure.

Next, we consider a *realistic* problem, which we model by $\beta = 0.05$. Consequently, $T_p = 0.6h + \frac{11.4h}{p}$. For our task pool execution model, this means processors compete for available tasks, or the workload of transferred tasks is quite small compared to the overhead of moving tasks. Therefore, it is characterized by a flattening decrease of the parallel runtime T_p as the number of processors p increases. As before, we determine the communication rate based on idle time. Accordingly, the rate of received bytes has again a nearly linear growth from $B_1 = 0GB/h$ to $B_{32} = 13.3GB/h$. (Note that these values are in line with our experimental results from Section 6.3.2). Figure 5 shows the cost constituents of Equation 1 as well as the resulting total costs $C(p)$ of the *realistic* problem.

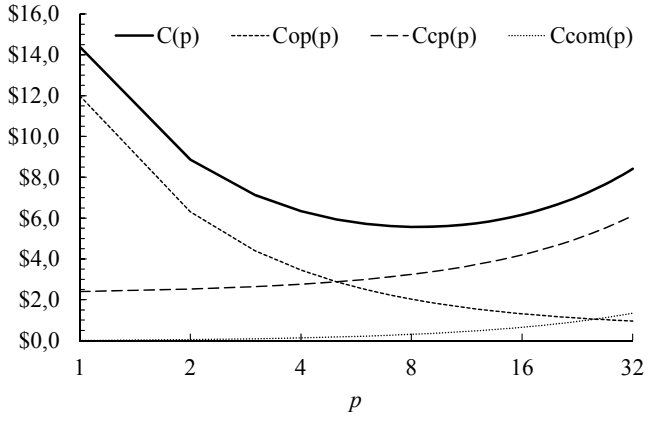

Fig. 5 Costs for a realistic problem for $p = 1 \ldots 32$.

Using the insights from the previous discussion of the perfectly- and non-parallelizable problems allows us to explain the characteristics of the resulting line graphs in this figure: Up to a certain number of processors, which is $p = 8$ in the case at hand, it is profitable to utilize additional processors since the reduction of opportunity costs outweighs the increased computing infrastructure costs. However, exceeding this boundary increases the total costs with every additional processor.

A characteristic that plays a significant role for the cost optimization method discussed in the next section is the unimodal shape of our cost function, which means that it exhibits only a single minimum. This results from the unimodality of the three constituent cost-functions and their continuous flattening/increasing growth rate. The gained saving of opportunity costs of every new processor gets smaller with increasing number of processors, whereas communication and computing costs increase more for every additional processor.

4 Heuristic Cost Optimization

Using the cost function discussed in the previous section, we can determine the total costs $C(p)$ of a parallel cloud computation with p processors by measuring the parallel runtime T_p. This implies that the computation has been completed. However, our primary goal is to employ the cost function to find the most cost-efficient number of processors for a computation, hence, choosing p so that the total costs $C(p)$ are minimal. In principle, this scenario requires a priori knowledge of the correlation of p and T_p, i.e., the scalability characteristics of the application at hand.

There are several ways of gaining insight into the scalability characteristics. If an application class shows a reproducible scalability behavior for different computations, one can measure T_p for all relevant numbers of processors p using a prototypical application and input data. The resulting cost curve can be used for optimizing future parallel computations of this application. However, this approach is not feasible for applications that exhibit an unpredictable interdependence between input values and scalability, like some classes of irregularly structured problems, which we are focusing on in this paper. As a consequence, for such applications, it is inevitable to capture the scalability characteristics at runtime by an appropriate heuristic. We accomplish this by applying the concept of *adaptive parallelism*, i.e., by measuring and analyzing the impact of adaptations of the number of utilized processors during parallel computations. This results in a set of measurements of runtime data for different numbers of proces-

sors (hereinafter referred to as *probe points*), which we employ for approximating the cost curve and finding the most cost-efficient number of processors. Generally, for determining the scalability characteristics there is a conflict between the accuracy of the approximation and the resulting overhead. On the one hand, a higher number of probe points results in a more accurate picture of the computation's scalability and the resulting cost curve. On the other hand, a higher number of probe points results in a higher overhead since more time of the computation is spent on utilizing a non-optimal number of processors. To deal with this situation, we employ a heuristic, which is responsible for approximating the scalability and the computation's cost curve and at the same time finds the most cost-efficient number of processors. Thus, our approach enables cost driven auto-scaling for parallel cloud computations that are characterized by an unknown scalability behavior.

4.1 Dynamic Cost Approximation

In order to be able to dynamically approximate the cost function $C(p)$ of a parallel computation, some adaptations need to be carried out. One obvious issue is that the values for the parallel runtime T_p and the communication volume $T_p * B_p$ are only available after a parallel program run with p processors is completed. We overcome this situation by considering the parallel efficiency $E(p)$. The parallel efficiency of a computation represents the fraction of time spent on essential work. The remaining time constitutes the parallel overhead resulting from communication, excess computation, or processor idling. Hence, we substitute the parallel runtime according to $T_p = \frac{T_s}{E(p)*p}$ [24] in Equations 2 to 4, delivering Equation 5. As we will discuss below, the efficiency can be approximated by taking performance probes at runtime. Considering the parallel efficiency also allows an estimation of the communication costs $C_{com}(p)$, provided that the communication rate B_p is known. In line with the probing mechanism discussed below, we can also estimate the value of B_p.

$$C(p) = \underbrace{(\frac{T_s}{E(p)} * c_\pi)}_{C_{cp}(p)} + \underbrace{(\frac{T_s}{E(p)*p} * B_p * c_\theta)}_{C_{com}(p)} + \underbrace{(\frac{T_s}{E(p)*p} * c_\omega)}_{C_{op}(p)} \quad (5)$$

Note that the sequential runtime T_s does not affect the location of the cost minimum, but only the value of the minimum. For finding the most cost-efficient number of processors the actual value of T_s is irrelevant (thus we can assume w.l.o.g. $T_s = 1$).

Algorithm 1 Probe

Output:
$\widetilde{C}_{p,t}$ - Normalized costs for current number of processors

1: $p \leftarrow$ *currentNumberOfProcs()*
2: $t \leftarrow$ *currentTime()*
3: **array** *efficiency[p]*
4: **array** *communication[p]*
5: **for** i=0 **to** p-1 **step** +1 **do**
6: *efficiency[i]* $\leftarrow$ *Probe efficiency of processor* π_i
7: *communication[i]* $\leftarrow$ *Probe comm. rate of processor* π_i
8: **end for**
9: $\widetilde{E}_{p,t} \leftarrow$ *avg(efficiency[])*
10: $\widetilde{B}_{p,t} \leftarrow$ *sum(communication[])*
11: $\widetilde{C}_{p,t} \leftarrow (\frac{1}{\widetilde{E}_{p,t}} * c_\pi) + (\frac{1}{\widetilde{E}_{p,t}*p} * \widetilde{B}_{p,t} * c_\theta) + (\frac{1}{\widetilde{E}_{p,t}*p} * c_\omega)$

To estimate the parallel efficiency, we measure the time that a worker-thread of our task pool occupies a processor and periodically compare this time to the time elapsed. This is accomplished by instrumenting the program code of the task pool at the corresponding locations, allowing us to take probes with a varying duration d on all participating processors. The arithmetic mean of all probes from the same point in time t delivers the approximated efficiency $\widetilde{E}_{p,t}$. With this technique, two constituents of overhead, namely communication, and idling, can be captured. An analysis of our task pool implementation revealed that excess-computation is negligible compared to the other constituents of overhead. Monitoring the communication rate of the parallel computation is done in a similar way, i.e., by probing the transmitted data of all communication sockets, delivering the approximated communication rate $\widetilde{B}_{p,t}$.

Note that the duration d of a probe may influence the quality of our method: A long duration can lead to more accurate and reliable results but limits the flexibility of making fast decisions, whereas a short duration may be inaccurate and unreliable but leads to more flexibility.

Algorithm 1 formalizes the described procedure for approximating the costs. It is executed by a coordinator process that monitors the workers of the task pool under control of our optimization method, which we discuss next.

4.2 Dynamic Cost Optimization

The primary challenge our optimization method has to cope with is to perform scaling operations based on limited knowledge about the shape of the cost function of the current computation. Basically, we continuously

extend this knowledge by monitoring the application's efficiency and communication rate for different numbers of processors. To keep the overhead low, it is crucial to check only as few as possible such probe points because each probe causes overhead. Actually, in the course of the parallel computation we vary the number of processors used, take efficiency and network probes, calculate the resulting approximated costs and use the results to steer the overall optimization process.

Specifically, our approach is based on the *ternary search method* that recursively finds the minimum of a unimodal function f [31]. Figure 6 illustrates the principal idea behind this approach for two different example functions. In each iteration we evaluate f for two probe points x_a and x_b that are located between the borders x_{start} and x_{end} of the considered search interval. Actually, this divides the search interval into the three intervals $[x_{start}, x_a]$, $[x_a, x_b]$, and $[x_b, x_{end}]$. By comparing $f(x_a)$ and $f(x_b)$ we can either drop $[x_{start}, x_a)$ or $(x_b, x_{end}]$ and recursively proceed with the remaining two intervals. For example, consider a situation where $f(x_a) < f(x_b)$ holds. Note, that in Figure 6 this is the case for both example functions. For a unimodal function it is obvious that the minimum cannot be in the interval $(x_b, x_{end}]$. However, it is not clear whether the minimum is in interval $[x_{start}, x_a]$ or in interval $[x_a, x_b]$. For similar reasons, we can drop the interval $[x_{start}, x_a)$ when $f(x_a) > f(x_b)$ holds.

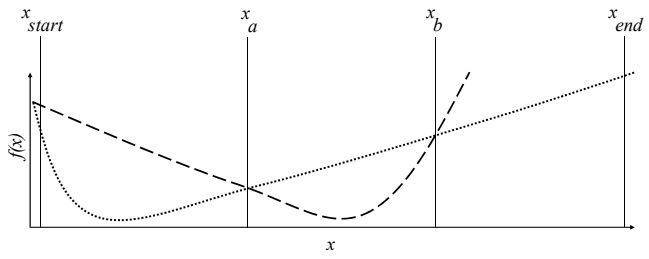

Fig. 6 Search intervals of ternary search method.

Algorithm 2 shows how we employ the ternary search technique for minimizing the cost function of a running parallel computation. It is executed by a coordinator process concurrently to the worker processes of the task pool that perform the actual computation. Basically, it monitors the state of the worker processes employing the previously discussed Algorithm 1 for taking probes and triggers scaling actions using Algorithm 3.

According to the ternary search method, two probe points are taken in every iteration for narrowing the search interval. These correspond to two different numbers of processors executing worker processes for which we take efficiency and network probes and evaluate our cost function.

Algorithm 3 dynamically scales the number of processors used for worker processes. Increasing the number of processors is trivial for our task pool execution model, where new processors just start participating in the computation by stealing tasks (cf. *AddToPool()* in Algorithm 3, Line 15). However, decreasing the number of processors involves multiple steps to guarantee consistency (cf. *RemoveFromPool()* in Algorithm 3, Line 5). Specifically, we have to ensure that a processor owns no tasks before it is removed, to prevent losing tasks. For this, we have to stop the current task of the processor, cancel all pending steal requests, and migrate all remaining tasks to other processors.

