Scalable algorithm for the service selection problem

Yanik Ngoko, Christophe Cérin
Université de Paris 13, LIPN
99 Avenue Jean Baptiste Clément, 93430 Villetaneuse
{yanik.ngoko, christophe.cerin}@lipn.univ-paris13.fr

Alfredo Goldman
DCC-IME-USP
Rua do Matão 1010
gold@ime.usp.br

Abstract
In this paper, we are interested in fast algorithms for the service selection problem. Given an abstract services' composition, the objective in this problem is to choose the best services for implementing the composition such as to minimize a given penalty function.
Our work contributes to both the sequential and parallel resolution of this problem. For the sequential resolution, we show how to extend a prior algorithm for QoS prediction to obtain a fast sequential resolution of the service selection problem. Our proposal innovates in the optimization techniques (variable ordering, branch and bound, etc.) used for the runtime minimization. For the parallel resolution, we discuss on two possible formulations for the parallelism: task and data parallelism. We show that on our problem, the latter formulation is adequate because it leads to a more scalable resolution. Finally, we conduct various experiments that show that super-linear speedups can be reached with our new parallel algorithm.

Keywords: Service Selection; QoS Prediction; Graph Reduction; Domain Decomposition; Work Stealing; Backtracking.

1. Introduction

With the emergence of clouds and service-oriented systems, middleware has become one of the most active tools in modern distributed infrastructures. In these infrastructures, various design concepts and technologies are used for supporting several levels of parallelism: from low level ones achieved by GPU units and cores to supercomputers and large data centers in which we can obtain massive parallelism. If in most cases, middlewares are designed for helping external applications to benefit from this computing power, for taking decisions, they do not always efficiently use the large potential of their underlying infrastructures.

We are convinced that there is a gap between the exploitation of parallelism in middlewares and the potential of the computational power on which they are deployed. Our conviction is supported by the practical utilization of some middlewares tools (e.g OAR (OAR2 2012) in Grid’5000, Slurm (Slurm, 2011) on clusters and an analysis of the common approaches and philosophies adopted when targeting middleware problems. In more detail, we conducted a study on the conference papers (research and industry tracks) of the IEEE Service Computing Conference (SCC) held in Anchorage in 2014. On over the 105 papers we analyzed, at most 10% between them focused slightly or deeply on scalability or parallelism for solving the service computing problem they addressed. This would not have been a concern if none of these papers dealt with a computing-intensive problem. However, we noticed that 29% of these papers targeted the resolution of a NP-hard, non linear or exponential problems like the service selection problem, virtual machines allocation, graph partitioning. Moreover only 3% of the papers in which the focus was to solve an exponential or NP-hard problem proposed to develop a parallel or scalable solution. We did a similar study on the proceedings of the 2014 IEEE cloud conference held in Anchorage. In research sessions, only 4% of the 25% of the papers where an NP-hard or exponential problem (e.g virtual machine consolidation, virtual machines placements, resources partitioning etc.) was the target focused on proposing a scalable or parallel solution. At least two reasons might justify this little interest in parallelization. The first is the usage of the client-server model. In distributed computing, it seems natural to consider parallelism in a context where a set of users
requests are each treated by a different instantiation (of a web service, function, objects), run on a part of the entire machine. In this view, one can parallelize the treatment of users requests in creating several instantiations of a middleware service or function that each runs a sequential algorithm. This approach for parallelization has a main interest: the ease of its implementation. For instance, the clouds today incorporate automatic load balancing tools for supporting this request-based parallelism on any cloud-service. The limit of this solution is that requests might address computing-intensive problems like the service selection problem. In these cases, a sequential resolution might be inefficient for optimally solving such a problem in a real time setting. The second reason for the little interest in parallelism is the use of heuristic optimization techniques. Indeed, many papers that we analyzed dealt with optimization techniques like greedy algorithms, genetic algorithms and approximation algorithms. If these techniques can help to improve the response time, let us notice however that they can deteriorate the quality of the results expected by the users. For stating this clearer, let us for instance consider the service selection problem studied in this paper. Being given a set of abstract services that collaborate in a business process, the objective is to choose the best concrete services for its implementation such as to optimize the total response time and the energy consumed in the collaboration. In applying an optimization technique for quickly solving the problem, we could obtain a choice of concrete services that is too far from the optimal response time and energy consumption we could have expected in the service collaboration.

The conviction that supports our work is that for improving the quality of service in the servicing of middleware requests, one must consider for each request a balanced view in which heuristic optimization and parallelism are both considered. While optimization is used for simplifying the problem addressed by the request, parallelism serves to quickly find optimal solutions under the optimization assumptions. Typically, given an NP-hard problem, our idea is to apply a robust optimization approach for deriving a sequential algorithm that will then be parallelized.

This paper underpins this balanced usage of parallelism and optimization in proposing a scalable algorithm for the service selection problem. Parallel algorithms for service selection have been investigated in previous work (Hening & Balke, 2010; Pathak et al., 2006; Bartolos & Bielikova, 2009). We differ from them on two main points. Firstly, we use a more classical representation for services' compositions and parallelize a sequential algorithm obtained by performing different optimizations like the variable ordering or backtracking. Secondly, we analyze various options for the parallelization of service selection and prune a data parallelism that leads to more scalability. In more details, our parallelization is achieved with two techniques: the domain decomposition (in particular the variable partitioning approach (Platzner et al., 1996) and work stealing (Blumofe & Leiserson, 1999). Finally, we conduct a large set of experiments on representative benchmarks that demonstrate that super-linear speedups can be achieved by our approach.

The remainder of the paper is organized as follows: in the next section, we discuss the related work. In Section 3, we give a formal description of the variant of the service selection problem studied in this paper. Section 4 presents the sequential optimization we performed for reducing the runtime. In Section 5, we discuss about the parallelization of our sequential algorithms. Section 6 is devoted to experimental results and we conclude in Section 7.

2. Related work

As stated in the introduction, there is a large literature around the service selection problem.

