Practical Algorithms for Execution Engine Selection in Data Flows

Georgia Kougka, Anastasios Gounaris, Kostas Tsichlas

{*georkoug,gounaria,tsichlas*}*@csd.auth.gr*

Abstract

Data-intensive flows are increasingly encountered in various settings, including business intelligence and scientific scenarios. At the same time, flow technology is evolving. Instead of resorting to monolithic solutions, current approaches tend to employ multiple execution engines, such as Hadoop clusters, traditional DBMSs, and stand-alone tools. We target the problem of allocating flow activities to specific heterogeneous and interdependent execution engines while minimizing the flow execution cost. To date, the state-of-the-art is limited to simple heuristics. Although the problem is intractable, we propose practical anytime solutions that are capable of outperforming those simple heuristics and yielding allocation plans in seconds even when optimizing large flows on ordinary machines. Moreover, we prove the NP-hardness of the problem in the generic case and we propose an exact polynomial solution for a specific form of flows, namely, linear flows. We thoroughly evaluate our solutions in both real-world and flows synthetic, and the results show the superiority of our solutions. Especially in real-world scenarios, we can decrease execution time up to more than 3 times.

Keywords: data flows, resource allocation, heterogeneous machines, anytime algorithms

1. Introduction

Our entrance into the era of big data has signalled notable changes in the way scientific research is conducted and enterprises operate. More and more emphasis is put on processing large volumes of data in less, if not real, time in order to accomplish scientific or business intelligence tasks [1, 2]. The most common approach to this end is to design and execute data flows, using

Preprint submitted to Future Generation Computer Systems *September 25, 2014*

workflow tools and platforms that take over the integration of multiple data sources, data manipulation and service orchestration.

Our work is largely motivated by the needs of modern business intelligence (BI) applications and data-intensive scientific workflows, e.g., in bioinformatics. Traditionally, BI builds on top of data-warehouses and data-marts, which are populated by periodic Extract-Transform-Load (ETL) flows. This setting has evolved in two ways. First, flows have become more complex encompassing text analytics and machine learning operations along with data transformation activities. In addition, they operate on both stored data and external, rapidly evolving runtime data, such as feeds and click streams. Second, flows are no longer executed on a single processing engine but their execution may span multiple engines; such flows are also referred to as hybrid flows [3]. Examples of execution engines include Hadoop clusters, traditional DBMS, R scripts and stand-alone tools, each of which may come in several different instances (e.g., both mysql and Oracle RDBMSs) or configurations (e.g., number of reducers in Hadoop) resulting in a big set of candidate execution platforms for executing a single flow (e.g., [4, 3, 5, 6, 7, 8]).

We can follow two main approaches to executing data flows.[1] The first one involves the manual, low-level script-based design of flows, which are then executed in a step-wise fashion. Such an approach is prone to errors and sub-optimal execution, due to the complexity of the flows. The second approach views the workflows at a higher logical level and relies on flow optimizers to decide the technical execution details; this is akin to the role of optimizers in database systems. Optimizing data flows is a challenging multi-dimensional task; two of the most important dimensions include (i) the optimization of the structure of flows, which comes in a form of a directed acyclic graph, but its vertices do not necessarily have clear semantics, as is the case for relational operators; and (ii) the allocation of each of the flow vertices to a potentially different execution engine, choosing among multiple candidates.

Our work focuses on the second aspect mentioned above, and more specifically, aims to devise a mapping of flow nodes to execution engines so that the performance is maximized putting emphasis on keeping the optimization overhead low. The performance is measured in terms of the sum of the execution costs over all flow activities (or flow nodes). For this problem, only

[1] Due to the increased impact of the volume of data in such flows, in the remainder of the paper, we will use the terms workflows, data flows or simply flows interchangeably.

simple heuristic or non-scalable algorithms are known to date [4]; here we show how we can significantly improve upon the state-of-the-art. Moreover, we show how we can benefit from the existence of multiple execution engine options, rather than sticking to simple single-engine solutions. The main challenges of tackling this problem are posed by the following factors: the number of flow nodes and candidate engine or engine configurations may be large, the engines are heterogeneous in the sense that each engine is capable of executing a flow node in different time, and shipping data from one execution engine to another or switching between engines incurs cost, i.e., choosing the best execution engine for each flow node in isolation does not imply optimality [4].

Overall, we make the following contributions:

- We propose a set of anytime algorithms (Section 3) that, as shown in our experimental section, they are capable of yielding mappings of flow nodes to execution engines that are significantly better than naive approaches (Section 2), even when the flows are very large and our proposals are allowed to run only for a few seconds on an ordinary machine. These anytime algorithms fall into three main categories: branch and bound, random walk and set-cover ones.

- We propose an optimal solution with polynomial time complexity for the specific case, where the flow structure is linear, i.e., the flow is a chain of activities. Specifically, we present a polynomial dynamic programming algorithm that can yield exact solutions for linear flows and can act as an efficient approximate solver in more generic cases (Section 4).

