Future Generation Computer Systems 27 (2011) 341–347

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Decentralized execution of linear workflows over web services

Efthymia Tsamoura *, Anastasios Gounaris, Yannis Manolopoulos

Department of Informatics, Aristotle University of Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 18 January 2010
Received in revised form
25 June 2010
Accepted 26 July 2010
Available online 1 August 2010

Keywords:
Workflow optimization
Web services
Decentralized workflows
Bottleneck cost metric

ABSTRACT

The development of workflow management systems (WfMSs) for the effective and efficient management of workflows in wide-area infrastructures has received a lot of attention in recent years. Existing WfMSs provide tools that simplify the workflow composition and enactment actions, while they support the execution of complex tasks on remote computational resources usually through calls to web services (WSs). Nowadays, an increasing number of WfMSs employ pipelining during the workflow execution. In this work, we focus on improving the performance of long-running workflows consisting of multiple pipelined calls to remote WSs when the execution takes place in a totally decentralized manner. The novelty of our algorithm lies in the fact that it considers the network heterogeneity, and although the optimization problem becomes more complex, it is capable of finding an optimal solution in a short time. Our proposal is evaluated through a real prototype deployed on PlanetLab, and the experimental results are particularly encouraging.

1. Introduction

The rapid spread of wide-area distributed infrastructures, such as the Grid [1], has provided the opportunity to scientific communities ranging from high-energy physics to astronomy and biology to perform computational and data-intensive experiments that were prohibitively expensive in the past. Such experiments are typically expressed as workflows [2]; as a result, the development of mechanisms that provide effective and efficient orchestration and management of workflows has become essential [3]. This need has motivated the development of several scientific workflow management systems (WfMSs), such as Taverna [4], Triana [5], Kepler [6], DAGMan [7], Pegasus [8], GridFlow [9] and Swift [10].

Current WfMSs provide tools that simplify the workflow composition and enactment actions. Effective workflow management capabilities, such as data provenance and user interaction, are also supported, while fault tolerance mechanisms are employed in order to identify and effectively handle failures. Furthermore, WfMSs may provide access to remote data resources and perform complex tasks employing remote computational resources usually through calls to web services (WSs). Nevertheless, one of the main problems in workflow execution, especially when remote calls to WSs are involved, is that of poor performance in terms of throughput and response time. In this article, we focus on improving the performance of long-running workflows consisting of multiple calls to remote WSs, which is a common scenario in e-science (e.g., [11,12]).

The performance of workflow execution is influenced by several factors including resource allocation, subtask scheduling, and the manner in which the constituent services communicate. Scheduling issues are particularly relevant in workflows in which changing the order of WSs results in logically equivalent workflows with different performance characteristics. When there are only a few alternatives, it may be practical for the workflow designers to manually construct the optimal workflow. This is in line with the main concept of existing data integration platforms over the web, such as Yahoo Pipes [13] and DAMIA [14]. Both of them enable a Web 2.0 approach to compose data-intensive queries over distributed data sources, like RSS/Atom feeds and XML files, and provide user-friendly tools for workflow composition; however, users have to explicitly specify the query processing logic procedurally, which is not a trivial task, especially for unskilled users. However, the number of alternative execution plans increases exponentially with the number of services, the order of which may change. As such, advanced workflow optimization algorithms are required.

Optimizing the order of WS calls in a workflow is an important problem that arises in many e-science problems. An example taken from bioinformatics is presented in [15], where, given a set of proteins taken from 12 Bacillus bacterium species, the goal is first to classify the secreted proteins and then to analyze them. In such a workflow, when the order of the performed checks for identifying the kind of a protein (e.g., lipoprotein) is modified, the workflow efficiency changes, too.

At a conceptual level, optimizing the order of WS calls in a workflow resembles the optimization in database query plans. The way in which WSs communicate plays a crucial role. For

* Corresponding author.
E-mail address: etsamour@csd.auth.gr (E. Tsamoura).

0167-739X/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2010.07.009

example, if each WSs processes its whole input before the next service starts its execution, then the optimal ordering depends on the service cost per data item and the service selectivity, and in [16] a very effective algorithm has been proposed to solve this problem. However, typical e-science experiments analyze data in the range of hundreds of megabytes to petabytes, and it is common for each service to process data items (or tuples, following the database terminology) independently. In this case, it is more efficient to execute the workflow in a pipelined fashion rather than sequentially. Pipelined execution allows multiple remote services to process different data items simultaneously, and it can greatly reduce the response time by increasing the throughput. In fact, the response time in pipelined execution is determined by the bottleneck WS, i.e., the WS that spends the most time per input tuple, and Srivastava et al. [17] have proposed an efficient algorithm for this problem.

A main limitation of existing WfMSs (e.g., [4]) and state-of-the-art optimization algorithms, such as the one in [17], is that, even when they provide support for pipelined execution, they assume that each WS passes its results to the next WS through an intermediary or a coordinator. In this work we propose a novel algorithm for building linear WS plans. By linear plans, we mean plans where the output of a service is fed to only one service, and there is a single data resource. The main novelty of our work lies in the fact that we cover the case in which WSs communicate directly with each other, while the data communication costs between the services are different, i.e., there is an heterogeneity in data transfer costs between services. This flexibility comes at the expense of additional complexity, given that the heterogeneous transmission times between WSs must be taken into account. By allowing services to communicate directly with each other, the execution is rendered decentralized.

The main contribution of our work is twofold: (a) the proposal of a novel WS ordering algorithm that is based on the branch-and-bound optimization algorithm for improving the response time of WS workflows executed in a pipelined fashion assuming decentralized execution, and (b) the performance evaluation through real large-scale experiments with a view to obtaining clear insights into the actual benefits of adopting our algorithm. The real measurements are particularly encouraging, since our algorithm can improve on network heterogeneity-oblivious approaches by up to an order of magnitude.