Differently to our work, most of these contributions implicitly target sequential execution contexts. In distributed contexts, we refer to the work that have be done in (Li Fei et al., 2006; Xin Li et al., 2013; Alrifai et al., 2012 ). Fei Li et al. showed how we can limit the occurrence of bottlenecks in the exchange of registry information required for the composition of services. The idea of a distributed composition of services in geographically distributed clouds is developed in the work of Xin Li et al. The work of Alrifai et al. proposes a distributed algorithm whose idea is to decompose the global SLAs constraints into local ones. Doing so, they show that we can quickly obtain good approximations for the service selection problem. Our work differs from these contributions on two points. Firstly, they focus on building services' compositions in a distributed context (not necessarily parallel) while we are interested in parallelizing the composition process. Secondly, we are interested in finding optimal solutions and not approximated ones as Alrifai et al. The parallelization of the service selection problem was also investigated. Beran et al. (Beran et al., 2012) described two parallel algorithms for the service selection problem. The first one is a master-slave parallelization of a genetic algorithm. Their second proposition is the parallelization of an A* like algorithm for service selection. Though interesting, their proposal leads to near-optimal solutions. Hening and Balke (Hening & Balke, 2010) proposed a parallel framework for service composition. The parallelization is achieved in partitioning the services graph. We differ from their solution on three
points. Firstly they used a particular representation of the services' composition (the binary tree based web composition system) while we use a more classical one. Secondly, we consider a different sequential algorithm to parallelize. As it will be discussed further, our sequential algorithm includes several optimizations that can drastically reduce the search space of the service selection problem. Finally, we both use graph partitioning for creating parallelism. But, our solution is related to the domain decomposition technique. The difference in the way we formulate parallelism is practically observed by the fact that while our parallelization leads in some cases to a super-linear speedup (with 2,4,8 or 16 threads), the one proposed by Hening and Balke is only near-linear. In (Pathak et al., 2006) and (Bartolos & Bielikova, 2009), two parallel algorithms for service compositions are proposed. As criticized in (Hening & Balke, 2010), these solutions only exploit the parallelism that is inherent to the interactions among operations of the services' composition graph; this can result to a poor scalability. In our solution, it is the set of services that can be associated with abstract ones that restrict the parallelism. Since the practical hardness of the service selection problem somehow depends on the size of this set, we have a more scalable algorithm. 

The work proposed in this paper is built upon our prior work in (Ngoko et al, 2014a) on the sequential resolution of the service selection problem and the parallelization we proposed in (Ngoko et al, 2014c). In this paper, we go further in the sequential resolution in introducing two data structures. The first data structure helps to optimize repetitive calls in the iterative process at the core of the sequential resolution. The second is used to generalize the sequential process such as to make it more flexible to runtime optimization. In comparison to the parallel contributions we did before, we discuss in this paper about an alternative formulation of the parallelism based on the map-reduce paradigm (Dean & Ghemawat, 2008). We show the limits of this solution, explain the interest of our alternative approach and discuss about its scalability.

3. The Service selection problem

The problem formulation we use was introduced in previous work (Yu & Lin, 2004; Ngoko et al., 2013). As input, we have a services' composition described as a hierarchical services graph (HSG). A HSG is obtained in composing three graphs: an operations' graph that describes a set of business process interactions between operations, a services' graph stating the services that implement each operation and a machines' graph that defines the machines on which each service is run. In Figure 1, a representation of such a graph is proposed.

In our HSG, we admit that the operations (in the operations' graph) are defined by the set $O$. Given an operation $u$, the set of its possible services implementations is referred to as $Co(u)$. Each implementation is characterized by a service response time ($SRT$) and energy consumption ($EC$). The objective is to choose the best implementations of operations for minimizing a penalty function while fulfilling a set of SLAs constraints. More formally, we describe the problem as follows.

**Problem inputs:**

A set of operations $O$. For each operation $u$, an implementation set $Co(u) = \{u_1, ..., u_v\}$. For each $u_v$, we have the mean response time $S(u_v)$ and the energy consumption $E(u_v)$. We assume two upper bounds that are issued from SLAs constraints: the bound $MaxS$ for the service response time and $MaxE$ for energy consumption. Finally, we have a tuning parameter $\lambda \in [0,1]$.

**Problem objectives:**

We are looking for an assignment of service implementations to $O$ that fulfills the following constraints:

1. each operation must be associated with a unique implementation;
2. the QoS of the resulting composition must not exceed $MaxS$ in response time and $MaxE$ in energy consumption (SLAs constraints);
3. if $S$ is the service response time and $E$ the energy consumption, then the assignment must minimize the global penalty $\lambda S + (1 - \lambda) E$.

Here, $\lambda$ is provided by the user; it serves for prioritizing the SRT or the EC in the optimization. One drawback in the constraint 3 is that it does not seem natural to add the service response time to the energy consumption since they are expressed in distinct units. Alternatively, we can adopt a normalized version in which the goal is to minimize:

$$\lambda S + (1 - \lambda) \frac{E}{S + E}$$

In this paper, we will keep the first formulation of the penalty. Let us observe that the solution we propose in this paper can easily be adapted to the normalized case.
In the problem formulation, we assumed that the SRT and EC are real values. This choice is questionable because the operations deal with different types of inputs. Fortunately, the resolution approaches that we will propose can be extended to more complex formulations as the probabilistic modeling studied in (Ngoko et al., 2014b).

For completing the problem definition, the structure of the operations graph must be defined. We must also define how to infer the QoS (SRT and EC) of a services’ composition once implementations are associated with operations. On these points, we will use the considerations made in our prior works (Ngoko et al., 2013). In particular, the structure of the operations’ graph will be presented in the next sections.

4. Sequential algorithm for service selection

The service selection problem can be solved by an iterative process whose idea is to evaluate all possible assignments of operations to implementations. In this process each iteration includes the following tasks:

1. definition of an assignment of possible implementations to operations;
2. Estimation of the resulting SRT and EC and check of the non-violation of SLAs constraints.
3. If a constraint is not violated, estimation of the penalty of the assignment and comparison with the best known solution.

The possible assignments of implementations are defined by the HSG in which the operations’ graph is contained. For instance, according to Figure 1, A is implemented in w1 and w2. E is implemented in w2 and w3 and C is only implemented in w1.

In the iterative process we defined, the service selection problem is based on two algorithms: (1) A QoS predictor that given an assignment computes the SRT and EC that it leads to; (2) an iterative algorithm that explores assignments and call the QoS predictor for checking the non-violation of constraints and, estimating their objective value. We will refer to this process as the exhaustive search process.

