Decentralized execution of linear workflows over Web Services

Efthymia Tsamoura, Anastasios Gounaris and Yannis Manolopoulos

Department of Informatics, Aristotle University of Thessaloniki, Greece

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 short time. Our proposal is evaluated through a real prototype deployed on PlanetLab, and the experimental results are particularly encouraging.

Key words: workflow optimization, web services ordering, decentralized execution, heterogeneous communication costs, pipeline execution model, bottleneck cost metric

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 more than necessary [3]. This need has motivated the development of several scientific workflow management systems (WfMSs), such as Taverna [4], Triana [5], Kepler [6], DAGMan [7], UNICORE [8] and GridFlow [9].

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., [10]).

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 [11] and IBM DAMIA [12]. 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.

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 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, i.e., the average ratio between output and input data items. In [13], a very effective algorithm has been proposed to solve this problem. However, typical e-science experiments analyze data in the range of hundreds megabytes to petabytes and it is common 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 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 Shrivastava *et. al.* [14] 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 [14], 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. This flexibility comes at the expense of additional complexity, given that the heterogeneous transmission times between WSs must be taken into account. In this work we investigate the scenario in which WSs communicate directly with each other, so that the execution is rendered decentralized. The main contribution of our work is twofold: (a) the proposal of a novel WS ordering 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. The problem we deal with and the proposed algorithm are presented in Section 3. The evaluation results appear in Section 4, and Section 5 concludes the paper.

2. Related work

Our work relates to the broader areas of distributed query optimization and pipelined operator ordering. Distributed query optimization algorithms differ from their centralized counterparts in that communication cost must be considered and there is a trade-off between total work optimization and the harder problem of response time optimization [15]. Proposals for the latter case either employ more sophisticated dynamic programming techniques (e.g., [16]) or resort to heuristics.

In [14], 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 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 afore-mentioned 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 [14] 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 [17, 18] 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 [19], 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 [19] applies to the problem of minimizing the total work.

A common feature of all these algorithms is that they assume homogeneous communication links, which is not the case in our work. The network heterogeneity is taken into account in [20], where the aim is to minimize the aggregate

communication cost rather than the response time, and in [21], 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 time when the execution is both pipelined and decentralized and the network heterogeneity is explicitly taken into account.

3. Optimal linear plan construction algorithm

3.1. Problem Statement

Our goal is to build a WS invocation ordering with minimum response time, where data is exchanged directly between services through heterogeneous links. As stated in the introduction, the response time of a query plan is controlled by the bottleneck service. Let $C = WS_0 WS_1 \dots WS_{N-1}$ 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. Then, for every input tuple to C, the average number of tuples that a service WS_i needs to process is given by:

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

where $P_i(C)$ is the set of WSs that are invoked before WS_i in the plan 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 $R_i(C)(c_i + \sigma_i t_{i,i+1})$. 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 C is determined by the service that spends the most time per input tuple. Thus, the bottleneck cost of a services plan C is given by:

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

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 [14]. Here we deal with the generic –and more realistic– case, where $t_{i,j}$ values may differ.

3.2. Algorithm Description

The proposed algorithm is based on the branch-and-bound optimization approach. It proceeds in two phases, namely the *expansion* and the *pruning* one. During expansion, new WSs are appended to a partial plan C, while during the latter phase, WSs are pruned from C with a view to exploring additional orderings.

The decision whether to append new services or prune existing ones from a partial plan C is guided by two cost metrics, ϵ and $\overline{\epsilon}$ respectively. The former corresponds to the bottleneck cost of 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 C:

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

The algorithm proceeds as follows. It starts with an empty plan 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 $C = WS_l WS_r$ is formed having bottleneck cost $T_{l,r}$. New services may be appended one at a time. In every iteration of the algorithm, we heuristically 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 C, finishes when the condition $\overline{\epsilon} \le \epsilon$ is met. That condition implies that the ordering of the services that are not yet included in C does not affect its bottleneck cost. As a consequence, all plans with prefix the partial plan C have the same bottleneck cost. So, a candidate optimal solution $\mathcal{S}$ is found that consists of the current plan C followed by the rest services in any order[1]. The bottleneck cost ρ of the best plan found so far $\mathcal{S}$ is set to $\rho = \epsilon$.