We can shut down a processor (i.e., stop billing) if the search process reaches a state where it is guaranteed that this processor does not take part anymore. This applies whenever the search procedure drops the last third of the current search interval, as seen in Line 23 - 27 of Algorithm 2. However, if there is uncertainty as to whether a processor is part of subsequent search intervals, we keep the processor in a standby pool for later use (cf. Line 6 of Algorithm 3).

4.3 Phases of the Computation

In our approach, the parallel computation can be in one of the following two phases: *Scaling phase* or *stable phase*. In the scaling phase, the optimization procedure previously discussed takes place (cf. Algorithms 1 to 3). Once the number of processors with optimal costs is found the stable phase starts, keeping the number of processors at a constant value. Nevertheless, taking efficiency and communication probes for evaluating the cost function is continued (cf. Algorithm 1). In case of a significant change of the costs $\widetilde{C}_{p,t}$, a transition to another scaling phase needs to be carried out in order to ensure the accuracy of our method. As a result, in the course of a computation, a continuous alternation of both phases is taking place.

During the stable phase, we only know the recent progression of the costs $\widetilde{C}_{p,t}$ for the current number of processors. Generally, we cannot make any assumptions regarding the future evolution of $\widetilde{C}_{p,t}$, whether for the current number of processors nor for all other numbers of processors. One possibility to deal with this situation is to preventively switch to a new scaling phase at fixed time intervals.

However, performing phase transitions can be accomplished more effectively, provided that the application at hand allows additional assumptions on the runtime characteristics. For certain parallel applications,

Algorithm 2 Search

Input:
p_{start} - Start point of search interval,
p_{end} - End point of search interval
1: **persistent array** *costs[maxNumberOfProcs()+1]*
▷ Probes already taken. Position 0 is unused
2: $p_a \leftarrow \left\lceil \frac{p_{end}-p_{start}}{3} \right\rceil + p_{start}$
3: $p_b \leftarrow p_{end} - \left\lfloor \frac{p_{end}-p_{start}}{3} \right\rfloor$
4: **if** *costs*$[p_a]$ == ***undefined*** **then**
5: *Scale*(p_a); *costs*$[p_a] \leftarrow$ *Probe()*
6: **end if**
7: **if** *costs*$[p_b]$ == ***undefined*** **then**
8: *Scale*(p_b); *costs*$[p_b] \leftarrow$ *Probe()*
9: **end if**
10: **if** $p_{end} - p_{start} \leq 3$ **then**
11: *Scale*(p_{start}); *costs*$[p_{start}] \leftarrow$*Probe()*
12: *Scale*(p_{end}); *costs*$[p_{end}] \leftarrow$*Probe()*
13: $c_{min} \leftarrow \infty$
14: **for** i=p_{start} **to** p_{end} **step** +1 **do**
15: **if** *costs*$[i] < c_{min}$ **then**
16: $c_{min} \leftarrow$ *costs*$[i]$
17: $p_{min} \leftarrow i$
18: **end if**
19: **end for**
20: *Scale*(p_{min})▷ Minimum found, algorithm terminates
21: **return**
22: **end if**
23: **if** *costs*$[p_a] <$ *costs*$[p_b]$ **then**
24: **for** i=p_b **to** $p_{end} - 1$ **step** +1 **do**
25: *Shutdown*(π_i)
26: **end for**
27: *Search*(p_{start}, p_b)
28: **else if** *costs*$[p_a] >$ *costs*$[p_b]$ **then**
29: *Search*(p_a, p_{end})
30: **end if**

Algorithm 3 Scale

Input:
targetNumberOfProcs - Target number of processors
1: $p \leftarrow$ *currentNumberOfProcs()*
2: **persistent list** *standbyList* ▷ Procs. in standby mode
3: **if** *targetNumberOfProcs* $< p$ **then** ▷ Remove procs. from task pool
4: **for** $i = p - 1$ ***to*** *targetNumberOfProcs* **step** -1 **do**
5: *RemoveFromPool*(π_i)
6: *standbyList.add*(π_i)
7: **end for**
8: **else** ▷ Add procs. to task pool
9: **for** $i = p$ ***to*** $targetNumberOfProcs - 1$ **step** +1 **do**
10: **if** *!standbyList.isEmpty()* **then**
11: *standbyList.remove*(π_i)
12: **else**
13: *Boot*(π_i)
14: **end if**
15: *AddToPool*(π_i)
16: **end for**
17: **end if**

we can assume that $\widetilde{E}_{p,t}$ decreases as the computation progresses, whereas $\widetilde{B}_{p,t}$ increases. Often, this holds for applications that perform dynamic problem decomposition since the granularity of created tasks is continuously becoming smaller, resulting in a larger degree of parallel overhead. Moreover, we can assume that if the parallel overhead of any number of processors increases, then the parallel overhead of all greater number processors increases by at least the same.

As a consequence, a change of $\widetilde{E}_{p,t}$ and $\widetilde{B}_{p,t}$ can shift the cost minimum only to a lower number of processors since the values $\widetilde{E}_{p+1,t}$ and $\widetilde{B}_{p+1,t}$ both have deteriorated to the same extent. Based on these assumptions we can restrict the search interval after a transition to a new scaling phase to a number of processors lower than the current number.

A transition to a scaling phase requires a trigger that signals a possible change of the most recently approximated cost curve, and therefore a potential new location of the cost minimum. Throughout the stable phase, the continuously updated values of $\widetilde{C}_{p,t_n}$ are compared to the value $\widetilde{C}_{p,t_m}$ that was taken at the beginning of the current stable phase. Furthermore, the user defines a parameter *sensitivity value* s that specifies the percentage increase of $\widetilde{C}_{p,t_m}$ that triggers a transition. Thus, if the condition in Equation 6 is detected, a transition to a new scaling phase takes place.

$$\widetilde{C}_{p,t_m} * (1 + s) \leq \widetilde{C}_{p,t_n} \tag{6}$$

4.4 Costs of Approximation

In this section, we investigate on the costs resulting from a parallel computation using our optimization method. These costs, called C_{total}, include all additional costs owed to our dynamic optimization process and differ from the costs $C(p)$ (cf. Equation 1), which represent the costs of a computation with a constant number of processors p. We can assume that the optimization process starts with a number of processors that is generally not equal to the cost minimum number of processors. Hence, it results in additional costs, compared to the cost minimum.

Figure 7 shows an example of the heuristic's operation. It illustrates the varying number of utilized processors during computation as well as the corresponding phases. The total runtime of a computation that is optimized by our heuristic is denoted as T_{total}.

We can see, that during the initial scaling phase all scaling operations are heading towards $p = 7$. The cost minimal number of processors is found with a small cost overhead since the set of possible solutions is continuously reduced. In the shown example the initial solution is found after nine probes in a set of 32 possible

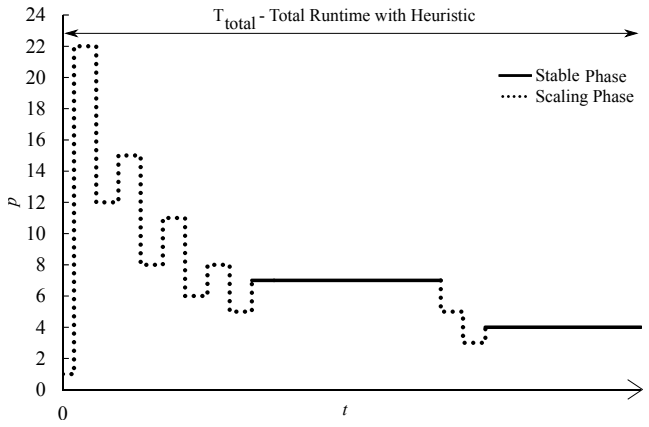

Fig. 7 Number of processors p utilized during scaling and stable phase.

solutions. We employ Equation 7 to determine the total costs C_{total} of a program run optimized by our method.

$$C_{total} = \left(\sum_{i=0}^{p_{max}-1} T_{\pi_i} * c_\pi\right) + (T_{total} * c_\omega) + \left(\sum_{i=0}^{p_{max}-1} T_{\pi_i} * B_{\pi_i} * c_\theta\right) \quad (7)$$

This equation considers the basic constituents of our cost model (cf. Equation 1-4), taking into account the varying number of processors employed during the computation (p_{max} denotes the maximum number of processors used). The costs for the computing resources can be calculated by considering the time span between boot and shutdown (and thus billed time) T_{π_i} of each processor π_i. The opportunity costs are calculated analogously to Equation 4 by using T_{total}. Finally, the costs for communication are determined by accumulating the communication volume $T_{\pi_i} * B_{\pi_i}$ of each processor π_i.

5 Architecture and Implementation

To put our work into practice, we implemented a Java-based prototype of a distributed task pool and our optimization method. In this section, we give a brief overview of the architecture of our prototype, along with the corresponding components and their role. Figure 8 shows the overall architecture that is organized into two layers. The foundation forms an *OpenStack* based *IaaS* cloud [42]. *OpenStack* is currently the most widely-adopted open-source cloud platform. However, all methods discussed in our paper can also be easily implemented on top of other IaaS cloud offerings like *Amazon AWS* or *Google Cloud Platform*.

Basically, we employ two OpenStack IaaS components for our implementation: The monitoring service (called *Ceilometer*) and the compute service (called *Nova*). *Ceilometer* keeps track of the worker's metrics, whereas *Nova* executes the worker's life-cycle operations.

The tenant's domain (located on top of the IaaS layer) operates the worker VMs of the elastic task pool and a VM that acts as a controller by managing the elastic task pool as discussed in Section 4. The entire communication between provider and tenant is accomplished by using *OpenStack4j* [3], an open-source client library for *OpenStack*.

Next, we show the interaction of the architecture components within the scope of a single iteration of Algorithm 2. The order of interactions is highlighted in Figure 8. During computation, each worker sends its metrics (efficiency and communication rate) at predefined time intervals to the monitoring service (step 1). The master, in turn, polls these metrics (step 2), analyzes them and calculates the resulting monetary costs of the current execution environment (step 3). Based on this information, a scaling decision is made and sent to the compute service (step 4). To prevent inconsistencies in the task pool, all workers have to be informed about the scaling decision. Subsequently, the compute service adapts the execution environment, either by booting new or shutting down existing workers (step 5). At this point, a single iteration comes to its end, followed by next iterations in the same way.

We conclude this section with a brief outline of the implementation of our distributed task pool, as shown in Figure 8. Each worker maintains a synchronized shared queue, intended for tasks that can be stolen by idle workers. The thief chooses its victim in a round-robin procedure. Provided that all tasks have been completed, the computation's termination needs to be detected. Detecting termination in a distributed computation like in our task pool execution environment is a non-trivial task since every worker has only a partial knowledge of remaining tasks. Thus, for our prototypical implementation, we employed a central approach, where the coordinator maintains a state list of all tasks.