The exhaustive search is a natural option for finding optimal solutions to the service selection problem. But, there is a non negligible set of useless explorations performed in this search that we can avoid. In this work, we will propose an alternative resolution of the problem. From the exhaustive search process, we will retain two important requirements for solving the service selection problem: the need of a mechanism for QoS prediction and the need of a search process within possible assignments. Below, we present the mechanism for QoS prediction that we will use.

4.1 QoS prediction: a graph reduction approach.

Given an assignment of services to operations of a HSG, the graph reduction approach defines a method for computing the SRT and the EC. The precondition for the application of this method is that the operations’ graph of the HSG must be recursively decomposable in known patterns. This means that there is a set of subgraphs structures that we can use for composing this graph. In our case, we assumed that our services’ composition automates a business process. Therefore, we can define the subgraphs structures from known patterns used in business process modeling. In this work, we restricted the set of patterns that we will use to the ones of Figure 2.

Fig 2: Subgraph patterns for the operation graph

Here, each Pi is either a subgraph based on the same patterns, or an operation. When for a pattern all the Pi are operations, we say that we have an elementary subgraph.

Let us now admit that the operations’ graph is decomposable in these patterns. The graph reduction approach occurs through several stages where an elementary subgraph is reduced into a single node whose SRT and EC are the ones of the reduced subgraph. An illustration of the execution is provided in Figure 3. A key challenge in this process is the search of elementary subgraphs to reduce. In our prior work (Goldman & Ngoko, 2012), we introduced the notion of reduction order. Such an order defines the successive set of elementary subgraphs that we must consider in the reduction. More practically, a reduction order is a stack whose elements have the key (root, leaf). In this notation, root is the root node of the elementary subgraph and leaf is the node of maximal depth of the graph.

Considering for instance the Figure 3, (g3, g4) refers to the elementary subgraph that comprises the operations C, D. After the reduction of (g3, g4), (B, g3) will be an
elementary sequence with the node B and the reduced node representing \((g_3, g_4)\). After the reduction of \((B, g_3)\), \((g_1, g_2)\) will also be an elementary subgraph.

The reduction order states how to recursively reduce the operations’ graph to a unique node. Let us notice this can lead to the computation of QoS if in the reductions of the subgraphs, we aggregate the operations’ QoS (SRT and EC). In Ngoko et al. 2013, we defined the aggregation rules we need for this purpose.

**4.2 Exploring services implementations with graph reduction.**

Let us assume an assignment of implementations to the graph of Figure 3. For the prediction of the resulting QoS, the exhaustive search starts by reducing the subgraph \((g_3, g_4)\). At this stage, we will have the SRT and EC of \((g_3, g_4)\); from a simple comparison, it can happen that the assignment made on this subgraph already violates a SLAs constraint. However, in the exhaustive search, this information will not be exploited. One will reduce totally the graph and check only at the end whether or not a SLA is violated. The process will be repeated until we explore all possible assignments.

The exhaustive search has an obvious drawback: it does not exploit the local information produced by the graph reduction for accelerating the detection of SLAs violations. The runtime overhead that this leads to can be important. In particular, let us assume that we have \(n\) operations in the graph and \(d\) implementations for each operations. In the case, where we can detect a SLAs violation from the reduction of two operations, we will only explore \(2^d\) assignments on two operations. With the exhaustive search, we will explore \(n^d\) assignments on \(n\) operations. Consequently, the exhaustive search process can be particularly inefficient in the case where we do not have any solution to the service selection problem or in the cases where there are few solutions.

For improving the exhaustive search process, we proposed in our prior work (Ngoko et al., 2014a) to explore solutions based on Nested List of operations’ Ordering (NeLO). We discuss this data structure below.

**4.2.1 Using Nested Lists for Ordering in the search of solutions**

NeLO are built for two purposes: the first one is to define an ordering in the generation of assignments that we explore in the resolution. The second purpose is to define a set of evaluations on SLAs constraints that will serve to quickly detect violation of constraints. For understanding the key point of this data structure, we briefly recall below what we name assignment. Given the graph of Figure 3, let us assume that A can be associated with the implementations set \(\text{Co}(A) = \{A_1, A_2\}\). Let us also assume that B can be associated with the implementations \(\text{Co}(B) = \{B_1, B_2, B_3\}\); then we define a possible assignment as \([(A,A_1), (B,B_3)]\).

Depending on the number of nodes of the operations’ graph, an assignment can be complete or partial. In the former case, all operations are associated with an implementation while in the latter, it is the case only for some operations.

Given the graph of Figure 3, \([(A,A_1), (B,B_3)]\) is a partial assignment; a complete assignment must define an implementation for A, B, C, D, E, F.

With the NeLOs, the idea is to solve the service selection problem in using partial assignments that are progressively completed if no constraints violation is found. This means for instance that in Figure 3, we start by defining a partial implementation for D and C. Once done, we can evaluate the subgraph \((g_3, g_4)\) on SRT and EC. If no SLA violation is found, we continue the completion of this partial assignment by choosing an implementation to B. The process (completion of assignment and SLAs evaluation) is repeated until we find a complete assignment or a constraint violation. As data structure, a NeLO is based on a stack that defines the ordering in which assignments will be made to implementations. Some entries of the stack can point towards a list in which are specified the reductions to perform once assignments are made to the referring sub-stack. For the graph of Figure 3, an example of NeLO is given in Figure 4.
an implementation. After, an implementation must be defined for C. Once done, one must reduce the graph \((g_3,g_4)\) and then must check the SRT and EC that this will result to. If we do not have a constraint violation, we continue in defining an assignment to B; the graph \((g_1, g_2)\) will then be elementary and reduced. Then, one checks the SRT and EC that their reduction will lead to. One continues in the same way until reducing the entire graph. This will be done after defining an implementation to A.

There are several challenges in the design of a NeLO. For instance, how must we order the operations in the completion of assignments and what are the constraints evaluation to perform on sub-stacks? For more details about their design, we invite the reader to take a look at the work we did in (Ngoko et al., 2014a). The main information we can retain is that polynomial time algorithms exist for the design of NeLOs in which operations are ordered such as to maximize the number of local checks of SLAs violations.