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

i \ j	1	2	3	4	5	6	7	8	9	10
1	-	10.43	21.89	13.80	29.01	15.35	23.56	21.65	14.63	20.30
2	16.58	-	28.82	21.28	34.15	33.30	42.69	33.97	24.19	32.30
3	34.88	31.96	-	23.21	32.33	32.06	32.51	29.58	42.60	15.23
4	20.52	20.86	20.87	-	31.48	34.93	33.71	32.58	28.62	33.32
5	22.47	21.05	19.94	20.73	-	20.97	19.52	19.31	19.45	17.37
6	18.19	23.51	21.26	25.09	23.76	-	20.50	21.93	22.73	27.31
7	39.60	50.49	34.66	41.50	32.27	32.47	-	31.38	40.49	42.13
8	24.33	27.74	19.41	27.62	18.86	23.41	19.10	-	30.95	13.22
9	14.32	15.46	22.59	18.27	16.89	19.39	20.66	21.96	-	18.97
10	27.08	30.48	10.34	33.14	13.90	39.98	34.23	18.22	28.98	-

Table 1: Aggregate cost matrix **T**.

WS_i	1	2	3	4	5	6	7	8	9	10
σ_i	0.61	0.79	0.92	0.73	0.17	0.40	0.93	0.91	0.41	0.89

Table 2: Selectivities of WSs in $\mathcal{W}$.

Having found a candidate solution, we explore other plans with potentially lower bottleneck costs. An efficient strategy is to prune the WSs in C after the bottleneck service, including the latter. Let $C = WS_0WS_1 \dots WS_n$ and WS_i be the bottleneck WS of C, where $0 \leq i \leq n < N$. Then C is pruned as follows:

$$C = \begin{cases} \emptyset, & i = 0, \\ WS_0WS_1 \dots WS_{i-1}, & 0 < i \leq n < N \end{cases} \tag{4}$$

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 (which is $T_{i,i+1}$ in our case). It is clear that it is worthless to investigate plans with prefix $WS_0...WS_i$.

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

The pruning step is also triggered when the bottleneck cost ϵ of a partial plan C is higher than or equal to the bottleneck cost of the best services plan $\mathcal{S}$ found so far, i.e. $\epsilon \geq \rho$. The algorithm can safely terminate when the less expensive pair of WSs that is not prefix of any plan already visited, cannot improve the current minimum bottleneck cost ρ, which means that the best possible linear plan not yet visited has at least as high bottleneck cost ϵ as ρ.

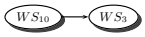
To summarize, the proposed algorithm consists of the following simple steps. Starting with an empty plan C and an empty optimal linear plan $\mathcal{S}$ with infinity bottleneck cost, in every iteration of the algorithm, the parameters ϵ and $\overline{\epsilon}$ are computed. If the bottleneck cost ϵ of C is lower than $\overline{\epsilon}$, then a new service is appended to C as described above. If the bottleneck cost ϵ of the current plan C is higher than or equal to the bottleneck cost ρ of the best plan found so far $\mathcal{S}$, then C is pruned following the 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 [22]. In [22], it also shown how precedence constraints between services (i.e., some services must always execute before others) can be considered through trivial extensions.

3.3. An example

Let $\mathcal{W} = \{WS_1, \dots, WS_{10}\}$ be a set of 10 services with corresponding aggregate costs and selectivities shown in Table 1 and 2, respectively. Fig. 1 shows the partial plans at the end of each iteration.

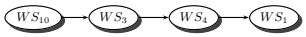
Initially, the plans C and $\mathcal{S}$ are empty and the bottleneck cost of $\mathcal{S}$ is set to ∞. The algorithm starts by identifying the WS pair, which incurs the minimum bottleneck cost. The corresponding WSs are $WS_{10}WS_3$. After that, $C = WS_{10}WS_3$. In the second iteration, since $\epsilon = 10.3425 < \overline{\epsilon} = \sigma_{10} \times \sigma_3 \times T_{7,2} = 41.5973$ and $\epsilon < \rho = \infty$, a new WS is appended to C, the one having the minimum aggregate cost with respect to WS_3; that service is WS_4. In the third