6 Experimental Evaluation

6.1 Experimental Platform and Setup

The experiments presented in this section were conducted on a private cloud hosted at Reutlingen University. This OpenStack based cloud provides a platform for researchers and students supporting various research and teaching activities. All experiments were

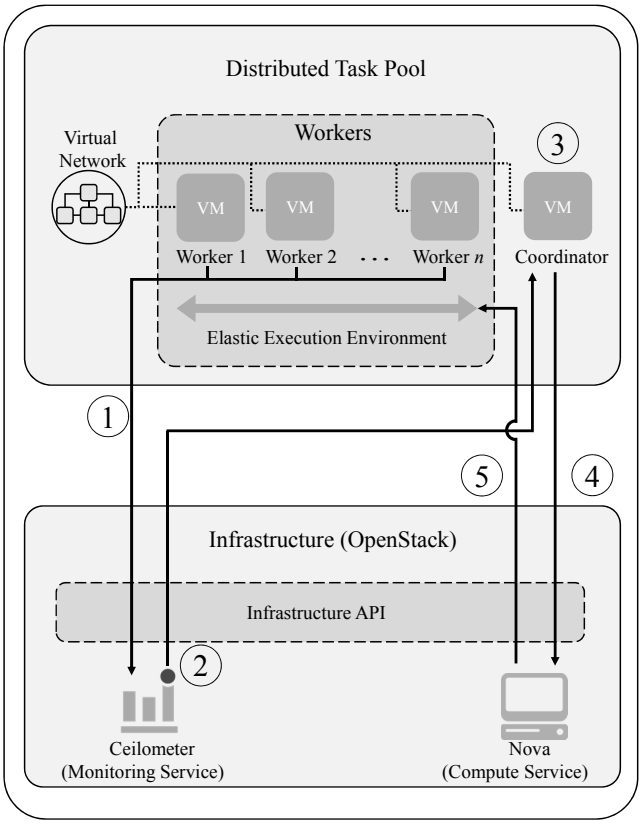

Fig. 8 System architecture and interaction of components.

performed during regular multi-tenant operation of the cloud.

The underlying hardware consists of identically configured servers, each equipped with two Intel Xeon E5-2650v2 CPUs and 128 GB main memory. All CPUs have eight cores operating at 2.6 GHz. The servers are connected by multiple networks separating management, storage, and tenant VM communication. Specifically, all self-service virtual networks connecting tenant VMs are operated on a 10 GBit/s physical ethernet network.

For executing our elastic distributed task pool (cf. Figure 8), we utilize identical virtual machines with 1 vCPU and 2 GB main memory for hosting workers. Thus, in our experiments, elasticity is based on horizontal scaling, i.e., the number of processors p employed directly translates to the number of virtual machines. The coordinator is executed on a VM with 2 vCPUs and 4 GB main memory. All VMs are connected by a self-service virtual network.

As discussed in Section 3.2, we set the cost parameters to $c_\pi = \$0.2/h$, $c_\omega = \$1/h$, $c_\theta = \$0.1/GB$. We assume fine-grained billing, with a granularity of seconds and megabyte, respectively. This corresponds to pricing models currently offered by major cloud providers like *Amazon AWS* [1] or *Google Cloud Platform* [5].

6.2 Methodology

Our goal is to perform an extensive experimental evaluation of the presented concepts by employing exemplary problems that exhibit a high degree of irregularity. As stated in Section 2, this particularly holds for the class of *state space search* problems. Algorithms for this class of problems implicitly construct a search tree where each node represents a state, edges constrain state properties, and paths in the tree represent results. During computation, tree nodes are selected (often employing a heuristic) and expanded to child nodes in order to find profitable paths possessing desired properties. As a result, the shape and size of the tree highly depend on the input data and cannot be estimated by a (semi-)static analysis. In case of parallel computation, the varying and unpredictable sizes of subtrees lead to load imbalance, which in turn significantly affects scalability.

State-of-the-art implementations of representative state space search problems, like Boolean satisfiability or discrete optimization, typically incorporate randomization, most often leading to highly varying parallel runtimes for the same input [9], [46], [8]. Moreover, we cannot directly control the degree of irregularity and thus the scalability. This renders a systematic performance investigation of our method unfeasible. For this reason, we employ a generic state space search application (denoted *GSSSA*) exhibiting all relevant characteristics of state space search applications, particularly w.r.t. parallel execution. Moreover, it permits to explicitly control the irregularity and therefore the scalability by means of a small set of parameters. In Section 6.3 we first give a detailed description of *GSSSA*. Subsequently, we report on a detailed investigation of the characteristics of our optimization method enabled by the *GSSSA* approach.

To see how our optimization method performs with a real-world application, we conducted an additional performance analysis, using different variations of the well-known *Traveling Salesman Problem* (TSP). The results are discussed in Section 6.4.

6.3 Generic State Space Search Application

6.3.1 Problem Description

For constructing a generic state space search application we first have to identify their common properties with respect to parallel execution. Common to parallel execution of state space search problems is the principle of exploratory problem decomposition: Nodes resulting from one expansion (i.e., having a common parent node) can be allocated to different tasks for parallel

processing. Each task, in turn, contains all state data to process the respective subtree, enabling a parallel construction of individual subtrees. The workload generated during expansion depends on the application. Typically, this is a CPU intensive workload, i.e., checking if the current state fulfills defined conditions. However, scalability usually depends on the (ir)regularity, i.e., the balancing of workload, of subtrees resulting from expansions. In general, we can assume that each expansion falls either in the category *regular* or *irregular* and the ratio of both determines the scalability.

GSSSA is based on a binary search tree where each node is associated with a workload value w representing the (remaining) workload modeled as w random SHA-1 hash calculations. Tree nodes can either expand to two child nodes, partitioning the workload of the parent according to specific rules (see below) or act as leaves, performing the remaining hash calculations. The search tree is comprised of two separate subtrees, both originating at the root. One subtree represents the regular fraction and the other the irregular fraction (cf. Figure 9). We model this structure by starting the computation with two nodes, for which the workload is defined by the parameter w_r and w_i, respectively. Thus, we can assign a weighting to the regular and irregular fraction. Additionally, the irregular fraction of the tree is associated with a balancing parameter b, specifying the workload partitioning between expanded child nodes. The regular fraction of the tree is associated with a fixed balancing value of 0.5, i.e., expansion leads to two child nodes each having half of the parent's workload. To limit the granularity of node expansion, the global granularity parameter g specifies the smallest allowed workload of a node. Table 1 provides an overview of all parameters of *GSSSA*.

In a concrete application, the mixture of regular and irregular trees would manifest itself in subtrees randomly, which is in contrast to our *GSSSA* approach, where expansions of nodes in the same subtree always result in more subtrees, which are either regular or irregular. However, this does not limit our approach since parallel execution leads to the desired mixture and distribution of regular and irregular subtrees across all participating workers.

Each parameter affects the structure of the resulting search space differently. Together they allow a precise control of characteristics of parallel execution like scalability while providing a generic problem that exhibits no difference in parallelization compared to any specific state space search problem. For example, the degree of irregularity, and thus the scalability, is controlled by the balancing parameter, whereas the initial workload (w_r and w_i) controls to which extent the overall tree structure is affected. To allow a broad scope of experiments, we employed *GSSSA* to generate three problem instances with different scalability characteristics (cf. Table 2).

Name	*Description*	*Range*
w_r	Workload of regular fraction	$\mathbb{N}$
w_i	Workload of irregular fraction	$\mathbb{N}$
b	Balancing of workload between child nodes in irregular fraction	$[0.0, 0.5]$
g	Granularity, i.e., smallest allowed workload of a node	$\mathbb{N}$

Table 1 Overview of parameters used for *GSSSA*.

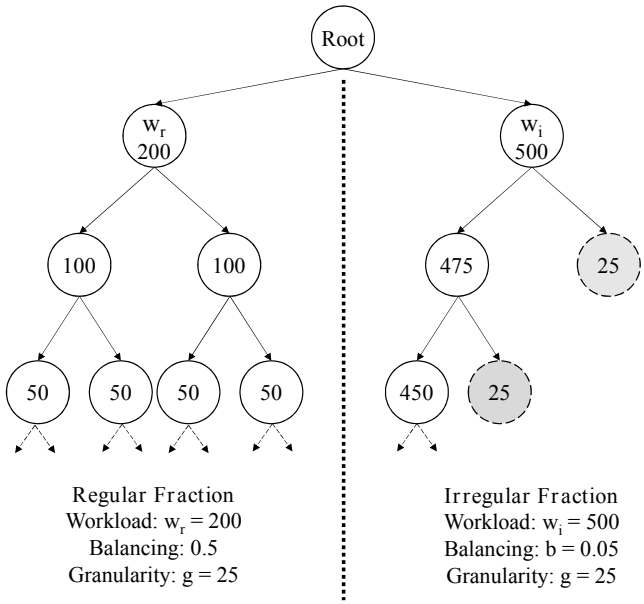

Fig. 9 Exemplary structure of an implicitly constructed *GSSSA* search tree.

6.3.2 Analysis of Example Problems

First of all, we analyze parallel computations of the three example problems presented in Table 2. The resulting insights are intended to be twofold: We want to show that the computations yield to reproducible and controllable scalability characteristics. Secondly, we will use the results later as reference values, enabling us to draw a comparison between computations both with and without employing our optimization method. To

Name	*Trend of overhead*	w_r	w_i	b	g
GSSSA_IW	weakly increasing	5 000 000 000	15 000 000 000	0.001	1 000 000
GSSSA_IS	strongly increasing	1 000 000 000	19 000 000 000	0.0002	1 000 000
GSSSA_C	constant	0.0	20 000 000 000	0.0	1 000 000

Table 2 Description of three *GSSSA* example problems used for experimental evaluation.

this end, we carried out each computation three times with a fixed number of processors, for every number of processors p and calculated the arithmetic mean of the parallel runtime T_p, and communication volume $T_p * B_p$. Based on these values, we determined the corresponding costs $C(p)$, as defined in Equation 1. The graphs in Figure 10 show each cost constituent of $C(p)$ (for a fixed number of processors p, $p = 1 \dots 32$) for computations of *GSSSA_C*, *GSSSA_IW*, and *GSSSA_IS*, whereas the corresponding runtime metrics and the coefficient of variation are shown in Table 3.

The results in Table 3 show that we obtain the typical characteristics of parallel computations with *GSSSA*. On the one hand, we can see that the parallel runtime decreases, whereas communication overhead increases, as the degree of parallelism increases. However, the gained runtime reduction is getting continuously smaller for each additional processor. Moreover, the characteristics demonstrate that each example problem exhibits a different scalability. Furthermore, the data in Table 3 show that all metrics have a small variation, which further indicates that *GSSSA* fulfills the stated requirements.