We introduced NeLOs in criticizing a worst case situation in the exhaustive search: when the problem does not have any solution or when there exist few solutions. It is obvious to notice that this situation is optimized in using NeLO. Indeed, in the graph of Figure 3, let us consider that a SLAs violation exists on the subgraph \((g_3, g_4)\). Following the NeLO we will evaluate this subgraph after assigning an implementation to D and C. We will then avoid to explore a huge set of assignments that in exhaustive search would have been considered.

Regarding search methodologies, the exploration of assignments based on NeLOs can be referred to as a backtracking exploration with a variable ordering strategy. For instance, given an assignment if a constraint violation is detected in the reduction of \((B, g_4)\), one can backtrack and change the assignment made to B such as to consider \([D, D_1], (C, C_2), (B, B_2)]\).

If again some constraints are violated, one can backtrack for changing the implementation associated with C. In addition to backtracking with variable ordering, we propose to add a bound control in the exploration of potential solutions. The idea is to evaluate the objective value (the function \(\lambda S+(1-\lambda)E\)) of any partial assignment such as to compare it with the one of the best known solution. This means that from the assignments made to D, C and B, we propose to compute the local objective value in SRT and EC that the partial assignment leads to. If this local objective function exceeds the one of the best known solution, then there is no interest in completing this partial assignment. For more details about bound control, we refer the reader to the work we did in (Ngoko et al. 2014a).

At this point, we showed how NeLO can improve the search of the optimal solution for the service selection problem. In the next sections, we will discuss about other aspects of the search optimization in using NeLOs.

### 4.2.2 Making successive reductions

For finding the optimal solution of the service selection problem, we must define various assignments of the operations' graph. For the evaluation of partial assignments, the NeLOs suggest to make reductions that will destroy this graph. This implies that for the evaluation of a complete assignment, we must destroy the operations' graph. How then should we proceed since we need again to have this graph for evaluating other assignments? This is the main challenge that will be discussed in this part.

Our first answer to this challenge is to use local copies. Before starting any assignment, we make a local copy of the operations’ graph. Then, one progressively assigns implementations to its operations while making the reductions suggested by the NeLO. Once we end with this assignment, we delete the remaining nodes and create a new copy for defining the next assignment. In appearance, this approach is simple; however, let us notice that things can be more complex when we must backtrack. Indeed, let us assume that after making an assignment to D, C, B, E one wants to backtrack for changing the implementation of D.

According to the NeLO of Figure 4, the node D will already be deleted in the reduction of \((g_3, g_4)\). This means that for changing even a part of a partial assignment, we must create again local copies of the operations’ graph.

The idea of local copies is then inappropriate in backtracking because it can result in a huge set of local copies to create. For quantifying this, let us consider that the operations graph is a sequence of \(n\) operations. Then we must pay an overhead of \(O(2n-1)\) for the creation of each local copy (we have \(n-1\) edges). Assuming that each node is an operation with \(d\) possible implementations. Then, in the best case, we have at least \(n^d\) assignments to evaluate (we only counted complete ones). In this case, the overhead in copies will be in \(O(n(2n-1))\). Here, we did not include the cost of deleting nodes and edges of elementary subgraphs in the reductions. In conclusion, the idea of keeping a local copy of the graph can lead to an exponential time cost: how can we improve it?

Our alternative to local copies is based on the left naming assumption defined below.

**Left Naming assumption:**

In NeLOs, once a subgraph \((root, leaf)\) is reduced, it will be further referred to only with its left name: \(root\).

This is the case in the NeLO of Figure 4. Indeed, in referring to the reduction of \((B, g_3)\), we consider here that \(g_3\) is obtained from the reduction of \((g_3,g_4)\). We have the same consideration when referring to the reduction of \((g_1, F): g_1\) refers to the reduction of \((g_1, g_2)\).

It is trivial to generate a NeLO in which the left naming assumption is respected. The interest is that we do not need
again to destroy the operations’ graph in the reduction. More precisely, we propose to use a QoS array that works as follows.

For each node of the operations’ graph, there is an entry in the QoS array that points towards a SRT and EC value. We interpret the reduction of an elementary subgraph as: the computation of the SRT and EC that it leads to followed by the update of these results in the QoS entry of the root node of the reduced graph. This interpretation exploits the fact that we can latter refer to a reduced graph as the root node of the subgraph it denotes. Summarizing, instead of making effective reductions, we proceed by updating the SRT and EC of the root nodes of the elementary subgraphs considered.

There are some critical points to address in the formulation of this solution. For instance, we might have a situation where the NeLO mentioned to reduce a graph that is not practically elementary since we did not destroyed the nodes. This will be the case in Figure 3 when we will try to reduce (B, g3) or (g1, g2). This situation is easy to address. This is because before the reduction of (B, g3), we will already have the necessary updates made in the QoS entries of g3. It suffices to use them and update the QoS entry of B accordingly. The same tricky situation will occur in the reduction of (g1, g2). Again, at this point, all the internal entries of the root nodes of the subgraphs will be updated.

Summarizing, our solution for improving the runtime in backtracking consists of virtual reductions that are performed on an internal associative array of QoS and to create a NeLO based on left naming. The entries of the associative array consist of tuples (node name, SRT, EC). Initially, the SRT and EC of these tuples are only defined for operations. For making a virtual reduction on the subgraph (x, y), we update the entry (x, srt(x), ec(x)) of the associative array. The values of this new entry are then used each time we refer to the subgraph (x, y). In this solution, we have neither the overhead induced by the copies of operations’ graph nor the overhead caused by the destruction of subgraphs. In comparison, we must just keep the associative array update.

We showed how to efficiently operate several reductions using NeLOs. Until now, we considered that NeLOs are built for maximizing the number of local checks for SLAs violations. Below, we discuss of alternative views.

**4.2.3 Generalizing NeLOs for handling best cases in lazy reduction**

We introduced NeLO as a data structure that serves to optimize the constraints check in partial assignments. NeLOs are useful over the worst cases in exhaustive search but what about their efficiency over the best cases?

Our answer to this question is that a NeLO in which we maximize the number of local checks for SLAs violation is not necessarily optimized over exhaustive search. In some situations, it might be preferable to delay the constraints checks or the bound comparison that follows. Let us recall that while constraints check means that we compare the local SRT and EC with the bound of the service selection problem, bound comparison consists of comparing the local objective function of a subgraph with the best solution, known for the service selection problem.

