A Cloud-Based Framework for QoS-Aware Service Selection Optimization

Peter Paul Beran  
University of Vienna  
Faculty of Computer Science  
A-1010 Vienna, Austria  
peter.beran@univie.ac.at

Elisabeth Vinek  
CERN  
CH-1211 Genève, Switzerland  
elisabeth.vinek@cern.ch

Erich Schikuta  
University of Vienna  
Faculty of Computer Science  
A-1010 Vienna, Austria  
erich.schikuta@univie.ac.at

ABSTRACT
In distributed, service-oriented systems, in which several concrete service instances need to be composed in order to respond to a request, it is important to select service deployments in an optimal and efficient way. Quality of Service attributes of deployments and network links are taken into account to decide between workflows that are identical in terms of their functionality. Several heuristic approaches have been proposed to solve the resulting QoS-aware service selection problem, known to be NP-hard. In our previous work, motivated by two concrete application scenarios, we proposed a blackboard and a genetic algorithm and compared them in terms of solution quality, performance and scalability. In order to seamlessly run and evaluate further approaches and parallel versions of the current algorithms in a distributed environment, a general framework for service selection optimization has been implemented using Cloud Computing resources. A performance study on sequential and parallel blackboard and genetic algorithms for solving service selection problems has been carried out in the Cloud.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures—languages,patterns; G.1.6 [Numerical Analysis]: Optimization—global optimization; I.6.3 [Simulation and Modeling]: Applications

Keywords
Service Selection, Optimization, Genetic Algorithm, Blackboard

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. uWAS2011, 5-7 December, 2011, Ho Chi Minh City, Vietnam. Copyright 2011 ACM 978-1-4503-0784-0/11/12...$10.00.

1. INTRODUCTION
In Quality-of-Service (QoS) aware service selection problems, an abstract workflow has to be mapped into a concrete one by choosing an appropriate service instance (in the following called deployment) out of several possible ones for each service specified in the workflow. This selection procedure is mapped to an optimization process aiming at maximizing a custom utility value that takes into account QoS metrics such as availability, reliability and performance. Depending on the use case, the optimization can be single- or multi-objective, and local as well as global constraints can be imposed. The resulting challenge can be modeled as a multi-dimension multi-choice knapsack problem knowing to be NP-hard in the strong sense. Several heuristics have been proposed in various application domains to solve the stated problem.

In previous work, we have proposed two different approaches – a blackboard and a genetic algorithm – to solve QoS-aware service selection problems, and have applied them to two different scenarios. A comparative study of those approaches has shown that the blackboard reaches the optimal solution in all test cases and outperforms the genetic algorithm for a small number of deployments. Above a threshold of about 100 deployments however, the genetic algorithm outperforms the blackboard, but only reaches about 95% of the optimal value on average. In this paper, two main contributions are made. First, we extend the two approaches by exploring parallelization techniques in order to improve the runtime performance. The parallelization is carried out in a Cloud environment. Second, we present a Distributed Database and Deployment Optimization (DD-Optimization) Framework deployed on the Google App Engine [5]. This framework allows for a seamless integration of other algorithms for the same kind of service selection problems.

2. MOTIVATIONAL EXAMPLES
The motivational use cases and application areas of our approaches have been described in detail in [9]. The first application domain is a metadata application from ATLAS, a High-Energy Physics experiment at CERN. This so-called TAG system is composed of several world-wide distributed databases and deployments that have to be selected and composed as a user starts a request. The second one is a heterogeneous University information system consisting of many data sources and applications that partially share data and functionalities.
In both scenarios, when a user starts a request, there are several possibilities of combining available data sources and services. In the TAG use case the available services are for instance a web-based query interface and an engine transforming and extracting relational data into a custom file. In UCETIS, services can be join and selection operators acting across data sources. In both cases, some data source - service instance combinations can be very efficient, while others might suffer from high latencies, poor network throughput, or overloaded machines. It is important to point out that not only the concrete resources used have their limitations, but also the connections between them have to be taken into account. An optimized selection of data sources and service instances can thus considerably improve the performance over a selection based on simple rules. The two scenarios have been chosen as motivation because they present different characteristics, as shown in Figure 1. This leads to the assumption that different optimization approaches are required.

### 3. DEPLOYMENT SELECTION APPROACHES

In our previous work [9, 8], we have shown that blackboard and genetic algorithms are suitable and efficient approaches for the two described use cases. In [9] we have introduced a sequential implementation of the proposed algorithms and compared their performance and scalability by running them on local machines, using a shared data store. In the following, we present possible parallelization techniques and port all implementation versions to a Cloud environment.