Regarding Figure 10 we observe a broad spectrum of costs $C(p)$ for each example problem, which is attributable to the significant impact of the scalability. Note that the curves' shapes match the *realistic problem* scenario discussed in Section 3.2, confirming their validity. Comparing the runtime T_p in Table 3 of *GSSSA_IW* and *GSSSA_IS* shows a much poorer scalability of the latter. Adding more computing capacities minimizes costs until a certain number of processors is reached, from which costs increase. This turning point, which also is the most cost-effective number of processors, is moving to a higher number of processors for scalable computations. One can see that the most cost-effective number of processors p is located at quite different positions, namely $p = 6$ with \$1.17, $p = 10$ with \$0.75, and $p = 7$ with \$0.96, for *GSSSA_C*, *GSSSA_IW*, and *GSSSA_IS*, respectively. The maximum value of each curve in the search interval is located at the upper or lower limit of the interval, more precisely at $p = 32$ with \$3.43, $p = 1$ with \$2.47, and $p = 1$ with \$2.44, for *GSSSA_C*, *GSSSA_IW*, and *GSSSA_IS*, respectively. By utilizing additional processors beyond the cost minimum, the user would not only pay for inefficiently used computing resources; moreover, the additional resources induce more costs due to an increasing communication overhead, which is billed separately. This is particularly evident for the metrics of *GSSSA_IS*: Increasing the number of processors beyond the most cost-efficient point ($p = 7$) yields to almost no reduction of the runtime, whereas the resulting communication volume increases significantly. Consequently, the costs of *GSSSA_IS* increase drastically for $p > 7$. In contrast, the costs of *GSSSA_IW* increase only slightly beyond the most cost-efficient point ($p = 10$), leading to a different shape of the cost curve.

Following on from these results, we will take a closer look at the approximated runtime characteristics of our three example problems (cf. Figure 11). By continuously taking and analyzing probes for a fixed number of processors ($p = 32, 16, 8$), we get an insight into the data series of $\widetilde{E}_{p,t}$ and $\widetilde{B}_{p,t}$. Computations of *GSSSA_IW* and *GSSSA_IS* result in an increasing degree of overhead as computation progresses. Hence, utilizing a constant number of processors, the approximated efficiency $\widetilde{E}_{p,t}$ decreases over time, while the approximated communication rate $\widetilde{B}_{p,t}$ increases. This is in contrast to computations of *GSSSA_C* that result in an constant degree of overhead, i.e., constant values of $\widetilde{E}_{p,t}$ and $\widetilde{B}_{p,t}$ as computation progresses. Based on $\widetilde{E}_{p,t}$ and $\widetilde{B}_{p,t}$, we can determine the trend of approximated costs $\widetilde{C}_{p,t}$ in dependence of the time. Note that we have to multiply the approximated costs $\widetilde{C}_{p,t}$ with the sequential runtime T_s since the approximated cost curve is based on $T_s = 1$.

The results in Figure 11 give evidence that *GSSSA* possess the typical runtime characteristics of parallel state space search problems. As the graphs on the left side imply, both *GSSSA_IW* and *GSSSA_IS* have a trend of decreasing efficiency $\widetilde{E}_{p,t}$ and increasing communication rate $\widetilde{B}_{p,t}$ as the computation is progressing. Further analysis reveals a noticeable impact of the used parameters of *GSSSA* on the runtime behavior: The degree to which the efficiency decreases depends on the value of the balancing parameter b and the number of utilized processors. Since *GSSSA_IS* has a much smaller balancing parameter compared to *GSSSA_IW*, exploratory decomposition is highly irregular, leading

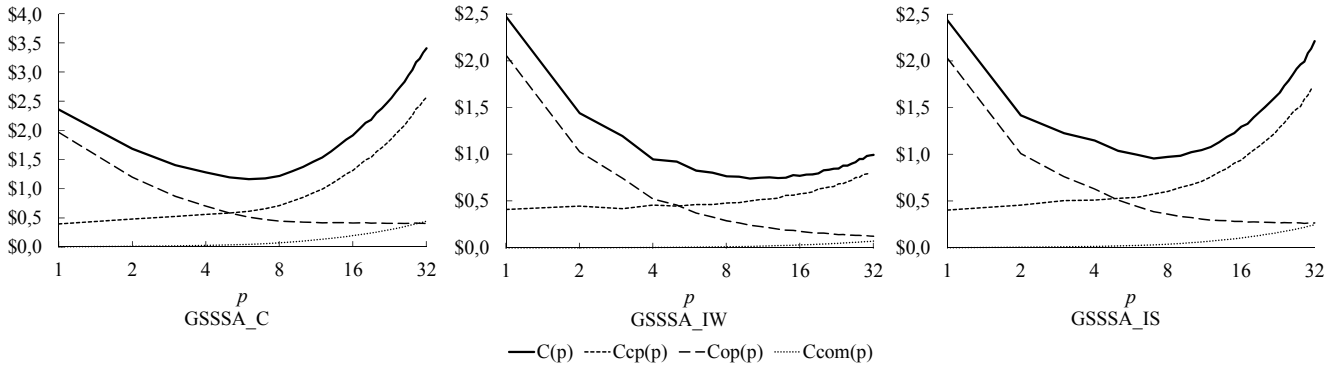

Fig. 10 Costs of *GSSSA_C*, *GSSSA_IW*, and *GSSSA_IS*, executed with different fixed numbers of processors p.

	Arithmetic mean						Coefficient of variation					
	Runtime - T_p [sec]			Comm. - $T_p * B_p$ [GB]			Runtime - T_p			Comm. - $T_p * B_p$		
	GSSSA_...			GSSSA_...			GSSSA_...			GSSSA_...		
p	C	IW	IS	C	IW	IS	C	IW	IS	C	IW	IS
1	7589	7404	7309	0.000	0.000	0.000	0.009	0.008	0.012	0.000	0.000	0.000
2	4417	3632	3634	0.097	0.016	0.058	0.024	0.012	0.004	0.010	0.052	0.002
4	2554	1864	2260	0.235	0.045	0.140	0.010	0.027	0.038	0.005	0.025	0.012
6	1845	1338	1583	0.419	0.079	0.244	0.008	0.008	0.010	0.004	0.007	0.009
8	1632	1040	1287	0.680	0.123	0.381	0.007	0.020	0.020	0.002	0.013	0.004
10	1560	878	1158	0.984	0.166	0.544	0.007	0.010	0.021	0.011	0.021	0.005
12	1517	777	1067	1.297	0.217	0.712	0.018	0.020	0.006	0.005	0.005	0.002
14	1526	686	1042	1.603	0.264	0.881	0.003	0.009	0.004	0.006	0.013	0.008
16	1479	633	1017	1.926	0.319	1.049	0.002	0.021	0.009	0.004	0.009	0.005
18	1494	586	996	2.231	0.370	1.226	0.002	0.013	0.007	0.001	0.006	0.006
20	1501	566	982	2.539	0.423	1.416	0.011	0.016	0.009	0.004	0.016	0.007
22	1476	534	968	2.849	0.471	1.574	0.003	0.009	0.015	0.004	0.006	0.003
24	1470	518	968	3.148	0.524	1.750	0.009	0.023	0.016	0.005	0.004	0.006
26	1452	494	965	3.481	0.575	1.992	0.002	0.006	0.010	0.008	0.015	0.006
28	1460	483	956	3.807	0.628	2.109	0.002	0.026	0.012	0.002	0.004	0.005
30	1459	466	954	4.090	0.675	2.289	0.008	0.019	0.021	0.002	0.005	0.009
32	1454	448	950	4.404	0.734	2.465	0.008	0.007	0.017	0.006	0.004	0.008

Table 3 Runtime metrics - arithmetic mean and coefficient of variation of *GSSSA_C*, *GSSSA_IW*, and *GSSSA_IS*.

to unbalanced tasks. This, in turn, results in a low parallel efficiency and an increasing communication rate. Applied to the approximated costs $\widetilde{C}_{p,t} * T_s$, shown on the right side, this results in a steady increase for *GSSSA_IS*, whereas *GSSSA_IW* abruptly reaches its peak. Furthermore, we see that *GSSSA_C* has quite constant values for $\widetilde{E}_{p,t}$, $\widetilde{B}_{p,t}$ and $\widetilde{C}_{p,t} * T_s$ as computation is progressing. For *GSSSA_C*, the values of $\widetilde{E}_{p,t}$ and $\widetilde{B}_{p,t}$ match closely the values of $E(p)$ and B_p at any point in time. In particular these values are $E(32) = 0.16$, $E(16) = 0.32$, $E(8) = 0.58$, and $B_{32} = 3.00[MB/sec]$, $B_{16} = 1.30[MB/sec]$, and $B_8 = 0.42[MB/sec]$.

6.3.3 Approximation of Cost Function

An important finding of our work is the implicit approximation of the cost function $C(p)$ that is continuously carried out during computation by taking probes. We determine the accuracy $\widetilde{A}_{p,t}$ of a single probe by comparing the approximated costs $\widetilde{C}_{p,t} * T_s$ against the actual costs $C(p)$, i.e., $\widetilde{A}_{p,t} = |C(p) - \widetilde{C}_{p,t} * T_s|$. The arithmetic mean $\overline{\widetilde{A}}$ of the accuracies $\widetilde{A}_{p,t}$ of all probes during a scaling phase reflects the accuracy of approximating the cost function $C(p)$.

Moreover, we determine the reliability $\widetilde{R}_p$ of a specific approximation point by using the standard deviation of all cost approximations performed for this point. The arithmetic mean $\overline{\widetilde{R}}$ of all reliabilities $\widetilde{R}_p$ of approximated points reflects the reliability of approximating the cost function $C(p)$.

The primary factor influencing an approximation's quality is the duration d for taking performance probes (cf. Section 4.1). On the one hand, we are interested in short durations to retrieve approximations quickly. On the other hand, a short duration may result in poor accuracy and poor reliability. Therefore, we inves-