A favorable situation in exhaustive search can happen when all partial assignments fulfill the SLAs constraints. For exhaustive search, this situation is favorable since the intermediate constraints checks we made in using NeLOs will not detect any violation. Since these local evaluations are time consuming, a challenge for us then consists of adjusting the definition of NeLOs for such situations.

For handling cases where sub-constraints checks are useless, we propose to make the NeLOs more flexible by enriching the data structure with two queues: a binary queue that defines the points at which the constraint check are made and a binary queue that defines the points at which the comparison with the objective function are made.

An example of such an enriched data structure is given in Figure 5. Associated with the NeLO, we put two queues that define when we will compare the local objective values with the global solution and when we will check whether or not SRT or EC are violated.

![Fig 5: in a) the queue of local objective check; in b) the queue of constraints checks; in c) the NeLO](image)

The top entry in these additional queues must always be equal to 1. This is required for checking constraints and evaluating the local objective function on complete assignments. But there is no requirement for the other entries. With these additional data structures, constraints checks can be reported. For instance, in the Figure 5, a report is made after the definition of an implementation to B. The SRT and EC will not be evaluated on (B, g3) because the corresponding entry in Figure 5b) is set to 0. In the same way, the local objective will not be evaluated on (g1, g2) because the corresponding entry in Figure 5a) is set to 0.

With the enriched NeLO we obtain, we can define several search configurations based on NeLOs. For instance, we obtain the exhaustive search in having a unique entry set to
parallelism here uses two types of threads: *mappers* that given a NeLO and a partial assignment of an operations’ graph extend the assignment if no constraints is violated and, *reducers* that given a NeLO and an assignment evaluate the constraints and objective values.

More concretely, let us consider the NeLO of Figure 5. From a *null assignment*, at the beginning, a mapper will start by *extending the null assignment* into an assignment for D and C. Let us suppose that the partial assignment defined at this stage is [(D, D₁), (C, C₁)]. The mapper will not continue further the extension of the partial assignment because according to the NeLO definition, SLAs constraints checks and the evaluation of objective values must be done on the subgraph (g₃, g₄). The mapper will let this task to a reducer and will look for another partial assignment to build. At a moment a reducer will evaluate the SLAs constraints and objective value on [(D, D₁), (C, C₁)]. If the assignment is *valid* (no constraints violations and optimized objective value), it will returns it to a mapper and will look for another reduction task to perform. A mapper will continue on [(D, D₁), (C, C₁)] in assigning an implementation to B. In this parallelization, there are two types of pools used as local memories: pools for assignments that mappers should extend and pools for assignments to be reduced. The run of a mapper starts by the choice of an assignment to perform from the mappers pools. It then extends and puts it in reducers’ pools; inversely, a reducer starts by picking an assignment to reduce from reducers pools and ends by putting the checked assignment in a mapper pool.

The main interest of this parallelization model is that in practice, we can expect to use optimized tools for map-reduce frameworks for defining it. However there are many drawbacks.

The first drawback is that it might be hard to coordinate mappers such as to not explore twice a same assignment. Indeed, the partial assignment [(D, D₁), (C, C₁)] can be the prefix of several distinct complete assignments; once a reducer will put it in the mappers pools, how must it be extended? In our view, for addressing this question, one must add other elements on partial assignments such as to state how to extend them. We will not focus on these aspects in this paper. Let us also notice that we can also have a favorable situation: it is when according to the queues of objectives values and constraints checks, there is only a unique check and evaluation to be done when all operations are assigned to an implementation. In this case (that corresponds to exhaustive search), mappers will each time perform a complete assignment.

The second drawback we see is that one must judiciously decide on the number of mappers and reducers. Ideally, these numbers must ensure that: (1) there are no mappers and reducers that do not work at anytime; (2) one consumes few local memories in the pools of mappers and reducers. Guaranteeing these two conditions is challenging because it depends on: the number of assignments to perform, the

### 4.3 Summary of optimizations of the sequential

We end here the presentation of our sequential algorithm for solving the service selection problem. The summary of the optimizations we performed on the sequential resolution of the service selection problem are given in Table 1.

In the next section, we will then consider the parallelization of this algorithm.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast detection of SLAs violation</td>
<td>NeLO and backtracking exploration</td>
</tr>
<tr>
<td>Check of local objective values</td>
<td>Evaluation of local objectives in NeLOs and Bound comparison</td>
</tr>
<tr>
<td>Runtime optimization in successive searches with NeLOs</td>
<td>Left naming, QoS associative array and virtual reductions</td>
</tr>
<tr>
<td>Flexibility in constraints checks and bound comparison</td>
<td>Binary queues for constraint checks and bound evaluations</td>
</tr>
</tbody>
</table>

Table 1: Summary of optimizations in the sequential resolution.

### 5. Parallel algorithm for service selection

In this part, we will discuss two approaches for the parallelization of the sequential resolution. In the first approach we propose to anchor the parallelization on the tasks defined in NeLO; in the second approach, the parallelization focused on the structure of the search space.

#### 5.1 Task parallelism based on NeLO: a map-reduce approach

A NeLO defines a set of tasks to perform for the evaluation of each assignment. We can range these tasks in two types: *assignments of implementations to operations* and *evaluations of QoS on elementary subgraphs*. For instance, in the NeLO of Figure 5, the assignments of implementations must be done for D, C, B etc. The evaluation of the QoS of (g₃, g₄), (B, g₃), etc.

From this classification, we can derive a map-reduce like parallelization for the evaluation of assignments. The
number of reduction tasks and the time duration of these tasks. Summarizing, if the map-reduce formulation we proposed can work well on exhaustive search, it is hard to define it efficiently in backtracking context. Because of this difficulty, we envisioned in our work another option for parallelization discussed below.

5.2 A data parallelism approach

For avoiding the drawbacks we observed in the map-reduce approach, we propose to consider only one type of thread: workers that combine both the functions of mappers and reducers defined below. The formulation of parallelism according to this view is discussed in the next sections.

5.2.1 Creation of parallelism

We consider a setting where the parallel algorithm will be executed with a number p of threads, defined by the users. In a multicore machine, a choice is to set p as the number of cores. The objective is to decide on the work done by each thread. For this, we propose to give to each thread a part of the service selection search space. For understanding the idea, two points are important: (a) how we represent the search space; (b) how we partition it between threads.