The remainder of this article is structured as follows. Section 2 discusses the related work, while Section 3 presents a use case example. The problem we deal with and the proposed algorithm are presented in Section 4. The evaluation results appear in Section 5, and Section 6 concludes the paper.

2. Related work

Our work relates to the broader areas of distributed query optimization, pipelined operator ordering and workflow management. Distributed query optimization algorithms differ from their centralized counterparts in that the communication cost must be considered, and there is a trade-off between total work optimization and the harder problem of response time optimization [18]. Proposals for the latter case either employ more sophisticated dynamic programming techniques (e.g., [19]) or resort to heuristics.

In [17], an efficient optimization algorithm is presented for optimizing pipelined workflows. A drawback of this algorithm is that it assumes that the output of a service is fed to the subsequent service indirectly through a central management component, and it cannot be extended to cases where arbitrarily distributed services communicate directly with each other. However, it is capable of building execution plans where the output of a service is fed to multiple services simultaneously when the service selectivities are higher than 1. Our work addresses the aforementioned limitation by proposing an efficient branch-and-bound algorithm for the linear optimal ordering of services when the services communicate directly with each other and the communication costs between the services may differ. It can be deemed as an extension to [17] accounting for decentralized execution, except that we are interested in linear orderings only, regardless of the selectivity values.

In the area of pipelined operator ordering, the proposals in [20,21] introduce faster algorithms that produce multiple plans to be executed concurrently with a view to maximizing the dataflow. Along with the definition of the set of interleaving plans, the proportion of tuples routed to each plan is decided as well, in order to maximize the aggregate processing rate. As in our case, all the plans are linear, i.e., each WS has at most one input service and one output. In [22], the goal is to develop solutions for the ordering of operators that are tailored to online, dynamic scenarios. The aforementioned proposals refer to the problem of minimizing the response time. The approximate algorithm in [22] applies to the problem of minimizing the total work.

A common feature of all these algorithms is that they assume homogeneous communication links, i.e., the data transfer costs between any pair of services, independently of the hosts that are deployed to, are equal, which is not the case in our work. The network heterogeneity is taken into account in [23], where the aim is to minimize the aggregate communication cost rather than the response time, and in [24], where the aim is to share data transmission across multiple tasks. To the best of our knowledge, our proposal is the first that aims at minimizing the workflow execution (i.e., response) time when the execution is both pipelined and decentralized and the network heterogeneity is explicitly taken into account.

Our work on workflow optimization is orthogonal to and can be combined with other complementary efforts related to WfMSs developed for e-science applications. A detailed discussion on modern WfMSs can be found in [25]. WfMSs can be broadly divided into two categories: task-based systems and service-based systems. Task-based systems (e.g., [8–10]) generally focus on efficiently mapping and executing abstract workflows. Mapping an abstract workflow instance to an executable form involves detecting (i) the resources that are available and can perform the computations, (ii) the data that is used in the workflow, and (iii) the necessary software. On the other hand, service-based systems (e.g., [4,6]) generally provide interfaces to services for easy workflow management and composition. In the aforementioned systems, workflow optimization, when addressed, deals with scheduling issues, rather than with subtask ordering, as in our case.

3. Use case example

The following example introduces a use case of the problem that we deal with. Abstracting from implementation details, we assume that WSs provide a high-level interface of the form $WS : X \rightarrow Y$, where X and Y are sets of attributes, i.e., given values for attributes in X, WS returns values for the attributes in Y, as shown in the following example adapted from [17]. In the generic case, the input tuples may have more attributes than X, while attributes in Y are appended to the existing ones. It is considered that each WS typically performs operations such as filtering out data items that are not relevant to the query, transforming data items, or appending additional information to each input tuple. The services are pre-allocated on host machines, while they are implemented using at least one input and one output thread, in order to ensure pipelining. The former thread receives tuples and places them in a tuple queue, while the latter takes tuples from the queue, processes them and outputs the results to a subsequent service. However, our approach can be adopted by pull-based workflow engines as well, with minor modifications.

Example 1. Suppose that a company wants to obtain a list of email addresses of potential customers, selecting only those who have a good payment history for at least one card and a credit rating above some threshold. The company has the right to use the following WSs that may belong to third parties and each service is deployed on a different host. The input data containing customer identifiers is supplied by the user.

WS_1 : *SSN id*(*ssn*, *threshold*) → *credit rating* (*cr*)
WS_2 : *SSN id* (*ssn*) → *credit card numbers* (*ccn*)
WS_3 : *card number* (*ccn*, *good*) → *good history* (*gph*)
WS_4 : *SSN id* (*ssn*) → *email addresses* (*ea*)

There are multiple valid orderings to perform this task, although there is one precedence constraint: WS_2 must precede WS_3. The optimization process aims at deciding on the optimal (or near-optimal) ordering under given optimization goals. Two possible WS linear orderings that can be formed using the above services are $\mathcal{C}_1 = WS_2WS_3WS_1WS_4$ and $\mathcal{C}_2 = WS_1WS_2WS_3WS_4$. In the first ordering, first, the customers having a good payment history are initially selected (WS_2, WS_3), and then, the remaining customers whose credit history is below some threshold are filtered out (through WS_1). The $\mathcal{C}_2$ linear plan performs the same tasks in a reverse order. The above linear orderings have different response time, as will be shown later. □

In order to execute such a workflow, users, independently of their language of choice, have to select (i) the services processing a single dataset, (ii) the location of the input data, which can be provided by an additional service that is connected to a data resource, and (iii) the precedence constraints between the services. However, users do not have to explicitly define the services' invocation order, since the latter is optimally found by the proposed algorithm.