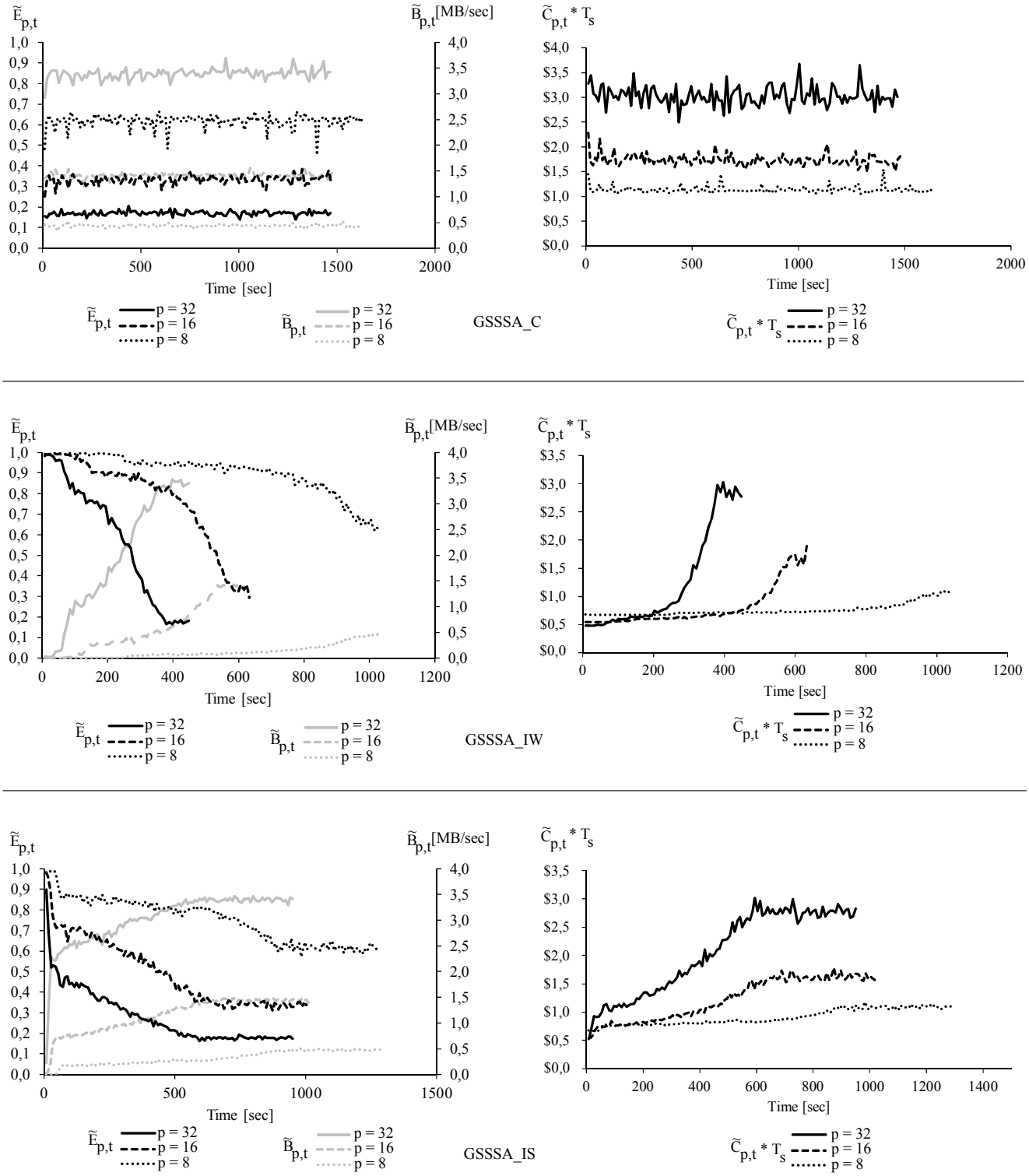

Fig. 11 Data series of the probing process and the corresponding approximated efficiency, communication and costs of *GSSSA_C*, *GSSSA_IW* and *GSSSA_IS*.

tigate next the effects of different durations of probes on the approximation. To achieve this, we analyze the initially approximated costs $\widetilde{C}_{p,t} * T_s$ for computations of *GSSSA_C*. This problem can be regarded as a special case since it is one with a constant degree of overhead over time. Hence, the data underlying the approxima-

tion process (measured efficiency $\widetilde{E}_{p,t}$ and measured communication rate $\widetilde{B}_{p,t}$) remain constant, independently from the progress of computation. Consequently, only one scaling phase is carried out since the resulting values of $\widetilde{C}_{p,t} * T_s$ are still current at any time during computation. This is in contrast to *GSSSA_IW* and *GSSSA_IS* where $\widetilde{C}_{p,t}$ varies over time, making a direct comparison of $C(p)$ and $\widetilde{C}_{p,t} * T_s$ less conclusive.

The graphs in Figure 12 visualize the accuracy and reliability for different durations of probes. Initially noticeable are the large intervals with missing points of approximated costs in all cases. This is owed to the proceeding of our optimization method, where the search interval is shortened by only taking a few probes, instead of probing the whole search interval. The measurements reveal that the accuracy decreases for a decreasing probe duration d. Moreover, the measurements also reveal a significantly decreasing reliability for durations $d < 5$ sec. Based on these results we see that a duration longer than 5 sec only marginally improves the approximations accuracy and reliability.

6.3.4 Estimation of Runtime Parameters

At this point, we want to analyze to what extent the values of the sensitivity s and the probe duration d influence the total costs C_{total} of a computation employing our optimization method. First, we focus on the probe duration d. In Section 6.3.3 we already discussed the influence of the probe duration on the quality of approximations. Figure 13 shows the resulting total costs C_{total} of the computations performed in the last section for different values of the probe duration d.

The results reveal that our optimization method performs most cost-efficient with probe durations of both $d = 5$ sec and $d = 1$ sec within the precision of measurements. However, considering the results of Section 6.3.3, we opt for a probe duration of $d = 5$ sec as the more favorable choice due its greater approximation quality.

Next, we analyze the effects of different sensitivity values s (cf. Equation 6), which control the transition into a scaling phase. For this analysis we employed the example problem *GSSSA_IS*. As previously seen in Figure 11, *GSSSA_IS* is a more challenging problem compared to *GSSSA_IW* since the latter has just a small time interval in where $\widetilde{C}_{p,t} * T_s$ increases. We conducted three program runs of *GSSSA_IS*, using sensitivity values of $s = 5\%$, $s = 100\%$, and $s = 500\%$. Furthermore, we monitored the point in time, the type (scale out or in), and the extent of all scaling operations. The results are shown in Figure 14.

As anticipated, employing small sensitivity values like $s = 5\%$ initiates many transitions into a scaling phase. Looking at Figure 14, we see that seven scaling phases were performed for this sensitivity value, whereas for a sensitivity value of $s = 500\%$ only an initial scaling phase is carried out. Consequently, a scaling phase and the resulting approximation of the costs have a more stable character for higher sensitivity values. This, in turn, results in resistance against temporary events that change the parallel overhead. Such events may interfere with individual probes, resulting in cost overhead due to unnecessary transitions into a scaling phase. However, at the same time, higher sensitivity values restrict the dynamics of the optimization method: In the scaling phase, once a most cost-effective number of processors is found, it is used for a longer period.

Figure 15 shows the resulting total costs C_{total} of computations of *GSSSA_IW* and *GSSSA_IS* for sensitivity values between $s = 5\%$ and $s = 500\%$. The graphs show that the sensitivity value greatly influences the resulting costs C_{total}. In general, we see a trend of increasing costs as well as a more significant variation between individual results for increasing sensitivity values. This holds true both for *GSSSA_IW* and *GSSSA_IS*. This correlation is attributed to the lowered frequency of performed scaling phases for higher sensitivity values, limiting the flexibility of our optimization method. Moreover, the graphs show that the costs C_{total} of *GSSSA_IW* increases much slower compared to *GSSSA_IS*. Since *GSSSA_IW* has a better scalability, the costs of utilizing a greater number of processors differ only slightly from the costs of the most cost-effective number of processors. Computations of *GSSSA_IW* are less sensitive to limited flexibility of the optimization method. Hence, C_{total} increases only slightly for higher values of the trigger sensitivity. Furthermore, in our experiments we observed that sensitivity values $s < 5\%$ caused a behavior where stable phases end immediately, effectively leading to continuous scaling operations. Thus, we can conclude that a sensitivity of $s = 5\%$ is best suited for our optimization method.

6.3.5 Analysis of Overall Cost Efficiency

In this concluding section of our experimental evaluation of *GSSSA*, we evaluate the overall cost-efficiency of our optimization method. Without using our method, computation generally would be performed utilizing a constant number of processors that is selected by the user. We employ the most cost-efficient and most cost-inefficient possible scenario of this selection as comparative data for our evaluation. This, in turn, allows to de-

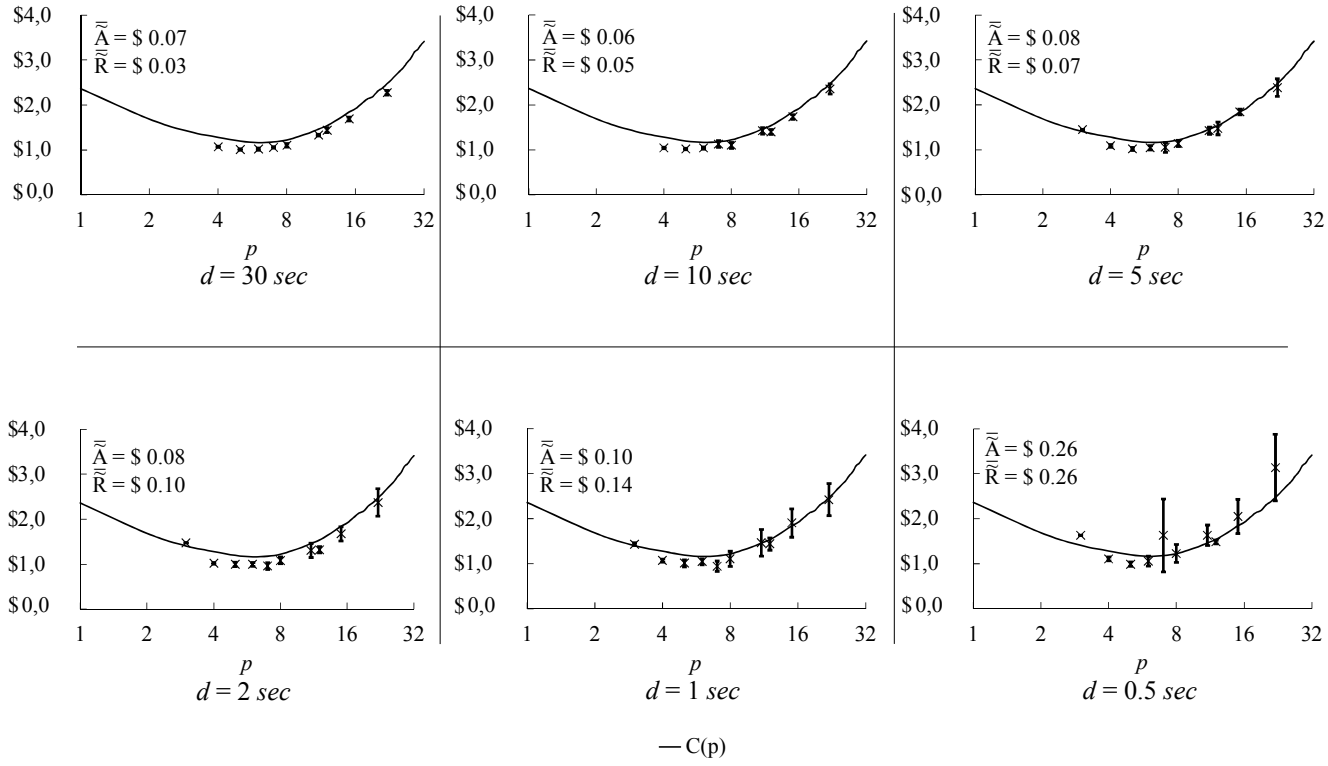

Fig. 12 Accuracy and reliability of cost approx. for different durations of probes. Values are based on 5 runs for each setting.

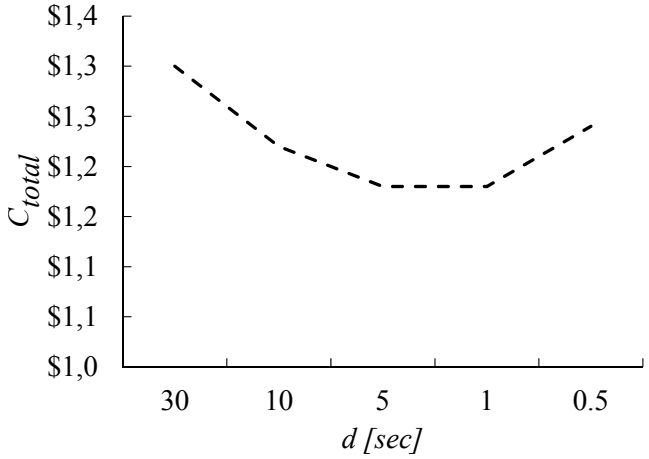

Fig. 13 Total costs for different probe durations. Values are based on 5 runs for each setting.

termine the cost overhead of our optimization method, which results from scaling operations and inaccuracies of cost approximations. As stated, for computations of *GSSSA_C* only the initial scaling phase is carried out, whereas for *GSSSA_IW* and *GSSSA_IS* a continuous alternation of both phases takes place. The parameters for the duration of a probe d and sensitivity s are set with respect to the results of Section 6.3.4, i.e., $d = 5$ sec and $s = 5\%$.