5.2.2 Search space representation

We represent the search space of the service selection problem as a tree. We will refer to it as the services' implementations tree. With each service selection problem, we associate a services' implementations tree by the means of the NeLO generated for the problem. Each level of the tree corresponds either to possible assignments of an abstract operation or, for the root node, to an abstract operation. This means that if the NeLO ordering states the ordering $o_1, o_2, \ldots, o_n$ for operations, then the nodes of level i will correspond to the possible assignments of implementations to $o_i$. In this representation, each branch captures a potential assignment of implementations to operations; the root node does not refer to an assignment but to the first abstract operation. The Figure 6 gives an intuitive idea of this representation.

5.2.3 Search space partitioning

In the parallel execution, each thread runs the sequential backtracking algorithm on a partition of the services' implementations tree. A partition is a sub-tree with the following properties: (1) it has the root node of the general services implementations tree; (2) it has two parts: (2.1) a linear part which is a sequence of nodes of degree 1 (in the partition) that starts from the root node; (2.2) a tree part (linked to the linear one) that is a sub-service implementation tree rooted on a node connected to the linear part.

Starting from the top graph of Figure 6, we give two examples of partitioning. In a), we have two partitions of the services implementation tree while we have 6 in b). In the former case, the linear parts of $T_1$ and $T_2$ are reduced to the node $D$. In the latter case, the linear part of $T_1$ is made of $D_1$ while the linear part of $T_6$ is made of $D, D_2$. For any partition, we will refer to the first node whose degree (in the partition) is greater than 1 as the root computing node. This term reflects the fact that the linear part of a tree defines a partial assignment to complete. For instance, in the linear part $D, D_1, E_1$, we have the assignments in which $D$ is set to $D_1$, $E$ to $E_1$.

A challenging question is to relate the number of partitions to the number of threads. We propose to use two guiding principles for deciding on this number. Firstly, the number of partitions must be greater enough for keeping busy all threads during the execution. Indeed, the partitions correspond to sub-services implementations trees in which threads must find solutions. If we create $p-q$ partitions then $q$ threads will not be busy during the execution. This will result in a loss of efficiency. The second guiding rule is that the number of partitions must be small enough for not dominating the sequential search runtime in each partition. Indeed, the creation of partitions is similar to an exploration. It has a cost that must be kept low enough in order to benefit from the expertise of the sequential backtracking algorithm.

![Fig 6: Example of search space partitioning](image)

From the proposed guiding principles, a good compromise consists in choosing a number of partitions equal to the number of threads. In Figure 6, if we have 2 or 6 threads, then we can adopt the decompositions that the figure describes. If now we have 4 threads, another solution must be found. We propose the following partitioning in 4.

We create a first partition whose root computing node is $D_1$ but that does not explore $E_2$, a second partition whose linear part is $D, D_1$ with $E_3$ as root computing node, a third partition whose root computing node is $D_2$ but which does not explore $E_2$ and a last partition whose linear part is $D, D_2$ and that explores $E_2$.

This solution creates one partition per thread; therefore it matches the compromise that we have from the two principles. However, it highlights a drawback of our compromise: the created partitions do not have the same
sizes; since each thread explores a partition, this suggests a priori a work unbalance. Moreover, if we consider in general partitions of different sizes, then, one can find several partitioning of a same search tree. In Figure 6, we have some partitioning in 2, 3, 4 etc. How to choose among them? For avoiding these difficulties we introduce the following general principle.

**Principle 1:** *(Equal size and sufficient partitioning)*

Given \( p \) threads, let us suppose that the ordering of abstract operations is \( o_1,…,o_n \). We will create \( p' = |Co(o_1)|×…×|Co(o_m)| \) partitions where:
1. Each partition is a sub-tree of the search tree and \( o_1,…,o_m \) is a prefix of \( o_1,…,o_n \);
2. \( p' \) is the smallest product for which
3. The size of the linear part of each partition is \( m-1 \) and the degree of the root computing node is \( |Co(o_m)| \).

We recall that \( Co(u) \) are the possible implementations of \( u \). In this general principle, the point 2 resumes the two guiding principles introduced before. The idea in the last point is both to have a solution for work balance and a simple approach for the creation of partitions. However, the simplicity pursued by the principle has a potential drawback. In practice, the number of partitions can exceed the number of threads. For example in Figure 6, this principle proposes to create 6 partitions when we have 4 threads. In these cases, how to define the work done by each thread?

Our answer is to use the work stealing technique (Blumofe & Leiserson, 1999). At the beginning, we create the threads and a *pool of services partitions (PSP)* that comprises the sub-trees of the search space. Details about this data structure will be provided later. The threads execution continues with the steal of sub-trees in the pool while they do not have something to process. In what follows, we will formalize these executions throughout two models: the one and the multi-levels models.

### 5.3 The one-level execution model

#### 5.3.1 Description

In this model, the parallel algorithm starts by creating threads. We will refer to the first created thread as the *master* and to the others as the *workers*. The first master task is the creation of the PSP. This means that it partitions the services’ implementations tree. Then, the master implements the behavior of the workers. The worker execution consists of stealing a task in the PSP and then searching in the corresponding sub-tree the best solution. For this, they use the sequential backtracking algorithm. Once a local optimum is found, threads (workers or master) update the *current global optimum*; this is the last optimal results found for the service selection problem. This value will, in the next, be compared with lower bounds computed on partial evaluations for reducing the search space. The functioning of the workers is summarized in Figure 7-A).

In the [Stealing] state, the worker tries to retrieve a partition from the PSP. If the steal is successfully concluded (there are unprocessed partitions), the worker enters into the [Processing] state. Otherwise, the execution of the worker ends. In the [Processing] state, the worker checks whether or not the lower bound resulting from the evaluation of the stolen linear part matches the SLAs constraints. If there is no violation, the worker runs the sequential backtracking algorithm; otherwise it sets that it does not have any value to update and returns in the [Stealing] state. Once it found a local optimum, it enters into the [Updating] state where it updates the current global optimum. For the master thread, we have the same functional structure. The only difference is that in the [Initializing] state, the master creates the partitions.

In this execution model, there are two main global variables: the current global optimum and the PSP. In our implementation, the concurrent access to these variables is controlled by a mutual exclusion variable.