4. Optimal linear plan construction algorithm

4.1. Problem statement

Our goal is to build a WS invocation ordering with minimum response time, where data is exchanged directly between services through links having heterogeneous data communication costs. As stated above, the response time of a query plan is controlled by the bottleneck service. Let $\mathcal{C} = WS_0WS_1 \ldots WS_{N-1}$ be a linear plan of N services. Let c_i be the average time needed by WS_i to process an input tuple, σ_i the selectivity of WS_i, and $t_{i,j}$ the time needed to transfer a tuple from WS_i to WS_j. We assume that c_i, σ_i and $t_{i,j}$ are constants and independent of the input attribute values. A service's selectivity is defined as the average ratio of output and input tuples. A WS that receives as input a country name and returns a list of neighboring cities has average selectivity above 1. Similarly, a service that may receive the name of any city in the world and returns airport codes only if the given city is nearby an airport has average selectivity below 1, since, worldwide, there are fewer airports than cities.

Then, for every input tuple to $\mathcal{C}$, the average number of tuples that a service WS_i needs to process is given by

$$R_i(\mathcal{C}) = \prod_{j|WS_j \in P_i(\mathcal{C})} \sigma_j, \tag{1}$$

where $P_i(\mathcal{C})$ is the set of WSs that are invoked before WS_i in the plan $\mathcal{C}$. The average time per input tuple that WS_i spends to process it and to send the results to a subsequent service WS_{i+1} is $R_i(\mathcal{C})(c_i + \sigma_i t_{i,i+1})$. We will refer to the cost $T_{i,j} = c_i + \sigma_i t_{i,j}$ as the aggregate cost of WS_i with respect to WS_j. Recall that the cost of plan $\mathcal{C}$ is determined by the service that spends the most time per input tuple. Thus, the bottleneck cost of a services plan $\mathcal{C}$ is given by

Table 1
Processing cost and selectivity values of the services presented in Example 1.

WS_i	1	2	3	4
Cost of $WS_i(c_i)$	1	3	2	3
Selectivity of $WS_i(\sigma_i)$	0.2	2	0.6	3

Table 2
Communication cost values of the services presented in Example 1.

i/j	1	2	3	4
1	–	11	14	8
2	11	–	10	9
3	14	10	–	7
4	8	9	7	–

$$\text{cost}(\mathcal{C}) = \max_{0 \le i < N} \left(R_i(\mathcal{C})T_{i,i+1}\right). \tag{2}$$

We define $t_{N-1,N} = 0$. If $t_{i,j}$ is equal for all service pairs, the problem can be solved in polynomial time, as shown in [17]. Here we deal with the generic – and more realistic – case, where $t_{i,j}$ values may differ.

Example 2. We continue with Example 1. Table 1 shows the per tuple processing costs and the selectivity values of the services introduced in Example 1, while Table 2 shows the interservice communication costs. Note that $\sigma_2 = 2$ and $\sigma_4 = 3$ mean that every customer has, on average, two credit cards and three email addresses, respectively. In this example, it is assumed that the cost spent to transfer input data to a service is negligible. However, if this hypothesis does not hold, we can realize the different data communication costs by using a source service WS_0 with zero processing cost and $\sigma_0 = 1$. That service is considered to be the source of input data. Regarding the $\mathcal{C}_1$ linear ordering, the average number of tuples that service WS_3 needs to process for every input tuple is $R_3(\mathcal{C}_1) = \sigma_2 = 2$, while the corresponding average number of tuples in $\mathcal{C}_2$ is given by $R_3(\mathcal{C}_2) = \sigma_1\sigma_2 = 0.2 * 2 = 0.4$. Thus, in $\mathcal{C}_1$, the cost spent by WS_3 in order to process an input tuple from the initial data resource and send the results to the subsequent service, which is WS_1, is given by $R_3(\mathcal{C}_1)(c_3 + \sigma_3 t_{3,1}) = 2 * (2 + 0.6 * 14) = 20.8$. Similarly, the processing and transferring cost that is incurred by WS_3 in $\mathcal{C}_2$ is $R_3(\mathcal{C}_2)(c_3 + \sigma_3 t_{3,4}) = 0.2 * 2 * (2 + 0.6 * 7) = 2.48$. Estimating the costs that are incurred by the other services in both linear orderings, we can see that the response time of $\mathcal{C}_1$ is $\max\{23, 20.8, 3.12, 0.72\} = 23$, while the response time of $\mathcal{C}_2$ is $\max\{3.2, 4.6, 2.48, 0.72\} = 4.6$. The bottleneck service, i.e., the one that incurs the maximum cost, is WS_2 in both plans. However, the maximum cost itself is different. The above example shows that two different WS orderings can differ in their response time. □

4.2. Algorithm description

The proposed algorithm is based on the branch-and-bound optimization approach. It proceeds in two phases, namely the *expansion* phase and the *pruning* phase. During expansion, new WSs are appended to a partial plan $\mathcal{C}$, while during the latter phase, WSs are pruned from $\mathcal{C}$ with a view to exploring additional orderings. The proposed algorithm is capable of efficiently exploring the solution space without sacrificing the optimality of the produced plans. To this end, in order to find an optimal solution, the algorithm does not have to build the entire linear plan, but only a part of it. Based upon these partial plans, it decides whether a partial plan has the potential to be part of an optimal plan or not. A prefix plan is just a prefix of a (partial) WS plan. The term, however, is used for partial plans (either already visited or not) that have useful properties. For example, all linear plans having a specific

prefix plan may be optimal, or, otherwise, partial plans with a specific prefix are not further explored for optimality, since their prefixes have bottleneck cost higher than the bottleneck cost of the best plan visited so far. These issues are made clear throughout the rest of this section.