Figure 16 shows that our optimization method performs computations in a cost-efficient manner for all three example problems. The costs C_{total} of a computation with our method are \$1.18, \$0.93 and \$1.13 for *GSSSA_C*, *GSSSA_IW*, and *GSSSA_IS* respectively. This is about 1%, 24%, and 18% more compared to the most cost-efficient execution of the respective computation. Moreover, these costs are 66%, 62%, and 54% smaller compared to the most cost-inefficient execution.

6.4 Traveling Salesman Problem

6.4.1 Problem Description

The *Traveling Salesman Problem* is to find the cheapest way of successively visiting multiple cities exactly once and returning to the starting point, given a collection of cities together with the distance of travel between each pair of them. For a *symmetric* TSP, the distance of traveling between city i and city j are the same in each opposite direction, i.e., $c_{i,j} = c_{j,i}$. However, for an *asymmetric* TSP, the distance of traveling between city i and city j might be different, resulting in a more com-

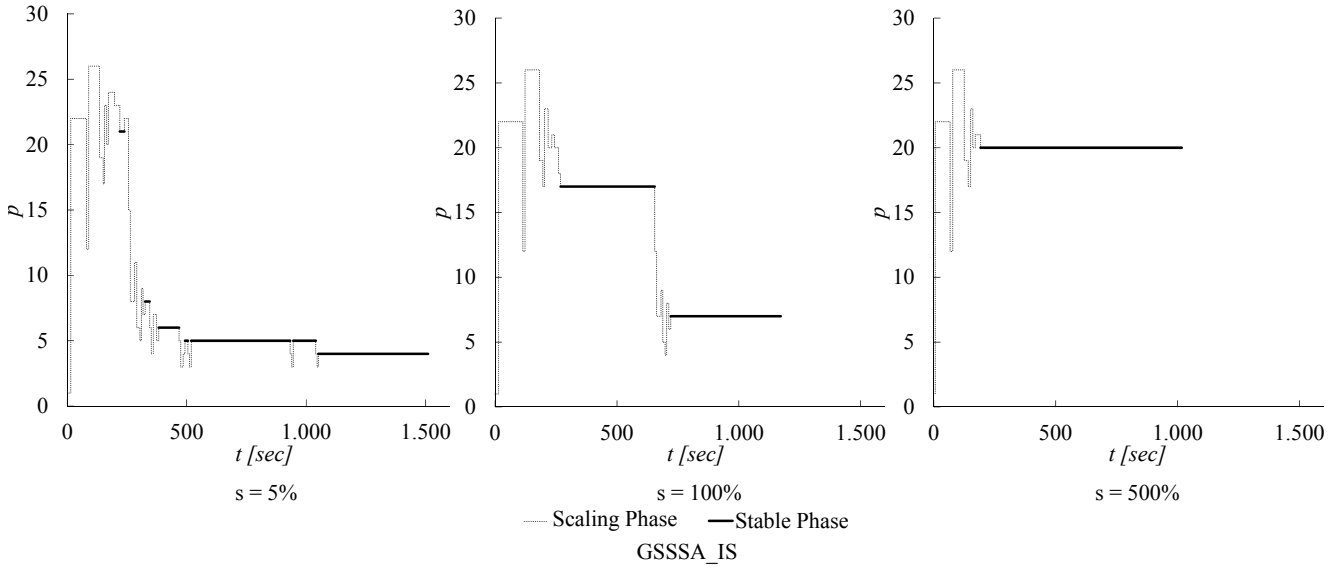

Fig. 14 Chronology of scaling operations with different sensitivity values.

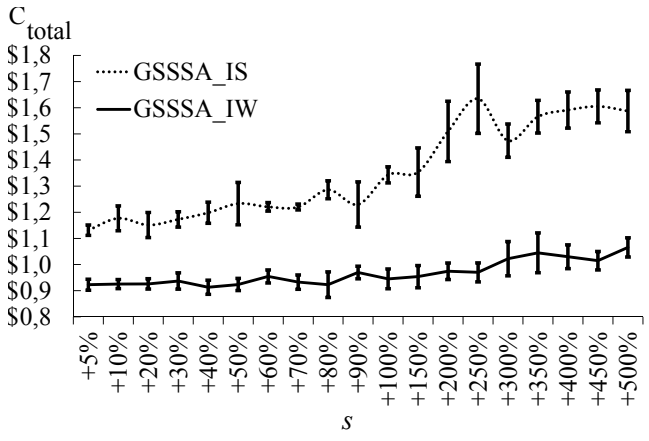

Fig. 15 Costs for computations with different trigger sensitivity. Values are based on 5 runs for each setting.

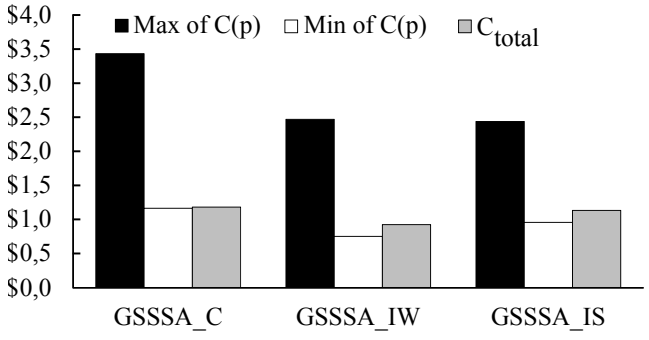

Fig. 16 Comparison of costs for computations of three GSSSA example problems.

plicated problem since the number of possible solutions doubles.

One way to model the problem is a complete graph with weighted edges. In this model cities are represented as vertices, paths between cities are edges, and a path's distance is the edge's weight. The TSP is an optimization problem, more precisely a minimization problem, with the goal of minimizing the distance of a round-trip that starts and ends at a specified city, i.e., to find a tour where the sum of individual distances along the paths is minimal.

Many algorithms for solving the TSP rely on state space trees. The root node represents the initial state, i.e., the city from which the salesman starts. The path from the root to a specific node represent a defined state of the problem, i.e., the cities that already have been visited, their order, and the distance of this path. Furthermore, all nodes located directly below the current node are the unvisited cities reachable or, more specifically, the states that are generated by visiting the selected city. Ultimately, a state is reached with no unvisited cities left, i.e., in this path all cities have been visited. This property holds true for all leaf nodes, which represent the state of a valid tour.

Employing a state space tree, the most straightforward method to solve the TSP is a brute force search that dynamically explores the complete state space tree to find the tour with minimum distance. However, the TSP belongs to the class of NP-hard problems, i.e., an increasing number of cities results in exponentially increasing computing time. Hence, even for instances of fairly small size an exhaustive search soon becomes forbiddingly large and finding the solution in reasonable time is impossible.

In practice, there are several methods for improving the computation time of a brute force search like *branch and bound*. During the computation, branch and bound is used to generate a continuously improving knowledge that is used to reduce the number of states of the state space tree. Instead of performing an exhaustive search over all possible states, branch and bound guides the search to potentially profitable regions of the tree.

Basically, the state space tree is partitioned into increasingly small subsets, called branches. Each branch, in turn, represents a subset of the feasible solutions (tours). Apart from this, for all resulting branches, the distance traveled so far is calculated, constituting the lower bound for all tours contained in this branch. This value is checked against an upper bound, which is the currently best known feasible solution. If the lower bound of the current branch is greater than the upper bound, it can be discarded since it will never lead to a better solution. Eventually, the search procedure reaches a leaf state of the state space tree, representing a single tour. This leaf state will be used as an upper bound and, provided that the tours distance is smaller than the current upper bound, a new best solution has been found. Termination takes place if there is certainty that the lower bounds of all unexplored branches are greater than the upper bound. At the same time, it is proven that the best solution found so far is also the optimal solution.

Parallelization of TSP solving becomes intuitive by leveraging the nature of the branch and bound approach: For problem decomposition, individual states of the state space tree are the input for generated tasks, which can be executed in parallel. As previously stated, this is done dynamically at runtime in order to achieve high parallel efficiency. In this context, one also has to keep in mind the bounding operations as well as the consequential pruning operations. To effectively eliminate unprofitable branches in the parallel version it is essential that all processors synchronize their local upper bounds. For this, a processor typically performs a broadcast when a new best solution has been found. However, bounding and pruning together result in highly irregular workload and furthermore in working anomalies.

In practice there exist many variations of the TSP like the *Traveling Salesman Subtour Problem* (TSSP). Generally, the TSP states that all cities of the given set have to be visited exactly once. Yet, in some problem scenarios, a time constraint may prevent the salesman to visit every city in the given period. Consequently, a tour must be determined that contains as many as possible cities but does not violate the time constraint. Solving TSSP is in most cases more complex than TSP since it requires additional evaluations. Finding an optimal solution requires not only selecting the set of cities to visit but also their optimal order. However, the discussed methods to solve the TSP can be employed by adding a time constraint that is evaluated in the same way as the upper bound.

To validate the cost-efficiency of our optimization method for real-world applications, we used three *TSP* instances of different types and different runtime characteristics as shown in Table 4. The instances *TSP_gr21* and *TSP_berlin52* are taken from the benchmark collection *TSPLIB95* [4], whereas *TSP_as30* is a randomly generated instance.

6.4.2 Analysis of Overall Cost Efficiency

In this concluding section of our experimental analysis, we evaluate the overall cost-efficiency of our optimization method for parallel computations of the three presented *TSP* instances. Like in Section 6.3.5 we compare the total costs of computations controlled by our optimization method C_{total} to the most cost-efficient and most cost-inefficient computations utilizing a constant number of processors. Also, we employ the same runtime parameters, i.e., a probe duration of $d = 5$ sec and a trigger sensitivity of $s = 5\%$.

Figure 17 illustrates the results of our analysis. As with *GSSSA*, our optimization method also results in cost-efficient computations for all three *TSP* instances. The costs C_{total} of a computation with our method are \$3.55, \$2.41 and \$5.21 for *TSP_gr21*, *TSP_berlin52*, and *TSP_as30*, respectively. This is about 13%, 30%, and 17% more compared to the most cost-efficient execution of the respective computation. Moreover, these costs are 65%, 30%, and 80% smaller compared to the most cost-inefficient execution.