#### 3.1 Genetic Algorithm

When implementing a genetic algorithm, the first step is to encode the problem with a suitable genome representation. In the concrete TAG use case, an ordered list of services corresponding to the abstract workflow is given as input. For each of these abstract services several instances (deployments) exist, and one of them has to be chosen. A solution to the problem, called a chromosome, is thus represented as an ordered list of deployment names or references.

The overall optimization process using the genetic algorithm is depicted in Figure 2. First, a random population of concrete workflows is initialized. Second, the fitness of each individual is evaluated, based the QoS attributes of deployments and links. Third, good individuals are selected for replication. A classical roulette wheel selection [6] has been implemented. Finally, crossover and mutation operators are applied. Crossover splits two parent workflows and combines the parts; mutation is implemented as random exchange of one deployment.

Nowostawski and Poli [7] proposed a taxonomy of parallel genetic algorithms, identifying three main motivations. First, some problems require a very large population that is highly memory-consuming and thus makes it impossible to efficiently run a sequential genetic algorithm on a single machine. Second, fitness evaluation can be very time-consuming. As the fitness has to be computed once for each individual, it is a natural candidate for parallel processing. Third, sequential genetic algorithms can get trapped in certain regions of the search space, and thus in local optima. Parallel algorithms however can search different regions in parallel and thus reduce the risk to return local optima. In our use case, the population does not need to be very large and the way it is encoded, it is nearly impossible that it affects memory performance. The other two motivations however are good reasons to explore the usage of parallel genetic algorithms. There are several models of parallel genetic algorithms, as summarized in [1]. Taking into account the concrete use case and the sequential implementation, two parallelization techniques are most promising: (1) **distributed fitness evaluation** and (2) the **island model**, where a single population is partitioned into several subpopulations exchanging results [1]. As a first proof-of-concept, the distributed fitness evaluation has been implemented and deployed inside the Google App Engine, as described in Section 4, where each chromosome of a generation is evaluated in parallel.

#### 3.2 Blackboard Algorithm

Blackboards were originally introduced by Corkill [2], they are tailored to complex problems characterized by incomplete knowledge and uncertainties regarding the attributes and the behavior of the involved components. Blackboards consist of a global blackboard, a knowledge base and a control component. A **global blackboard** represents a shared information space containing input data and partial solutions. In our DD-Optimization framework setup, the information consists of a list of deployments stored in a data store inside the Google App Engine. A **knowledge base** is composed of several independent regions, each of which is owned by a single expert disposing of specific knowledge. The global blackboard acts as "mediator", allowing the dif-
frent regions to communicate. In our implementation, the regions are the deployments that have to be selected for each abstract service in the input workflow. The control component defines the course of activities (phases) required to perform the optimization problem, as shown in Figure 3. A cost-based decision tree is generated based on cost estimations for the visited paths. The expansion of promising deployments for a step in the workflow is handled by an OpenList and BlockedList. The OpenList contains a list of possible steps (deployments to choose). Each of these steps is rated by applying a cost function that sums up the costs of past decisions and the cost of the next step. Considering this total cost value the cheapest step (deployment) is chosen for the next step in the optimization approach. The BlockedList contains steps (deployments) that do not fulfil the given requirements and therefore must be excluded from the set of possible solutions.

There are two possibilities for parallelizing the presented blackboard algorithm: (1) intra-parallelism with a parallel execution of the expansion phases, inside one code block, and (2) inter-parallelism with a parallel execution of the blackboard stages, followed by a block that links the results of the stages together. In our DD-Optimization framework intra-parallelism has been implemented, where the expansion of a node is performed in parallel.

4. DEPLOYMENT SELECTION IN THE CLOUD

Cloud Computing complements the abilities and features of Grids by allowing casual users to dynamically use services on a very simple and pay-per-use basis. Therefore we decided to build our optimization framework on a Cloud environment that allows us to use computational, data, network and API resources up to a certain amount in the context of dynamic deployment selection and its optimization. In literature a Cloud environment is typically categorized as an Infrastructure (IaaS), Platform (PaaS) or Software (SaaS) environment [4, 3]. Each segment usually fulfills a specific purpose and allows a user to use resources on a rather low and hardware-related level (IaaS) or on a high- and rather Software-related level (SaaS). For our approach PaaS is appropriate as it provides the required infrastructure on a high-level base and simultaneously eases the way of developing and deploying the code.

As our initial prototype is implemented in Python we were aiming to find an execution environment that natively supports Python and allows for an easy installation, implementation and deployment of our algorithms.