1st iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 50.4958, \rho = \infty$)



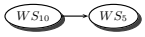
2nd iteration (at the beginning $\epsilon = 10.3425, \overline{\epsilon} = 41.5973, \rho = \infty$)

3rd iteration (at the beginning $\epsilon = 20.7436, \overline{\epsilon} = 30.7074, \rho = \infty$)



4th iteration (at the beginning $\epsilon = 20.7436, \overline{\epsilon} = 18.8983, \rho = \infty$)

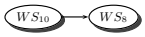
5th iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 45.1255, \rho = 20.7436$)



6th iteration (at the beginning $\epsilon = 13.9069, \overline{\epsilon} = 20.0812, \rho = 20.7436$)

7th iteration (at the beginning $\epsilon = 17.2575, \overline{\epsilon} = 7.29314, \rho = 20.7436$)

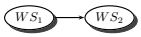
8th iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 45.1255, \rho = 17.2575$)



9th iteration (at the beginning $\epsilon = 18.2236, \overline{\epsilon} = 41.3758, \rho = 17.2575$)

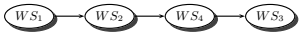
$\mathcal{C} = \emptyset$

10th iteration (at the beginning $\epsilon = 0, \overline{\epsilon} = 50.4958, \rho = 17.2575$)



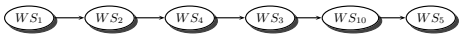
11th iteration (at the beginning $\epsilon = 10.4372, \overline{\epsilon} = 26.2738, \rho = 17.2575$)

12th iteration (at the beginning $\epsilon = 13.0996, \overline{\epsilon} = 17.0269, \rho = 17.2575$)



13th iteration (at the beginning $\epsilon = 13.0996, \overline{\epsilon} = 15.33, \rho = 17.2575$)

14th iteration (at the beginning $\epsilon = 13.0996, \overline{\epsilon} = 13.2628, \rho = 17.2575$)



15th iteration (at the beginning $\epsilon = 13.0996, \overline{\epsilon} = 6.21801, \rho = 17.2575$)

Figure 1: An example of the proposed algorithm.

iteration, since $\epsilon = 20.7436 < \overline{\epsilon} = \sigma_{10} \times \sigma_3 \times \sigma_4 \times T_{7,2} = 30.7074$ and $\epsilon < \rho = \infty$ the service WS_1 is appended to C forming the partial plan $C = WS_{10}WS_3WS_4WS_1$. Now, since $\epsilon = 20.7436 > \overline{\epsilon} = \sigma_{10} \times \sigma_3 \times \sigma_4 \times \sigma_1 \times T_{7,2} = 18.8983$, and $\epsilon < \rho = \infty$, a solution is found. Thus, $\mathcal{S}$ is set to C, $\rho = 20.7436$ and C is pruned following the Eq.(4). After the pruning $C = WS_{10}$ (the bottleneck WS is WS_3). 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: $T_{1,2} = 10.43$.

In the fifth iteration[2], since $\epsilon = 0 < \overline{\epsilon} = 45.1255$ and $\epsilon = 0 < \rho = 20.7436$, a new WS is appended to $C = WS_{10}$; that is WS_5. A new WS is also appended in the sixth iteration forming the partial plan $C = WS_{10}WS_5WS_8$. In the seventh iteration, since $\epsilon = 17.2575 > \overline{\epsilon} = 7.29314$ and $\epsilon < \rho = 20.7436$, another solution is found: $\mathcal{S} = WS_{10}WS_5WS_8$, and its bottleneck cost updates ρ; subsequently, C is pruned following Eq.(4) so that $C = WS_{10}$.

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

In the above iterations, the algorithm does not terminate because the cost $T_{1,2}$ is lower than the ρ values. In the eight iteration, WS_8 is appended to $C = WS_{10}$, while in the ninth iteration, the partial plan is set to $C = \emptyset$, as $\epsilon = 18.2236 > \rho = 17.2575$ and the bottleneck WS is the first one, i.e., WS_{10}. As a result, any other plan starting with WS_{10} can be safely ignored.