The decision whether to append new services or prune existing ones from a partial plan $\mathcal{C}$ is guided by two cost metrics, ϵ and $\overline{\epsilon}$, respectively. The former corresponds to the bottleneck cost of $\mathcal{C}$, and is given by Eq. (2), while the latter is the maximum possible cost that may be incurred by WSs not currently included in $\mathcal{C}$. If all selectivities are less than 1, $\overline{\epsilon}$ is given by

$$\overline{\epsilon} = \max_{l,r} \left\{ \begin{array}{ll} \left(\prod_{j \mid WS_j \in \mathcal{C}} \sigma_j \right) T_{l,r}, & WS_l \notin \mathcal{C}, WS_r \notin \mathcal{C} \\ \left(\prod_{j=0}^{l-1} \sigma_j \right) T_{l,r}, & WS_l : \text{ last service in } \mathcal{C}, WS_r \notin \mathcal{C} \end{array} \right\}. \quad (3)$$

If there exist $\sigma_i > 1$, then $\overline{\epsilon}$ in Eq. (3) is multiplied by the product of all $\sigma_i > 1$ such that $WS_i \notin \mathcal{C}$.

The algorithm proceeds as follows. It starts with an empty plan $\mathcal{C}$, to which we append the services WS_l and WS_r that incur the minimum aggregate cost $T_{l,r}$. After that, a new plan $\mathcal{C} = WS_l WS_r$ is formed having bottleneck cost $T_{l,r}$. New services may be appended one at a time. In every iteration of the algorithm, we select the service having the minimum aggregate cost with respect to the last service.

The above procedure, i.e., appending new services to the current plan $\mathcal{C}$, finishes when the condition $\overline{\epsilon} \leq \epsilon$ is met. That condition implies that the ordering of the services that are not yet included in $\mathcal{C}$ does not affect its bottleneck cost. As a consequence, all plans with prefix the partial plan $\mathcal{C}$ have the same bottleneck cost. So, a candidate optimal solution $\mathcal{S}$ is found that consists of the current plan $\mathcal{C}$ followed by the rest of the services[1]. The bottleneck cost ρ of the best plan found so far $\mathcal{S}$ is set to $\rho = \epsilon$.

Having found a candidate solution, we explore other plans with potentially lower bottleneck costs. An efficient strategy is to prune the WSs in $\mathcal{C}$ after the bottleneck service, including the latter. Let $\mathcal{C} = WS_0 WS_1 \ldots WS_n$ and WS_i be the bottleneck WS of $\mathcal{C}$, where $0 \leq i \leq n < N$. Then $\mathcal{C}$ is pruned as follows:

$$\mathcal{C} = \begin{cases} \emptyset, & i = 0, \\ WS_0 WS_1 \ldots WS_{i-1}, & 0 < i \leq n < N. \end{cases} \quad (4)$$

The intuition behind Eq. (4) is as follows. WS_{i+1} is the WS such that WS_i has the minimum cost $T_{i,i+1}$. Thus, the cost that may be incurred by any other WS appended to WS_i will be higher than the current bottleneck cost. It is clear that it is worthless to investigate plans with prefix $WS_0 \ldots WS_i$.

After the pruning, we append new services to the new plan $\mathcal{C}$, following the steps described above, i.e., the WS having the minimum aggregate cost with respect to the last service in $\mathcal{C}$ is appended. In order not to rebuild plans that have already been investigated during previous iterations of the algorithm, before $\mathcal{C}$ is pruned, its prefix plan up to the bottleneck WS is inserted in a list $\mathcal{V}$. Then, every time we want to append a new service WS_r to a plan $\mathcal{C}$, the plan produced after the concatenation of $\mathcal{C}$ and WS_r must not appear in $\mathcal{V}$. If any of the plans stored in $\mathcal{V}$ is a prefix of this new plan $\mathcal{C}WS_r$, WS_r is replaced by the next service.

The pruning step is also triggered when the bottleneck cost ϵ of a partial plan $\mathcal{C}$ is higher than or equal to the bottleneck cost of the best service plan $\mathcal{S}$ found so far, i.e., $\epsilon \geq \rho$. It can be shown that the bottleneck cost of a (partial) plan does not decrease if we append other services at the end of that plan. Thus, at a specific algorithm's iteration, the best linear plan not yet visited has bottleneck cost at least equal to $T_{l,r}$. WS_l and WS_r are the two services that (i) are not a prefix of any of the (partial) plans visited so far and (ii) incur the minimum per tuple processing and transferring cost when placed at the beginning of the plan. Based on the above, it is proven that the algorithm can safely terminate when the less expensive pair of WSs that are not a prefix of any plan already visited incurs a cost at least as high as the bottleneck cost of the currently best solution [26].

To summarize, the proposed algorithm consists of the following simple steps. Starting with an empty plan $\mathcal{C}$ and an empty optimal linear plan $\mathcal{S}$ with infinity bottleneck cost, in every iteration of the algorithm, the parameters ϵ and $\overline{\epsilon}$ are computed. If the bottleneck cost ϵ of $\mathcal{C}$ is lower than $\overline{\epsilon}$, then a new service is appended to $\mathcal{C}$ as described above. If the bottleneck cost ϵ of the current plan $\mathcal{C}$ is higher than or equal to the bottleneck cost ρ of the best plan found so far $\mathcal{S}$, then $\mathcal{C}$ is pruned using Eq. (4). Finally, whenever the condition $\overline{\epsilon} \leq \epsilon$ is met, a new solution is found. The last solution is the optimal one. The detailed description of the algorithm, along with the proofs of correctness and optimality, can be found in [26]. In [26], it is also shown how precedence constraints between services (i.e., some services must always execute before others) can be considered through trivial extensions.