We described the key points of the one-level model. It is important to notice that this model has many similarities with the *busy-leaves algorithm* proposed in (Blumofe & Leiserson, 1999) for the general scheduling of multithreaded computations. The difference is that we specify here a particular mechanism for the creation of parallelism (domain decomposition). Below, we analyze the speedup that it can achieve in extreme situations.

![Fig 7: Thread states and partition pool](image)

#### 5.3.2 Best and worst cases of the one-level model

The run of the backtracking algorithm is an exploration of complete and partial assignments. Let us assume that the PSP depth is \( i \in \{1,…,m-1\} \). While the complete assignments correspond to branches of PSP, partial assignments are sub-branches that, starting from the root node, end at a depth \( i \). Given a service composition problem, let us assume that the mean runtime required for exploring a branch until the depth \( i \) is \( \alpha(i) \). Naturally, \( \alpha(i) < \alpha(i+1) \). This is because the deeper we explore a branch, the greater is the exploration runtime. Let us assume that the sequential backtracking algorithm explored \( W_1^i \) complete assignments and \( W_1^p(i) \) partial assignments until the depth \( i \). Then, we can approximate the runtime of the sequential backtracking algorithm by the formula:
We will assume the parallel algorithm runs with \( p \) threads and that the computations are well-balanced. In this setting, we can characterize its runtime by the formula

\[
T_p \approx n \cdot \alpha(m) + \sum_{i=1}^{n-1} W_i^p(i) \cdot \alpha(i)
\]

Let us assume that the parallel algorithm runs with \( p \) threads and that the computations are well-balanced. In this setting, we can characterize its runtime by the formula

\[
T_p \approx \frac{1}{p} (W_1^p \cdot \alpha(m) + \sum_{i=1}^{n-1} W_i^p(i) \cdot \alpha(i)) + O_p
\]

Here \( O_p \) is an overhead induced by synchronization. In the best case scenario, \( O_p \) is negligible. In the general case, it is intuitive to consider that \( W_1^p = W_1^f \); however, in the best case we have \( W_1^p < W_1^f \). Indeed, let us assume that the optimal solution is found in the first branch of the \( p \)th partition. In this case, all threads will quickly have in their execution a strong lower bound for computations. Due to this better bound, some assignments that were completely evaluated in the sequential case will only be partially evaluated in the parallel case. Consequently, we might have \( W_1^p < W_1^f \). This observation can be generalized as follows.

In the best case, some assignments that were completely explored until a depth \( i' \) will only be explored until the depth \( i'' < i' \). Since \( \alpha(i'') < \alpha(i') \), we will have

\[
W_1^f \cdot \alpha(m) + \sum_{i=1}^{n-1} W_i^p(i) \cdot \alpha(i) < T_1
\]

Consequently in the best case the speedup \( S_p = (T_1/T_p) > p \). Hence, we have a super-linear speedup (with \( S_p = p \), we have a linear speedup).

In the worst case, we do not have well-balanced computations. We have \( p \) partitions such that in \( p-1 \) of these partitions, an SLA violation can be detected in exploring the first assignment until the depth 1. The \( p-1 \) threads will steal such partitions and end their executions quickly. The computations will therefore be concentrated in one thread.

We will have:

\[
T_p \approx W_1^p \cdot \alpha(m) + \sum_{i=1}^{n-1} W_i^p(i) \cdot (p-1) \cdot \alpha(1) + O_p
\]

Let us observe that here, we might even have \( W_1^p > W_1^f \). This is because the lower bounds will not necessarily be visited in the order of the sequential algorithm. Consequently, the speedup \( S_p \sim 1 \). With such a speedup, there is no interest in parallelization. Below, we will see how to overcome this worst case situation.

5.3 The multi-levels model

The worst case situation in the one-level model is exacerbated by the fact that \( p-1 \) threads are free while 1 thread is busy. One can avoid this situation if the busy thread could share its work with the others. It is the main idea that brings the multi-levels model. Here, the PSP is no more global but local to each thread. For the master, the pool is created in the [Initializing] state. For the worker, the first pool is created in the [Processing] state. This means that once a partition is stolen, the worker firstly subdivides it in \( p \) partitions according to the principle 1. Then, it processes each partition by using the backtracking algorithm.

When a thread ends the processing and updating of its PSP, it requests another partition from a chosen thread.

In these executions, the worst case situation of the one-level model is delayed since free threads can steal work from busy ones. A question that the model introduces is the choice of threads for stealing. More precisely, for entering in the [Stealing] state, to which thread must a request be sent? Our choice is to randomly select the target. Any free thread can steal work to a busy one.

The multi-levels modeling has a drawback: there might be several partition creations, this is time consuming. For limiting the number of creations, we introduce a parameter that refers to the partitioning level or granularity. In a simplifier manner, a partition created from the entire search tree is at level 1. From a partition of level 1, we create partitions of level 2 and from them partitions of level 3.

Finally, let us remark that when \( l = 1 \), we have the one level model. In the next, we will now discuss about the scalability of the proposed model.

5.4 Discussion about the scalability

In a general manner, the scalability of a parallel algorithm captures its capacity to process larger input by using more resources such as to maintain efficiency.

A critical question when addressing scalability is the measure of the problem input size. In our case, the inputs are given in the definition of the service selection problem: we have a HSG and the domains of services implementations. The question of the scalability can then be addressed for us in two manners: (a) how can we adjust our parallel algorithm such as to use more resources when the operations’ graph is larger? (b) What are our options when the set of possible implementations is larger?

For the former question, let us notice that we proposed through principle 1 to partition the operations’ graph such as to have enough work for each thread. More precisely, given an operations’ graph whose operations are \( o_1, \ldots, o_n \), the principle guarantee that we can create until \( p = \lceil \log_2 n \rceil \times \text{Co} \langle o_1 \rangle \times \ldots \times \text{Co} \langle o_n \rangle \) threads. Moreover, in using a multi-level model, we can even create more parallel tasks. As one can notice, the creation of useful parallelism in our approach is related to the number of operations. Consequently, it is reasonable to conclude that we can create more threads with a useful work when having larger graphs. According to our speedup analysis, this also means that we can expect a greater speedup on larger graphs.

We can formulate a similar answer in the case where the domains of the possible implementations are larger. Indeed, the bigger are the domains, the bigger is the number of threads we can create in the one-level model.