Since the plan C is empty, the algorithm searches for the WSs pair with the minimum aggregate cost. In our example, this pair consists of WS_1 and WS_2. Note that the requirement none of the plans stored in $\mathcal{V}$ to have prefix the plan WS_1WS_2 is met. In the iterations 11-14 new WSs are appended to C, forming the partial plan $C = WS_1WS_2WS_4WS_3WS_{10}WS_5$. In the fifteenth iteration, a new solution is found, since $\epsilon = 13.0996 > \bar{\epsilon} = 6.21801$ and $\epsilon < \rho = 17.2575$. Thus, $\mathcal{S} = WS_1WS_2WS_4WS_3WS_{10}WS_5$, the bottleneck service is WS_2 and ρ is set to 13.0996. After the pruning $C = WS_1$, and the algorithm safely ignore plans starting with WS_1WS_2. This causes the algorithm to terminate, since the cost of the less expensive WS pair except those beginning with WS_{10} or WS_1WS_2, which is WS_8WS_{10}, is higher than ρ: $T_{8,10} = 13.2293 > \rho = 13.0996$. So the algorithm terminates, after having essentially explored all the 10! orderings in just 15 iterations.

4. Evaluation

In the current section, we experimentally evaluate the proposed algorithm, which will be referred to as Optimal Linear Plan Constructor (OLPC), using the real-world distributed infrastructure of PlanetLab-EU [23]. The evaluation is conducted to investigate firstly the algorithm's performance, and secondly, 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 [14], 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 [22], 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 twenty-six 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 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 a next service in the pipelined plan. In wide area applications, the cost to transfer 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 create ten different workflows with 6 random WSs each. Each workflow processes 500 tuples of 512KB, i.e. approximately 500MB of data. WSs cannot be allocated at the same host. 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 fifteen tuples between any pair of WSs, e.g. WS_i

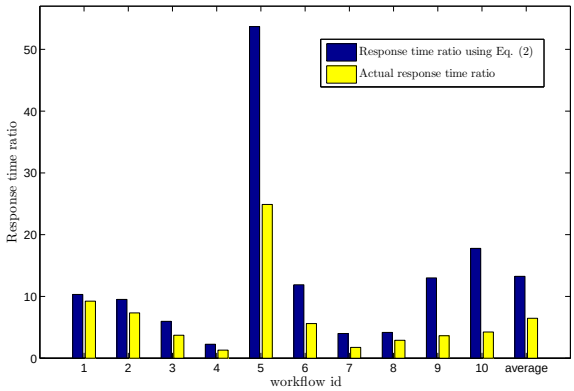


Figure 2: Results for 10 workflows with 6 services each.

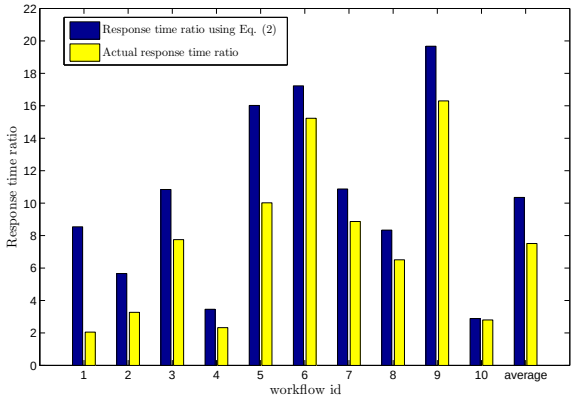


Figure 3: Results for 10 workflows with 10 services each.

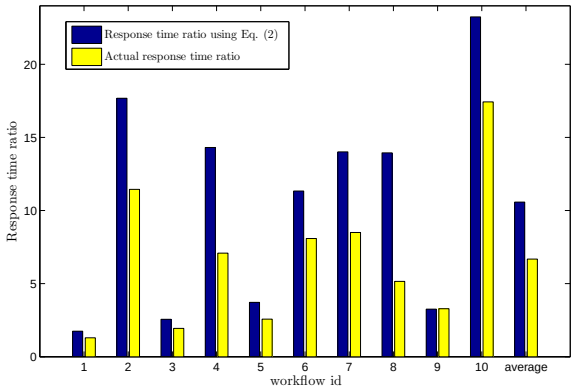


Figure 4: Results for 10 workflows with 14 services each.

and WS_j, and we set t_{ij} equal to the median of the fifteen communication times[3]. After that, the OLPC and Greedy

[3]We have observed that the WSs installed on the aforementioned PlanetLab hosts (de-)serialize a SOAP message in less than 0.5 seconds, which

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

Table 3: Number of iterations performed by OLPC.