The reason for adopting a branch-and-bound-style algorithm and not a linear programming one, as in [17], is that the problem that we deal with is NP-hard (see [27] for the proof), in contrast to the problem introduced in [17]. Thus, a linear programming formulation is insufficient to provide an optimal solution.

Table 3
Aggregate cost matrix **T**.

$i \setminus j$	1	2	3	4	5	6	7	8	9	10
1	–	46	37	42	35	36	40	39	31	29
2	34	–	25	24	29	27	28	25	31	30
3	88	70	–	85	102	73	93	91	91	75
4	65	43	55	–	59	51	48	45	65	48
5	42	44	51	46	–	43	35	42	21	36
6	106	100	92	98	106	–	80	88	104	102
7	34	30	33	28	27	26	–	39	29	31
8	38	29	37	29	34	30	45	–	28	33
9	36	47	45	49	23	41	39	33	–	35
10	37	53	44	43	41	47	48	46	39	–

Table 4
Selectivities of WSs in $\mathcal{W}$.

WS_i	1	2	3	4	5	6	7	8	9	10
σ_i	0.5	0.3	1.5	0.9	0.7	2	0.4	0.5	0.6	0.8

4.3. An example

Let $\mathcal{W} = \{WS_1, \ldots, WS_{10}\}$ be a set of 10 services with corresponding aggregate costs and selectivities shown in Tables 3 and 4, respectively. Fig. 1 shows the partial plans at the end of each iteration.

Initially, the plans $\mathcal{C}$ and $\mathcal{S}$ are empty, and the bottleneck cost of $\mathcal{S}$ is set to ∞. The algorithm starts by identifying the WS pair which incurs the minimum bottleneck cost. The corresponding WSs are WS_5 and WS_9. After that, $\mathcal{C} = WS_5 WS_9$. In the second iteration, since $\epsilon = 21 < \overline{\epsilon} = \sigma_5 \times \sigma_9 \times \sigma_3 \times \sigma_6 \times T_{4,1} = 81.9$ and $\epsilon < \rho = \infty$, a new WS is appended to $\mathcal{C}$, the one having the minimum aggregate cost with respect to WS_9; that service is WS_8. In the third iteration, since $\epsilon = 23.1 < \overline{\epsilon} = \sigma_5 \times \sigma_9 \times \sigma_8 \times \sigma_3 \times \sigma_6 \times T_{4,1} = 40.95$, the service WS_2 is appended to

[1] In our implementation the rest services are placed by ascending selectivity order.

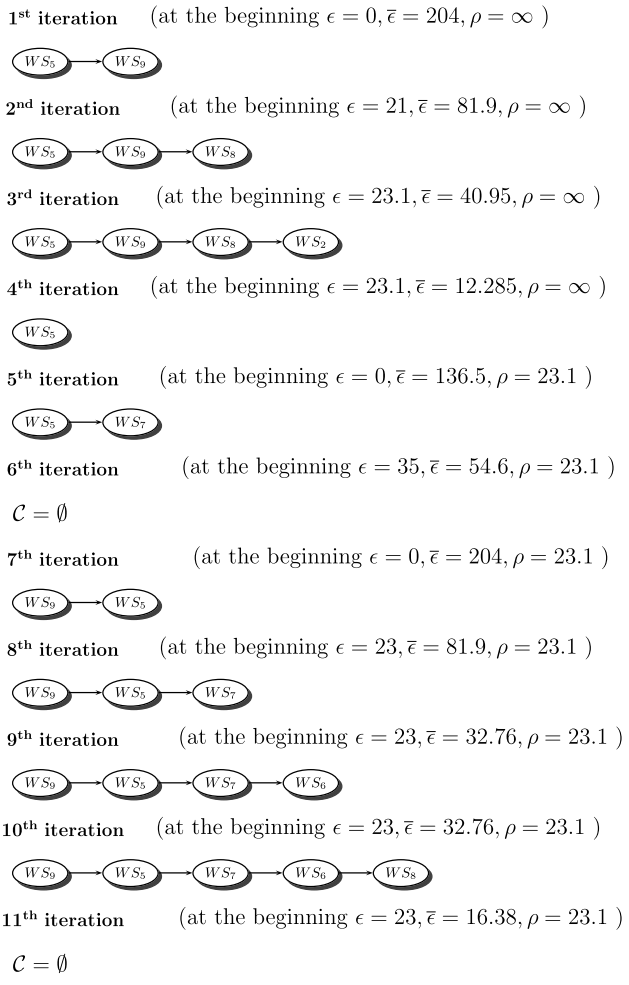

Fig. 1. An example of the proposed algorithm.

$\mathcal{C}$, forming the partial plan $\mathcal{C} = WS_5WS_9WS_8WS_2$. Now, since $\epsilon = 23.1 > \overline{\epsilon} = \sigma_5 \times \sigma_9 \times \sigma_8 \times \sigma_2 \times \sigma_3 \times \sigma_6 \times T_{4,1} = 12.285$, and $\epsilon < \rho = \infty$, a solution is found. Thus, $\mathcal{S}$ is set to $\mathcal{C}$, $\rho = 23.1$ and $\mathcal{C}$ is pruned using Eq. (4). After the pruning $\mathcal{C} = WS_5$ (the bottleneck WS is WS_9). The termination condition is not triggered given that there exists a two-service prefix that has not been investigated and its cost is lower than ρ: $T_{9,5} = 23$.

In the fifth iteration,[2] since $\epsilon = 0 < \overline{\epsilon} = 136.5$ and $\epsilon = 0 < \rho = 23.1$, a new WS is appended to $\mathcal{C} = WS_5$; that is WS_7. In the sixth iteration, the partial plan is set to $\mathcal{C} = \emptyset$, as $\epsilon = 35 > \rho = 23.1$ and the bottleneck WS is the first one, i.e., WS_5. As a result, any other plan starting with WS_5 can be safely ignored.