In conclusion, the creation of parallelism in our models is done such as to ensure that we can increase the number of
threads for gaining in speedup when the inputs of the service selection problem are increased. This is what we need for scalable algorithms.

6. Experimental evaluation

In the experiments, we evaluated parallel resolutions of the service selection problem. Here, we assumed that in the sequential resolution, the NeLOs are generated such as to evaluate the objective functions and constraints as soon as possible. We did several experiments for two purposes. The first one was to characterize the speedup that can be expected from the one-level model. The second one was to compare different settings for the multi-levels parallelization.

6.1 Speedup in the one-level model

For the experiments, we focused on the special case of services compositions that implement workflows. We chose two workflows from the Pegasus database (Metha, 2003) that we used for building the operations graph of our compositions.

One can criticize the choice of these flows on two points: (1) we chose small or medium workflow graphs; (2) the chosen graph are regular. For the first point, let us recall that the problem is NP-hard. Since we are providing an exact solution, our expectation is to have real-time results on small or medium problem instances. For the second point, it is important to notice that the regularity of the graph does not modify the runtime of our algorithm. What is important is the set of services implementations.

From each workflow we created a first set of 200 problem instances of services’ compositions. In the instances, each workflow’ activity corresponds to an operation for which we have d distinct implementations. d was set to 40 in the Genelife workflow and 8 in the Motif workflow. Each implementation is characterized by the service response time and energy consumption. The response time was drawn uniformly between 1 and 1500 ms and the energy consumption was computed from the formula $E = P \times S$, where $P$ is a power consumption value randomly drawn between 100,...150. Finally for each problem, we used the SLAs configuration $\text{MaxS} = 3500$, $\text{MaxE} = 7000$.

The value of $\lambda$ in the services' composition problem (see Section 3) was set to 0.5. We subdivided the problem instances in 5 classes of 40. In each class we ran the sequential backtracking algorithm and the parallel algorithm (one-level model) with 2, 4, 8, 16, 32 threads. In Figure 8, we depict the mean speedup that we obtained with various numbers of threads. Let us again recall that the speedup is the ratio between the time obtained from the sequential algorithm and the one of the parallel.

When compared to the theoretical speedup that can be expected (linear speedup), we can notice that the one-level model leads to a near linear speedup. In details however, we observed super-linear cases. The statistics of these expectations are reported in Figure 9.
We justify the occurrence of super-linear speedups with the analysis made in Section 5.3.2. One can also observe that the more we create threads, the smaller is the efficiency of the parallel algorithm (distance between practical and theoretical speedup). However, this is a classical phenomenon in parallel computing: more threads imply more synchronization.

Fig 12: Speedup in the 3-level model.

We also did additional experiments with the multi-levels. The objective was to see whether or not they lead to any improvement. In setting the granularity of parallelism to 2, we clearly observed an improvement for the Genelife workflow. This is summarized in Figure 10, 11, 12, 13 were we computed the ratio between the speedup of the two-levels model and the one of the one-level. This means that working on small partitions can in some cases be beneficial. However, with a granularity set to 3, we did not notice any particular improvement.

7. Conclusions

As middlewares evolve, new challenging computational problems are proposed for improving the servicing of users requests. In the context of large distributed systems, we do believe that for the resolution of these problems, we must adopt a balanced view based on sequential optimization and parallelism. In this paper, we developed this view on the service selection problem; our results state that we might expect super-linear speedups in the optimal resolution of this problem.

For continuing this work, we intend to develop again the frontiers of intelligent techniques that can be used for solving this problem. In particular in a real-time setting, a user SLA can be formulated as a maximal time expected in the resolution of the service selection problem. In this context, we must modify the way we optimize and parallelize the service selection problem such as to obtain a contract based algorithm that depending on the maximal resolution time, will propose the optimal solution that can be expected.

Regarding again intelligence in the resolution, let us notice that we formulated a sequential resolution that can be used for solving this problem. In particular in a real-time setting, a user SLA can be formulated as a maximal time expected in the resolution of the service selection problem. In this context, we must modify the way we optimize and parallelize the service selection problem such as to obtain a contract based algorithm that depending on the maximal resolution time, will propose the optimal solution that can be expected.

Regarding again intelligence in the resolution, let us notice that we formulated a sequential resolution that can be used for solving this problem. In particular in a real-time setting, a user SLA can be formulated as a maximal time expected in the resolution of the service selection problem. In this context, we must modify the way we optimize and parallelize the service selection problem such as to obtain a contract based algorithm that depending on the maximal resolution time, will propose the optimal solution that can be expected.
of the service selection problem. Personally, we do believe that selecting one configuration is not the best option: it is more interesting to investigate in the development of a cooperative search approach (Parkes & Huberman, 2001) in which various configurations are run concurrently.

8. Acknowledgment

The experiments conducted in this work were done on the SMP nodes (40 cores) of the Magi Cluster at the university of Paris 13. http://www.univ-paris13.fr/calcul/wiki/

9. References


Slurm: A highly scalable resource manager (2011)

from http://computing.llnl.gov/linux/slurm


10. Authors

Yanik Ngoko received his B.Sc. in computer science from University of Yaoundé I (UYI), Cameroon, his M.Sc. in computer science also from UYI, and his doctorate in computer science from the Institut National Poly-technique de Grenoble, France. He currently is a researcher in the Laboratoire d'Informatique de Paris Nord (France). His research interests include parallel and distributed computing, web services and energy modeling in cloud computing.

Christophe Céрин has been a Professor of Computer Science at the University of Paris 13, France since 2005. He has served the IEEE Computer Society for many years in different positions varying from Chair of the France CS computer chapter to the organisation of meetings or being Financial Chair for different CS sponsored or co-sponsored conferences. His industrial experience is currently with the Resilience project (Nexedi, Morpho, Alixen, Vifib) related to cloud computing. His research interests are in the field of high performance computing, including grid computing. He is developing middleware, algorithms, tools and methods for managing distributed systems.

Alfredo Goldman received his B.Sc. in applied mathematics from University of Sao Paulo (USP), Brazil, his M.Sc. in computer science also from USP, and his doctorate in computer science from the Institut National Polytechnique de Grenoble, France. He currently is an associated professor in the Department of Computer Science at USP. His research interests include parallel and distributed computing, mobile computing and grid computing.