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 [14].

The results are shown in Figure 2[4]. The figure shows (i) the estimated response time ratio ρ'/ρ of the Greedy and the OLPC plans after optimization (left bars); and (ii) the actual response time ratio after having executed the plans on PlanetLab (right bars). Two main observations can be drawn. Firstly, these results confirm the simulation results that the performance improvement can be substantial (up to an order of magnitude). Secondly, 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, e.g. 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 is repeated for workflows with 10 and 14 WSs. The results are shown in Figures 3 and 4, respectively. In general, the response times 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 3. 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 sec. on a machine with a dual core 2GHz 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.

5. 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 heterogeneous links and the different orderings produce similar results. The optimization goal is to minimize 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 in considerable performance benefits compared to approaches that do not consider the network heterogeneity.

is added to the communication cost.

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

References

[1] I. Foster, C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, 2nd Edition, Morgan Kaufmann Publishers, San Francisco, CA, USA, 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 (New York, N.Y.) 323 (5919) (2009) 1297–1298.

[4] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. Pocock, A. Wipat, P. Li, Taverna: a tool for the composition and enactment of bioinformatics workflows, Bioinformatics 20 (17) (2004) 3045–3054.

[5] M. S. I. Taylor, I. Wang, Resource management of triana p2p services, in: Grid Resource Management, 2003.

[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, Beowulf Cluster Computing with Linux.

[8] J. Almond, D. Snelling, Unicore: Secure and uniform access to distributed resources via the world wide web, White Paper (1998) 43–82.

[9] J. Cao, S. A. Jarvis, S. Saini, Gridflow: Workflow management for grid computing, in: In 3rd International Symposium on Cluster Computing and the Grid (CCGrid), IEEE CS Press, 2003.

[10] 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.

[11] Yahoo pipes. http://pipes.yahoo.com/pipes/.

[12] 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 Conference on Very Large Databases (VLDB), 2007, pp. 1370–1373.

[13] R. Krishnamurthy, H. Boral, C. Zaniolo, Optimization of nonrecursive queries, in: Proc. of VLDB, 1986, pp. 128–137.

[14] U. Srivastava, K. Munagala, J. Widom, R. Motwani, Query optimization over web services, in: Proc. of the 32nd Conference on Very Large Databases (VLDB), 2006, pp. 355 – 366.

[15] S. Ganguly, W. Hasan, R. Krishnamurthy, Query optimization for parallel execution., in: M. Stonebraker (Ed.), Proc. of SIGMOD, 1992, pp. 9–18.

[16] D. Kossmann, K. Stocker, Iterative dynamic programming: a new class of query optimization algorithms., ACM Trans. Database Syst. 25 (1) (2000) 43–82.

[17] A. Despande, L. Hellerstein, Flow algorithms for parallel query optimization, in: Proc. of the 24th International Conference on Data Engineering (ICDE), 2008, pp. 754 – 763.

[18] A. Condon, A. Despande, L. Hellerstein, N. Wu, Algorithms for distributional and adversarial pipelined filter ordering problems, ACM Transactions on algorithms 5 (2) (2009) 24 – 34.

[19] S. Babu, R. Matwani, K. Munagala, Adaptive ordering of pipelined stream filters, in: Proc. of the International Conference on Management of Data (SIGMOD), 2004, pp. 407–418.

[20] X. Wang, R. C. Burns, A. Terzis, A. Deshpande, Network-aware join processing in global-scale database federations, in: ICDE, 2008, pp. 586–595.

[21] J. Li, A. Deshpande, S. Khuller, Minimizing communication cost in distributed multi-query processing, in: ICDE, 2009, pp. 772–783.

[22] E. Tsamoura, A. Gounaris, Y. Manolopoulos, Optimal service ordering in decentralized queries over web services, in: Submitted for publication.

[23] D. Culler, Planetlab: An open, community driven infrastructure for experimental planetary-scale services, in: USENIX Symposium on Internet Technologies and Systems, 2003.