Since the plan $\mathcal{C}$ is empty, the algorithm searches for the WS pair with the minimum aggregate cost. In our example, this pair consists of WS_9 and WS_5. Note that the requirement none of the plans stored in $\mathcal{V}$ to have prefix the plan WS_9WS_5 is met. In iterations 8–10, new WSs are appended to $\mathcal{C}$, forming the partial plan $\mathcal{C} = WS_9WS_5WS_7WS_6WS_8$. In the eleventh iteration, a new solution is found, since $\epsilon = 23 > \overline{\epsilon} = 16.38$ and $\epsilon < \rho = 23.1$. Thus, $\mathcal{S} = WS_9WS_5WS_7WS_6WS_8$, the bottleneck service is WS_9, and ρ is set to 23. After the pruning $\mathcal{C} = \emptyset$, and the algorithm safely ignore plans starting with WS_9. This causes the algorithm to terminate, since the cost of the less expensive WS pair except those beginning with WS_5 or WS_9, which is WS_2WS_4, is higher than ρ: $T_{2,4} = 24 > \rho = 23$. So the algorithm terminates, after having essentially explored all the 10! orderings in just 11 iterations.

[2] The bottleneck cost of a single-service plan is 0.

5. Evaluation

In the current section, we experimentally evaluate the proposed algorithm, which will be referred to as the Optimal Linear Plan Constructor (OLPC), using the real-world distributed infrastructure of PlanetLab-EU [28]. The evaluation is conducted to investigate, first, the algorithm's performance, and second, the algorithm's efficiency. The performance of the algorithm is evaluated through the comparison of the response times of a wide range of query plans produced by OLPC and the *Greedy* algorithm in [17], which performs service ordering but considers only the computation cost and selectivity of each WS. The efficiency is measured in terms of the absolute time needed to construct the plans and of the number of iterations that the proposed algorithm performs. The experiments presented hereby complement further simulation results in [26], which show that our algorithm is very efficient. Although simulations allow us to investigate the behavior of our algorithm under a wide range of parameters, most of which would not be feasible to measure in real-world experiments, we must also prove that our algorithm behaves well compared to other simpler approaches under the specific level of network heterogeneity encountered in world-scale experiments.

OLPC always produces the optimal serial WS plan. The experiments that follow deal with the extent to which the response times of the Greedy plans are higher than the response times of the OLPC plans. This is done with the help of the response time ratio metric ρ'/ρ, where ρ and ρ' denote the response time of a plan built by OLPC and Greedy, respectively. Recall that the response time of a plan is given by Eq. (2). We restrict the experiments to WSs that have selectivity no greater than 1, since, for this case, both algorithms build linear plans.

Our prototype setting is as follows. The services are deployed on 26 hosts placed in eleven European and Asian counties, namely Greece, Italy, France, Germany, Poland, Spain, Portugal, UK, Sweden, Israel and Thailand. On each remote host we have installed an Apache Tomcat 6.0.9 server and the Axis engine for (de-)serializing the SOAP messages exchanged among the services. The WSs are developed in Java and have an interface of the form WS_i (tuple). The input tuples consist of a tuple identifier and a single data attribute. Each service is implemented using two threads: a consumer one and a producer one. The consumer thread receives input tuples from another service and places them in a queue. The producer thread takes tuples from the queue and sends them to the next service in the pipeline. In our implementation, WSs communicate synchronously. Thus, a service cannot send tuples to another one unless the destination service sends back an acknowledgment message. This message shows that the destination service has successfully placed the previously sent tuple in its queue. In our implementation, the service queues can hold up to 500 tuples. Apart from that, each service has a selectivity σ_i uniformly distributed between 0.3 and 1 and utilizes a random number generator in order to decide when to drop an input tuple or to send it to the next service in the pipelined plan. In wide-area applications, the cost of transferring data typically dominates the processing cost. Consequently, we consider that each WS does not perform any operations on input data and that all tuples contain a string of 512 kB. According to the above, each service WS_i tries to send directly an input tuple to another service in cases where the output of the random number generator is greater than σ_i; otherwise it simple drops it.

In the first set of experiments, we created ten different workflows with 6 random WSs each, so that WSs were not allocated at the same host. Each workflow processes 1000 tuples of 512 kB,

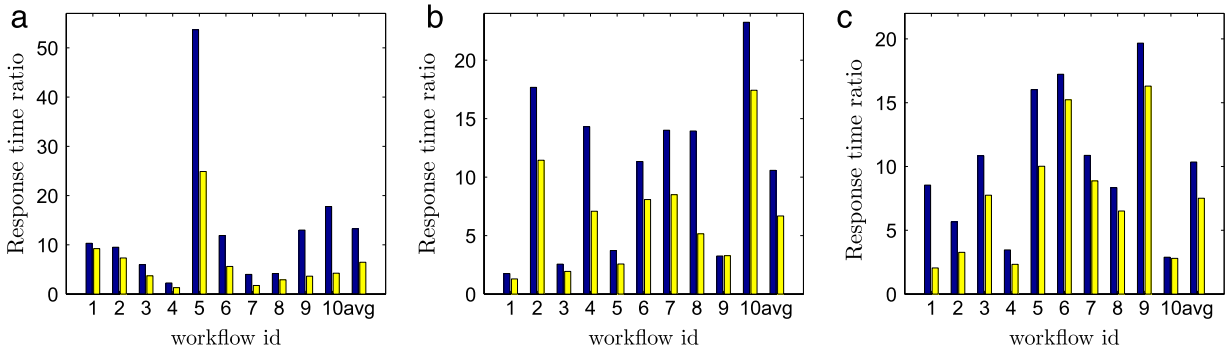

Fig. 2. Results for ten workflows with (a) 6, (b) 10 and (c) 14 services each. The left bars correspond to the estimated response time ratio ρ'/ρ of the Greedy and the OLPC plans after optimization, using Eq. (2), while the actual response time ratio after having executed the plans on PlanetLab correspond to the right bars.