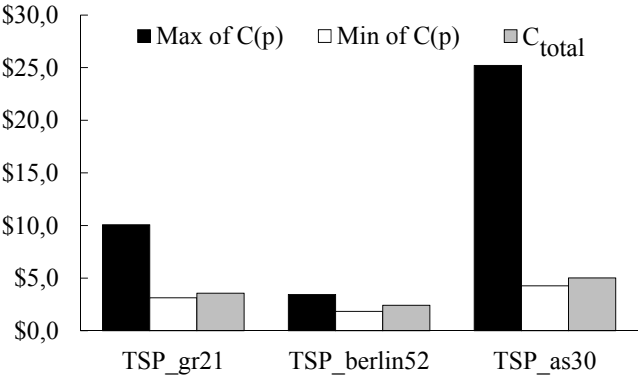

Fig. 17 Comparison of costs for computations of three TSP example problems. Values are based on 3 runs.

Name	No. of cities	*Type*
TSP_gr21	21	symmetric - No budget constraint
TSP_berlin52	52	symmetric - Budget constraint of 1200
TSP_as30	30	asymmetric - No budget constraint

Table 4 Description of three TSP problems used for experimental evaluation.

The results of our experimental evaluation of *GSSSA* and *TSP* provide strong evidence that our optimization method results in cost-efficient computations across different application domains and leads to a significant cost reduction.

7 Related Work

There exists a growing body of research on how to adapt parallel applications for cloud computing, e.g., [12], [6], and [18]. Various classes of parallel applications are well-suited for this kind of infrastructure. However, some applications can be executed only with limited efficiency on standard cloud computing offerings. This is mostly the case if nonstandard hardware (e.g., low-latency network-links) is required. Studies like [19], [37], and [28] investigate on the suitability of such applications for cloud computing. Recently, cloud computing providers have recognized the demand for high-performance cloud systems and launched offerings like the *H-Series* of Microsoft Azure, which provide performance-optimized VMs that are connected through InfiniBand [2]. Our work focuses on a class of parallel applications that are less tightly coupled, hence they do not show such specific hardware demands.

7.1 Cloud Based Distributed Task Pools

Many studies have been published on distributed task pools, e.g., [39], [43], and [41]. Specifically, the question of enabling distributed task pools for cloud computing was the topic of recent research. The authors of [50] present a modified version of the basic work-stealing algorithm, which makes decisions about victim-selection based on the latency of network links. Workers close to each other with low latencies are preferred victims in a steal-operation. While this research addresses the problem of unknown placement of VMs in the cloud infrastructure, we focus on the aspects of scalability and monetary costs.

Also the authors of [29] consider cloud-based task pools and analyzed the impact of available load information to optimize victim selection. Workers with many outstanding tasks are potentially profitable victims. Whereas this work focuses on the effectiveness of steal operations, our work investigates on cost optimization.

7.2 Modeling the Monetary Costs of Parallel Cloud Computations

Modeling the monetary costs of computations was the topic of several studies during the last decade and has considerably been gaining importance with the advent of cloud computing.

General cost models for cloud computing typically take into account the costs for various computing resources the considered cloud service is comprised of, like computing, network and storage resources [40], [49], [15].

In [22] the authors present a cost model for computations in the context of HPC installations (that are not cloud-based), taking into account the monetary costs of capital expenditures (i.e., costs of physical assets like servers) and operational expenditures (i.e., costs spent to run the system).

The traditional cost model for parallel computations defines costs as an accumulation of processing time [24]. The authors of [25] presented a cost model for cloud computing that extends the discussed cost model for parallel computations by the price charged from a cloud provider for a single processor-time unit.

To the best of our knowledge, our work is the first that proposes a cost model for parallel cloud applications that employs the concept of opportunity costs. Additionally, our cost model not only allows to determine the total monetary costs based on both runtime and utilized cloud resources; it also provides information concerning the costs of computations with unknown runtime and scalability characteristics, executed in elastic execution environments.

7.3 Optimizing Monetary Costs of Parallel Cloud Computations

In recent years there has been considerable interest in cost-effective resource provisioning for parallel com-

puting in cloud environments. Optimizing the monetary costs for executing parallel applications differs significantly from cost optimization of other application classes like web services or workflow executions w.r.t. to the optimization targets. For web services the conflicting optimization goals are monetary costs and response time [20], [47], for workflows these are monetary costs and throughput [30], [33], [32], [34]. This is in contrast to parallel computations where monetary costs and runtime/speedup are the primary optimization targets [24].

In [21], the authors deal with optimizing the execution of independent single-machine jobs, from different grid computing projects, in cloud environments. Knowing the workload of each task but not the time of upcoming submissions, the authors evaluated different online bin packing strategies used for scaling operations and scheduling to prevent idling of VMs. The examined strategies, in turn, resulted in individual trade-offs between execution time and monetary costs, obliging the user to select an appropriate strategy. This is contrary to our work, where the user only defines cost parameters, tasks have a hierarchical dependency, and the workload of each task is unknown.

Resource provisioning for parallel processing of master-slave iterative applications (e.g., Newton-Cotes numerical integration) in cloud environments has been addressed in [13], [14], and [45]. The master-slave execution environment is different from our distributed task pool execution environment, which is decentralized and employs dynamic load balancing. In their work, a central master performs resource provisioning decisions dynamically, based on the current application performance. To this end, the infrastructure's CPU-utilization is continuously monitored, and an adaptive threshold mechanism triggers scaling operations. On the other hand, we employ a cost model that utilizes the parallel efficiency to perform resource provisioning decisions dynamically. Costs, in turn, play a supplementary role in their work by using the cost-time product to consider trade-offs. By using the CPU-utilization, the authors investigated an alternative way of approaching the problem of not knowing the characteristics of a task like the workload. In their method, they assume that the CPU-utilization is a representative indicator of the performance. Unlike that, we use a probing mechanism for an approximation of performance metrics like the efficiency and communication rate. This allows us to determine the performance and parallel overhead better since a high CPU-utilization is not only induced by performance improving computations. As stated in our work, overhead like communication increase CPU-utilization while having a negative impact on performance. Moreover, the authors' application is designed from the cloud providers perspective: Increasing/decreasing the number of VMs is done in fixed steps, concerning the hosting capacity of the underlying physical server, meaning that in each scaling operation one physical server (hosting n-VMs) is added/removed. Our work, however, describes a method from the perspective of the cloud consumer and scaling is performed with a varying number of VMs, without information and mechanisms of the underlying physical hardware.

A way to achieve cost efficiency with adaptations on the level of the application itself is investigated in [44]. The discussed parallel application is based on a master-worker execution model, where a master instance is responsible for task scheduling and maintains a central task queue. The authors presented a self-tuning application that varies the sizes of tasks according to the available network bandwidth between VMs. The variation, in turn, is performed either dynamically during runtime or only once, i.e., when the application is started, depending on the application's setup. While in this work the application is adjusted as a result of unpredictable capacity changes of the infrastructure, we consider adjustments of the infrastructure, based on characteristics of the workload.

8 Conclusion

Parallel processing turned out to be absolutely essential for significantly enhancing the performance of a broad range of applications far beyond the supercomputing domain. This development stimulated sustained research activities on all aspects of parallel systems. The findings presented in our paper contribute to the understanding of the opportunities of novel, cloud-based platforms for parallel computing. Moreover, they are an illustrative example of cross-fertilization of concepts from the parallel and cloud computing domain.

We show that cloud computing offerings can be an attractive platform for implementing parallel environments that permit to optimize the monetary costs of parallel computations by the user. Specifically, our study provides a novel approach for facilitating auto-scaling to optimize the monetary costs of individual parallel cloud computations.

One focus of our future research will be to refine our method for the case of weak scalability, i.e., parallel applications that benefit from a growing problem size when the number of processors is increased. Another direction for future research may cover the extension of our optimization method to take into account dynamic pricing schemes of cloud resources, like the auction based spot market of Amazon AWS.