4.1 Decision for the Google App Engine

Out of the existing popular PaaS providers, we chose to deploy our application on the Google App Engine (GAE), because of several reasons. First, the GAE offers a daily free amount of 1 GB storage, over 100 Million API calls and 417 Million queries, and approximately 2.5 CPU-hours [5]. Second, GAE implements a PaaS, which is the appropriate service level for our application prototyping. Third, the GAE has recently been used successfully for various applications, ranging from research prototypes to business applications. Additionally, it provides a well-documented API that allows quick development and easy deployment. Finally, our initial prototype has been implemented in Python. As the GAE natively supports Python applications, there were only small changes required to run the algorithms inside the GAE.

4.2 DD-Optimization Framework in the GAE

With the DD-Optimization framework we envision to provide an easy to use simulation and execution environment for sequential and parallel algorithms. It allows for the implementation, deployment and benchmarking of arbitrary algorithms that solve the described service selection challenge for randomly generated test data, that can be of a various problem size ranging from only some rather “static” service deployments to a very “dynamic” scenario, where deployments often change their characteristics and do not rely on the same state for a long period.

The data model used is depicted in Figure 4. A Component can be of any of the following types: service, deployment or link, representing the abstract functionality, concrete service instance and network link between two instances respectively. Attributes are defined for deployments and links in the Attribute table. Finally, all simulation results are stored in a Benchmark table, with all required metadata. Based on this simple data model, the DD-Optimization Framework has been implemented using four GAE APIs: (1) the Datastore API persistently storing data, (2) the Memcache...
API as an additional facility to store temporary data for repeated queries, (3) the Task Queue API to asynchronously run background processes by inserting so-called tasks into a first-come, first-served queue, and (4) the URL Fetch API for performing HTTP and HTTPS requests within the program code. It has to be pointed out that the GAE has some limitations affecting the optimization. In particular, time restrictions for requests and URL fetches are tight. Therefore, applications requiring long requests will not scale in the GAE.

5. SIMULATION AND EVALUATION

The blackboard, genetic algorithm and random walk (as baseline) have been deployed in both sequential and parallel versions inside GAE and evaluation runs have been carried out. The random walk implements the service selection challenge by considering each possible combination of deployments that fulfill the requirements of the workflow, a chain of 4 abstract services. The datasets for our simulation runs follow the data model described in Section 4. The following system setup datasets, all with 4 abstract services, have been generated: 8 deployments (test set 1), 16 deployments (test set 2), 32 deployments (test set 3) and 40 deployments (test set 4). For each deployment (or link) a performance index attribute value is randomly generated containing a float value between 1 and 10. For the evaluation presented here, an even deployment distribution among services is assumed, e.g. two deployments for each of the four abstract services considering test set 1 with 8 deployments. As in [9], further tests can be made with different deployment distributions, but results are expected to be conforming to those obtained in [9]. The results obtained from the test runs are shown in Figure 5.

![Figure 5: Evaluation Results](image)

As expected, for a small number of deployments, the sequential version outperforms the parallel ones, because the URL fetch method introduces some handling overhead that does not pay off when running the optimization on small problem spaces. On test set 3 (32 deployments), the parallel genetic algorithm outperforms the sequential one. We believe that the fact that the situation is again reversed for test set 4 is an artefact and more runs need to be done to get better average values. Regarding the fitness/quality of the reached solutions, the results of the random walk has been taken as benchmark. It has been verified that the solutions of the blackboard and the genetic algorithm are within a range of 92% of the optimal solution. The benchmark results give indications on which algorithm to use in which use case scenario, especially as regarding the size of the problem space. Future test sets will be generated using incomplete data, more QoS values and more complex workflow patterns, in order to study the specific algorithms behavior.

6. CONCLUSION

In this paper, we discussed and implemented parallel versions of a genetic algorithm and a blackboard to solve QoS-aware service selection problems. In order to compare them, and possibly others, a Cloud-based framework for hosting, executing and evaluating those algorithms has been introduced. The concrete implementation and deployment have been carried out using the Google App Engine that allowed a quick code prototyping and deployment in the Cloud. Evaluation results verified results of our previous work and showed that GAE is a suitable environment for testing and comparing the proposed algorithms, both sequential and parallel. However, for larger problem setups, the GAE reaches its limits in terms of parallel request handling. In future work, further parallelization techniques will be explored for the proposed algorithms. Finally, we plan to include more algorithms into the Cloud-based framework, in order to get extensive benchmarking on algorithms for QoS-aware service selection problems.

7. REFERENCES