i.e., approximately 500 MB of data. Since PlanetLab is highly dynamic, the communication costs are identified through profiling, which takes place immediately before the execution of each workflow. During profiling, we send 15 tuples between any pair of WSs, e.g. WS_i and WS_j, and we set t_{ij} equal to the median of the 15 communication times.[3] After that, the OLPC and Greedy algorithms are executed in order to build WS plans. Greedy does not actually utilize the communication cost information obtained through profiling. Since the processing cost is not only negligible but also equal for each service, the plans built by the Greedy algorithm are linear orderings by increasing selectivity value; if the costs differed, then the ordering would be by increasing cost [17].

The results are shown in Fig. 2(a).[4] The figure shows (i) the estimated response time ratio ρ'/ρ of the Greedy and the OLPC plans after optimization, using Eq. (2) (left bars); and (ii) the actual response time ratio after having executed the plans on PlanetLab (right bars). Two main observations can be drawn. First, these results confirm the simulation results that the performance improvement can be substantial (up to an order of magnitude). Second, there may be non-negligible differences between the estimated response times and the actual response times. These differences are mainly attributed to the runtime variations of the communication cost values. In other words, the communication costs that were estimated through profiling have changed during the execution of the plans on PlanetLab. This calls for adaptive optimization algorithms, which we plan to investigate in future work. Greedy is more robust to such changes, since it ignores communication costs.

In cases where the response time deviations between the Greedy and the OLPC plans are high, the bottleneck services of the Greedy plans communicate with their descendants through expensive links; for example, the bottleneck service and its immediate one are deployed on PlanetLab hosts located in different continents. For example, in the fifth workflow, the bottleneck service of the plan built by Greedy is deployed on a host in Israel, while its descendant is deployed in Greece.

The same experiment was repeated for workflows with 10 and 14 WSs. The results are shown in Fig. 2(b) and 2(c), respectively. In general, the response time deviations between the two algorithms increase as the number of services increases; the main reason is that the probability to choose hosts placed in different continents becomes higher.

The number of iterations that were performed by the proposed algorithm in order to reach a solution for the different experiments are presented in Table 5. The number of iterations grows slowly with the number of services. In addition, the mean execution time of the proposed algorithm per plan is only 0.3 s on a machine with a dual core 2 GHz processor with 2GB RAM, which can be deemed as negligible. These results constitute a strong proof of the efficiency of the algorithm. In the future, we plan to investigate the average case complexity of the algorithm analytically.

Table 5
Number of iterations performed by OLPC.

Workflow type \ id	1	2	3	4	5	6	7	8	9	10	avg
6 WSs	10	13	3	2	5	17	3	3	3	39	9.8
10 WSs	5	4	6	3	8	12	9	15	3	4	6.9
14 WSs	3	19	7	4	3	6	5	5	8	4	6.4

6. Conclusions

In this work, we deal with the optimization of decentralized workflows consisting of calls to remote services. More specifically, we present an algorithm for finding the optimal ordering of pipelined services when the services communicate directly with each other through links characterized by different transmission times. While different orderings produce similar results, their response times may differ significantly. The optimization goal is to detect the ordering that minimizes the query response time, which, due to parallelism, depends on the bottleneck service. Our algorithm is capable of finding the optimal ordering, and it is of high practical significance since it is fast. We also present experiments using a world-scale infrastructure, which provide strong insights into the actual performance benefits of the proposed algorithm. The main conclusion is that, in real cases, our algorithm can lead to considerable performance benefits compared to approaches that do not consider the network heterogeneity.

Our work can be extended in several ways. Two of the most promising avenues for further research are, first, to support runtime adaptations to changes in the environment, and, second, to investigate nonlinear orderings, even if there is a single data resource.

References

[1] I. Foster, C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, 2nd ed., Morgan Kaufmann Publishers, 2003.
[2] A. Mayer, S. McGough, N. Furmento, M.G.W. Lee, S. Newhouse, J. Darlington, Workflow expression: comparison of spatial and temporal approaches in workflow, in: Grid Systems Workshop, GGF-10, 2004.
[3] G. Bell, T. Hey, A. Szalay, Computer science: beyond the data deluge, Science 323 (5919) (2009) 1297–1298.

[3] We have observed that the WSs installed on the aforementioned PlanetLab hosts (de-)serialize a SOAP message in less than 0.5 s, which is added to the communication cost.

[4] First, we verified that Eq. (2) can capture the actual response time.