References

1. Amazon EC2 Pricing. URL https://aws.amazon.com/en/ec2/pricing/on-demand/
2. Availability of H-series VMs in Microsoft Azure. URL https://azure.microsoft.com/en-us/blog/availability-of-h-series-vms-in-microsoft-azure/
3. OpenStack4j. URL http://www.openstack4j.com/
4. TSPLIB95 - Library of Sample Instances for the TSP. URL https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/
5. Google Cloud Platform - Compute Engine (2014). URL https://cloud.google.com/compute/
6. Anbar, A., Narayana, V.K., El-Ghazawi, T.: Distributed Shared Memory Programming in the Cloud. In: Proceedings of the 12th International Symposium on Cluster, Cloud and Grid Computing, CCGRID, pp. 707–708. IEEE/ACM, Washington, DC, USA (2012). DOI 10.1109/CCGrid.2012.48. URL https://doi.org/10.1109/CCGrid.2012.48
7. Berenbrink, P., Friedetzky, T., Goldberg, L.A.: The Natural Work-Stealing Algorithm is Stable. In: Proceedings of the International Conference on Cluster Computing, pp. 178–187. IEEE (2001). DOI 10.1109/SFCS.2001.959892. URL https://dx.doi.org/10.1109/SFCS.2001.959892
8. Blochinger, W.: Towards Robustness in Parallel SAT Solving. In: PARCO, pp. 301–308 (2005). DOI 10.1.1.88.5737. URL https://dx.doi.org/10.1.1.88.5737
9. Blochinger, W., Dangelmayr, C., Schulz, S.: Aspect-Oriented Parallel Discrete Optimization on the Cohesion Desktop Grid Platform. In: Proceedings of the 6th Symposium on Cluster Computing and the Grid, CCGRID, pp. 49–56. IEEE (2006). DOI 10.1109/CCGRID.2006.20. URL https://dx.doi.org/10.1109/CCGRID.2006.20
10. Blochinger, W., Sinz, C., Küchlin, W.: Parallel Propositional Satisfiability Checking with Distributed Dynamic Learning. Parallel Computing **29**(7), 969–994 (2003). DOI 10.1016/S0167-8191(03)00068-1. URL https://dx.doi.org/10.1016/S0167-8191(03)00068-1
11. Bonet, B., Geffner, H.: Planning as Heuristic Search. Artificial Intelligence **129**(1-2), 5–33 (2001). DOI 10.1016/S0004-3702(01)00108-4. URL https://dx.doi.org/10.1016/S0004-3702(01)00108-4
12. Carreno, E.D., Diener, M., Cruz, E.H.M., Navaux, P.O.A.: Automatic Communication Optimization of Parallel Applications in Public Clouds. In: Proceedings of the 16th International Symposium on Cluster, Cloud and Grid Computing, CCGRID, pp. 1–10. IEEE/ACM (2016). DOI 10.1109/CCGrid.2016.59. URL https://doi.org/10.1109/ccgrid.2016.59
13. Da Rosa Righi, R., Rodrigues, V.F., Da Costa, C.A., Galante, G., De Bona, L.C.E., Ferreto, T.: AutoElastic : Automatic Resource Elasticity for High Performance Applications in the Cloud. IEEE Transactions on Cloud Computing **4**(1), 6–19 (2016). DOI 10.1109/TCC.2015.2424876. URL https://doi.org/10.1109/TCC.2015.2424876
14. Da Rosa Righi, R., Rodrigues, V.F., Rostirolla, G., Da Costa, C.A., Roloff, E., Navaux, P.O.A.: A Lightweight Plug-and-Play Elasticity Service for Self-Organizing Resource Provisioning on Parallel Applications. Future Generation Computer Systems **78**(1), 176–190 (2018). DOI 10.1016/j.future.2017.02.023. URL https://doi.org/10.1016/j.future.2017.02.023
15. Deelman, E., Singh, G., Livny, M., Berriman, B., Good, J.: The Cost of doing Science on the Cloud: The Montage Example. In: Proceedings of the Conference on Supercomputing, SC '09, p. 50. ACM/IEEE (2008). DOI 10.1109/SC.2008.5217932. URL https://dx.doi.org/10.1109/SC.2008.5217932
16. Dorigo, M., Caro, G.D., Gambardella, L.M.: Ant Algorithms for Discrete Optimization. Artificial Life **5**(2), 137–172 (1999). DOI 10.1162/106454699568728. URL https://dx.doi.org/10.1162/106454699568728
17. Edelkamp, S., Schroedl, S.: Heuristic Search: Theory and Applications. Elsevier (2012). DOI 10.1016/B978-0-12-372512-7.00019-5. URL https://dx.doi.org/10.1016/B978-0-12-372512-7.00019-5
18. Ekanayake, J., Jackson, J., Lu, W., Barga, R., Balkir, A.S.: A Scalable Communication Runtime for Clouds. In: Proceedings of the 4th International Conference on Cloud Computing, CLOUD, pp. 211–218. IEEE (2011). DOI 10.1109/CLOUD.2011.21. URL https://dx.doi.org/10.1109/CLOUD.2011.21
19. Evangelinos, C., Hill, C.N.: Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazons EC2. In: 1st Workshop on Cloud Computing and its Applications, *CCA*, vol. 2, pp. 2–34 (2008). DOI 10.1.1.296.3779
20. Fokaefs, M., Barna, C., Litoiu, M.: Economics-driven Resource Scalability on the Cloud. In: Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS, pp. 129–139. ACM Press, New York, New York, USA (2016). DOI 10.1145/2897053.2897068. URL https://dx.doi.org/10.1145/2897053.2897068
21. Genaud, S., Gossa, J.: Cost-Wait Trade-Offs in Client-Side Resource Provisioning with Elastic Clouds. In: Proceedings of the 4th International Conference on Cloud Computing, CLOUD, pp. 1–8. IEEE, Washington, DC, USA (2011). DOI 10.1109/CLOUD.2011.23. URL http://dx.doi.org/10.1109/CLOUD.2011.23
22. Gholkar, N., Mueller, F., Rountree, B.: A Power-Aware Cost Model for HPC Procurement. In: Proceedings of the 30th International Symposium on Parallel and Distributed Processing, IPDPSW, pp. 1110–1113. IEEE (2016). DOI 10.1109/IPDPSW.2016.35. URL https://dx.doi.org/10.1109/IPDPSW.2016.35
23. Grama, A., Kumar, V.: State of the Art in Parallel Search Techniques for Discrete Optimization Problems. IEEE Transactions on Knowledge and Data Engineering **11**(1), 28–35 (1999). DOI 10.1109/69.755612. URL dx.doi.org/10.1109/69.755612
24. Grama, A., Kumar, V., Karypis, G., Gupta, A.: Introduction to Parallel Computing, 2nd edn. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (2003)
25. Gupta, A., Milojicic, D.: Evaluation of HPC Applications on Cloud. In: Proceedings of the 6th Open Cirrus Summit, OCS, pp. 22–26. IEEE (2011). DOI 10.1109/OCS.2011.10. URL https://dx.doi.org/10.1109/OCS.2011.10
26. Hansen, E.A., Zilberstein, S.: LAO*: A Heuristic Search Algorithm that Finds Solutions with Loops. Artificial Intelligence **129**(1-2), 35–62 (2001). DOI 10.1016/S0004-3702(01)00106-0. URL https://dx.doi.org/10.1016/S0004-3702(01)00106-0
27. Henderson, D.R.: The Concise Encyclopedia of Economics. Liberty Fund (2007)
28. Jackson, K., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H.J., Wright, N.J.: Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. In: Pro-

ceedings of the 2nd International Conference on Cloud Computing Technology and Science, CLOUDCOM, pp. 159–168. IEEE (2010). DOI 10.1109/CloudCom.2010.69. URL https://dx.doi.org/10.1109/CloudCom.2010.69
29. Janjic, V., Hammond, K.: Using Load Information in Work-Stealing on Distributed Systems with Non-Uniform Communication Latencies. In: Proceedings of the 18th European Conference on Parallel Processing, Euro-Par, pp. 155–166. Springer (2012). DOI 10.1007/978-3-642-32820-6{_}17. URL https://dx.doi.org/10.1007/978-3-642-32820-6_17
30. Kang, D.K., Kim, S.H., Youn, C.H., Chen, M.: Cost Adaptive Workflow Scheduling in Cloud Computing. In: Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication, ICUIMC, pp. 1–8. ACM, New York, New York, USA (2014). DOI 10.1145/2557977.2558079. URL https://dx.doi.org/10.1145/2557977.2558079
31. Kiefer, J.: Sequential Minimax Search for a Maximum. Proceedings of the American Mathematical Society **4**(3), 502–506 (1953). DOI 10.2307/2032161. URL https://dx.doi.org/10.2307/2032161
32. Mao, M., Humphrey, M.: Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC, pp. 1–12. IEEE (2011). DOI 10.1145/2063384.2063449. URL https://dx.doi.org/10.1145/2063384.2063449
33. Mao, M., Humphrey, M.: Scaling and Scheduling to Maximize Application Performance Within Budget Constraints in Cloud Workflows. In: Proceedings of the 27th International Symposium on Parallel and Distributed Processing, IPDPS, pp. 67–78. IEEE (2013). DOI 10.1109/IPDPS.2013.61. URL https://dx.doi.org/10.1109/IPDPS.2013.61
34. Mao, M., Li, J., Humphrey, M.: Cloud Auto-Scaling with Deadline and Budget Constraints. In: Proceedings of the 11th International Conference on Grid Computing, GRID, pp. 41–48. IEEE/ACM (2010). DOI 10.1109/GRID.2010.5697966. URL https://dx.doi.org/10.1109/GRID.2010.5697966
35. Marques-Silva, J.P., Sakallah, K.A.: Dynamic Search-Space Pruning Techniques in Path Sensitization. In: Proceedings of the 31st Design Automation Conference, DAC, pp. 705–711. ACM, New York, USA (1994). DOI 10.1109/DAC.1994.204192. URL https://dx.doi.org/10.1145/196244.196621
36. Marques-Silva, J.P., Sakallah, K.A.: Boolean Satisifiability in Electronic Design Automation. In: Proceedings of the 37th Design Automation Conference, DAC, pp. 675–680. ACM Press, New York, USA (2000). DOI 10.1145/337292.337611. URL https://dx.doi.org/10.1145/337292.337611
37. Mauch, V., Kunze, M., Hillenbrand, M.: High Performance Cloud Computing. Future Generation Computer Systems **29**(6), 1408–1416 (2013). DOI 10.1016/j.future.2012.03.011. URL https://dx.doi.org/10.1016/j.future.2012.03.011
38. Messac, A., Puemi-Sukam, C., Melachrinoudis, E.: Aggregate Objective Functions and Pareto Frontiers: Required Relationships and Practical Implications. Optimization and Engineering **1**(2), 171–188 (2000). DOI 10.1023/A:1010035730904. URL https://doi.org/10.1023/A:1010035730904
39. Min, S., Iancu, C., Yelick, K.: Hierarchical Work Stealing on Manycore Clusters. In: Proceedings of the 5th Conference on Partitioned Global Address Space Programming Models, PGAS, pp. 1–10 (2011). DOI 10.1.1.221.7137
40. Nguyen, T.V.A., Bimonte, S., D'Orazio, L., Darmont, J.: Cost Models for View Materialization in the Cloud. In: Proceedings of the 15th Joint EDBT/ICDT Workshop, pp. 47–54. ACM, New York, New York, USA (2012). DOI 10.1145/2320765.2320788. URL https://dx.doi.org/10.1145/2320765.2320788
41. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications. In: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, *PPoPP*, vol. 36, pp. 34–43. ACM (2001). DOI 10.1145/568014.379563. URL https://dx.doi.org/10.1145/568014.379563
42. OpenStack OpenSource Community: OpenStack Project (2013). URL https://www.openstack.org/
43. Quintin, J.N., Wagner, F.: Hierarchical Work-Stealing. In: Proceedings of the 16th European Conference on Parallel Processing, Euro-Par, pp. 217–229. Springer (2010). DOI 10.1007/978-3-642-15277-1{_}21. URL https://dx.doi.org/10.1007/978-3-642-15277-1_21
44. Rajan, D., Thain, D.: Designing Self-Tuning Split-Map-Merge Applications for High Cost-Efficiency in the Cloud. IEEE Transactions on Cloud Computing **5**(2), 303–316 (2017). DOI 10.1109/TCC.2015.2415780. URL http://ieeexplore.ieee.org/document/7065311/
45. Righi, R.d.R., Rodrigues, V.F., da Costa, C.A., Kreutz, D., Heiss, H.U.: Towards Cloud-based Asynchronous Elasticity for Iterative HPC Applications. Journal of Physics: Conference Series **649**, 012006 (2015). DOI 10.1088/1742-6596/649/1/012006
46. Schulz, S., Blochinger, W.: Parallel SAT Solving on Peer-to-Peer Desktop Grids. Journal of Grid Computing **8**(3), 443–471 (2010). DOI 10.1007/s10723-010-9160-1. URL https://dx.doi.org/10.1007/s10723-010-9160-1
47. Sharma, U., Shenoy, P., Sahu, S., Shaikh, A.: A Cost-Aware Elasticity Provisioning System for the Cloud. In: Proceedings of the 31st International Conference on Distributed Computing Systems, pp. 559–570. IEEE (2011). DOI 10.1109/ICDCS.2011.59. URL https://dx.doi.org/10.1109/ICDCS.2011.59
48. Sun, Y., Wang, C.L.: Solving Irregularly Structured Problems Based on Distributed Object Model. Parallel Computing **29**(11-12), 1539–1562 (2003). DOI 10.1016/j.parco.2003.05.006. URL http://dx.doi.org/10.1016/j.parco.2003.05.006
49. Viana, V., De Oliveira, D., Mattoso, M.: Towards a Cost Model for Scheduling Scientific Workflows Activities in Cloud Environments. In: Proceedings of the World Congress on Services, SERVICES, pp. 216–219. IEEE (2011). DOI 10.1109/SERVICES.2011.52. URL https://dx.doi.org/10.1109/SERVICES.2011.52
50. Vu, T.T., Derbel, B.: Link-Heterogeneous Work Stealing. In: Proceedings of the 14th International Symposium on Cluster, Cloud, and Grid Computing, CCGrid, pp. 354–363. IEEE/ACM (2014). DOI 10.1109/CCGrid.2014.85. URL https://dx.doi.org/10.1109/CCGrid.2014.85
51. Wu, C., Buyya, R.: Cloud Data Centers and Cost Modeling - A Complete Guide To Planning, Designing and Building a Cloud Data Center, 1st edn. Elsevier, San Francisco, CA, USA (2015)