[4] T. Oinn, M. Greenwood, M. Addis, N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. Pocock, M. Senger, R. Stevens, A. Wipat, C. Wroe, Taverna: lessons in creating a workflow environment for the life sciences, Concurrency and Computation: Practice and Experience 18 (10) (2006) 1067–1100.
[5] I. Taylor, M. Shields, I. Wang, Resource management of Triana peer-to-peer services, in: Grid Resource Management: State of the Art and Future Trends, Kluwer Academic Publishers, 2003, pp. 451–462.
[6] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the Kepler system, Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows 18 (10) (2005) 1039–1065.
[7] T. Tannenbaum, D. Wright, K. Miller, M. Livny, Condor — A Distributed Job Scheduler, The MIT Press, 2002.
[8] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, D.S. Katz, Pegasus: a framework for mapping complex scientific workflows onto distributed systems, Scientific Programming Journal 13 (3) (2005) 219–237.
[9] J. Cao, S.A. Jarvis, S. Saini, Gridflow: workflow management for grid computing, in: Proc. of the 3st International Symposium on Cluster Computing and the Grid, CCGrid, 2003.
[10] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, M. Wilde, Swift: fast, reliable, loosely coupled parallel computation, in: IEEE Congress on Services, 2007.
[11] D. De Roure, C. Goble, R. Stevens, The design and realisation of the myexperiment virtual research environment for social sharing of workflows, Future Generation Computer Systems 25 (2008) 561–567.
[12] D. Abramson, J. Kommineni, J.L. McGregor, J. Katzfey, An atmospheric sciences workflow and its implementation with web services, Future Generation Computer Systems 21 (1) (2005) 69–78.
[13] Yahoo pipes. http://pipes.yahoo.com/pipes/.
[14] M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. Simmen, A. Singh, Damia: a data mashup fabric for intranet applications, in: Proc. of the 33rd International Conference on Very large Databases, VLDB, 2007, pp. 1370–1373.
[15] T. Craddock, P. Lord, C. Harwood, A. Wipat, E-science tools for the genomic scale characterisation of bacterial secreted proteins, All Hands Meeting, 2006, pp. 788–795.
[16] R. Krishnamurthy, H. Boral, C. Zaniolo, Optimization of nonrecursive queries, in: Proc. of the 12th International Conference on Very Large Data Bases, VLDB, 1986, pp. 128–137.
[17] U. Srivastava, K. Munagala, J. Widom, R. Motwani, Query optimization over web services, in: Proc. of the 32nd International Conference on Very Large Databases, VLDB, 2006, pp. 355–366.
[18] S. Ganguly, W. Hasan, R. Krishnamurthy, Query optimization for parallel execution, in: Proc. of the 1992 ACM SIGMOD International Conference on Management of Data, SIGMOD, 1992, pp. 9–18.
[19] D. Kossmann, K. Stocker, Iterative dynamic programming: a new class of query optimization algorithms, ACM Transactions on Database Systems 25 (1) (2000) 43–82.
[20] A. Deshpande, L. Hellerstein, Flow algorithms for parallel query optimization, in: Proc. of the 24th International Conference on Data Engineering, ICDE, 2008, pp. 754–763.
[21] A. Condon, A. Deshpande, L. Hellerstein, N. Wu, Algorithms for distributional and adversarial pipelined filter ordering problems, ACM Transactions on algorithms 5 (2) (2009) 24–34.
[22] S. Babu, R. Motwani, K. Munagala, Adaptive ordering of pipelined stream filters, in: Proc. of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD, 2004, pp. 407–418.
[23] X. Wang, R.C. Burns, A. Terzis, A. Deshpande, Network-aware join processing in global-scale database federations, in: Proc. of the 24th IEEE International Conference on Data Engineering, ICDE, 2008, pp. 586–595.
[24] J. Li, A. Deshpande, S. Khuller, Minimizing communication cost in distributed multi-query processing, in: Proc. of the 25th IEEE International Conference on Data Engineering, ICDE, 2009, pp. 772–783.
[25] E. Deelman, D. Gannon, M. Shields, I. Taylor, Workflows and e-science: an overview of workflow system features and capabilities, Future Generation Computer Systems 25 (5) (2009) 528–540.
[26] E. Tsamoura, A. Gounaris, Y. Manolopoulos, Optimal service ordering in decentralized queries over web services, Technical report, available from: https://delab.csd.auth.gr/~tsamoura/publications.html.
[27] E. Tsamoura, A. Gounaris, Y. Manolopoulos, Brief announcement: on the quest of optimal service ordering in decentralized queries, in: Proc. of the 29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC, 2010, pp. 333–334.
[28] D. Culler, Planetlab: An open, community driven infrastructure for experimental planetary-scale services, in: USENIX Symposium on Internet Technologies and Systems, 2003.

Efthymia Tsamoura received her Diploma degree in computer science with honors (9.15/10) in 2007. She is currently pursuing a Ph.D. in the Computer Science Department, Aristotle University of Thessaloniki (AUTH).

From October 2007 to June 2008, she was a research assistant with the Centre of Research and Technology Hellas CERTH. Her research interests lie in the field of distributed query optimization and grid computing, as well as image and video analysis processing. She was awarded from the State Scholarship Foundation of Greece, for exceptional performance during the academic years 2003–2004, 2004–2005, 2005–2006 and 2006–2007.

Anastasios Gounaris is a lecturer in the Department of Informatics of the Aristotle University of Thessaloniki, Greece. Prior to that, he was a visiting lecturer with the University of Cyprus and a research associate with the School of Computer Science of the University of Manchester, and he has also collaborated with the Centre of Research and Technology Hellas CERTH. He received his Ph.D. from the University of Manchester (UK) in 2005. He was also awarded an M.Phil. in Computation in 2002 by UMIST (UK) and a B.Sc. in Electrical and Computer Engineering in 1999 by the Aristotle University of Thessaloniki. His research interests are in the area of Grid and distributed data management, resource scheduling in Grids, autonomic computing, adaptive query processing and semantic technologies. He has served as a program committee member at several international conferences and workshops. He is a member of ACM and IEEE. More details can be found at http://delab.csd.auth.gr/~gounaris/.

Yannis Manolopoulos received his B.Eng. (1981) in Electrical Eng. and his Ph.D. (1986) in Computer Engineering, both from the Aristotle University of Thessaloniki. Currently, he is a Professor in the Department of Informatics of the latter university. He has been with the Department of Computer Science of the University of Toronto, the Department of Computer Science of the University of Maryland at College Park and the Department of Computer Science of the University of Cyprus. He has published more than 200 papers in refereed scientific journals and conference proceedings, and he is co-author of four research monographs. His work has received over 2000 citations from over 450 institutional groups. His research interests include databases, data mining, web and geographical information systems, bibliometrics/webometrics, and performance evaluation of storage subsystems.