- We evaluate our proposals using both real flows and synthetic in a wide range of settings. We declare winners among our proposals, depending on the type of the flow. In summary, the value of our solutions lies in that they are both effective in improving performance and easy to implement and light-weight. The dynamic programming approach performs remarkably well in many real-world settings and along with the anytime heuristics, we perform consistently better than current heuristics. The anytime proposals can run for any number of iterations tolerated by the users, e.g., to meet real-time constraints, and they are capable of yielding improved performance in short time. Especially when the flows are near-linear, as happens in many real-world cases, the execution cost can be decreased by more than 3 times. If the flows are completely linear, the improvements are even larger (Section 6).

- We prove the $\mathcal{NP}$-hardness of the problem at hand (which means that no solution with polynomial complexity can be found in the generic case) and at the same time it is impossible to approximate it within a small constant, unless $\mathcal{P} = \mathcal{NP}$ (Section 5).

2. Problem Definition and Background

In this paper, we will investigate resource allocation techniques, where each flow activity can run on multiple execution engines, of which, only one should be selected. At this point, we will not consider the optimization of the ordering of flow activities or the technical specifications of the available processors. To begin with, we represent the logical view of a flow as a directed acyclic graph (DAG), where each activity corresponds to a node in the graph and the edges between nodes represent intermediate data shipping among activities. Since we have different activity implementations for a specific engine or multiple engines, each flow activity has a processing cost in time units, which differs between engine instances or engine configurations. Additionally, data transfer from one engine to another and/or switching between engines has also a cost.

The main notation and assumptions of this Flow Activity Allocation problem (henceforth named FAA) are as follows:

- Let $G = (A, E)$ be a directed acyclic graph, where A denotes the nodes of the graph and E represents the data flow among the nodes, i.e., which activity feeds data to which activity.

- Let $A = \{a_1, ..., a_n\}$ be a set of (possibly streaming) activities of size n. Each flow activity is responsible for one or both of the following tasks: (i) reading or retrieving or storing data, and (ii) manipulating data. The definition of the activities and the complete flow G is left to the flow designer.

- Let $E = \{edge_1, ..., edge_{n'}\}$ be a set of edges of size n'. Each edge $edge_i, 1 \leq i \leq n'$ equals to an ordered pair (a_j, a_k), so that $edge_i^{head} = a_j$ and $edge_i^{tail} = a_k$.

- Let $ENG = \{e_1, ..., e_m\}$ be a set of execution engines that activities can be allocated to; ENG's size is m. In general, the number of execution engines tends to be smaller than the number of flow activities. However, different engine instances and/or configurations (e.g., multiple Hadoop clusters, each with varying number of reducers) are essentially treated as different engines, so that the number of different engines at our disposal may well be larger than the number of the flow activities. Note that nowadays, it has become easier to support multiple execution engines for each activity; for example, in [5], it is discussed how a logical data flow activity definition can automatically be translated to several distinct physical implementations according to the underlying execution engine including SQL, pig-latin and PDI[2] scripts.

- Let $c_{i,j}$ be the execution time of an activity a_i when mapped to engine e_j. We assume that this information is available, through e.g., micro-benchmarking as in [6], and we do not deal with the engine configuration ourselves.

- Let $ce^{a_k}_{e_{i \to j}}$ be the cost associated with the graph edges. It consists of a) the engine switching cost from engine e_i, which executes activity a_k, to engine e_j, which executes the subsequent activity; and b) the data shipping from the output of activity a_k (executed on engine e_i) to the subsequent activity. The subsequent activity is the activity the edge points to. The first component depends on the two engines, while the second depends additionally on the data volume transferred across the edge; this volume depends on the sender activity. Overall, ce depends on the sender activity and the execution engines of the activities connected through the edge. As above, we assume that this metadata is available to our algorithms either through micro-benchmarking or through log files. We can support arbitrary settings of $ce^{a_k}_{i \to i}$ values denoting the edge cost for activities running on the same engine instance; however, in the remainder, we assume that $ce^{a_k}_{i \to i}$ cost from engine e_i to engine e_i is 0, because there is no data transfer over the network and/or engine configuration changes involved and we will refer to the ce cost as *inter-engine* cost.

Our goal can be stated as the derivation of an allocation function f :

[2] http://www.pentaho.com/product/data-integration

$[1, n] \to [1, m]$, which expresses the mapping between activities and processor engines, so that (i) the *total execution time is minimized*; (ii) each activity is mapped to one and only one engine; and (iii) our allocation algorithms run in seconds at most. Generally, we denote the mapping between an activity a_i and a processor engine e_j as $f(i) = j$, where $1 \leq i \leq n$ and $1 \leq j \leq m$. The total execution time TET for a specific allocation is the sum of all the execution costs of each activity on their engines plus the cost of transferring data and switching between different engines. The latter occurs whenever two nodes of an edge belonging to E are allocated to distinct engines:

$$TET = \sum_{i=1}^{n} c_{i,f(i)} + \sum_{i=1}^{n'} ce^{edge_i^{head}}_{f(edge_i^{head}) \to f(edge_i^{tail})}$$

In a more generic scenario, there are constraints between allocations to denote the fact that not all activities can run on all engines. In the constrained case, the goal is to allocate all activities so that the total execution time cost is minimized subject to the allocation constraints.

As explained later, our solutions behave differently depending on whether the flows are linear or not. Linear flows are those that contain one and only one activity with no incoming edges and one and only one activity with no outgoing edges; all the other activities have exactly one incoming edge and one outgoing edge.

2.1. Motivational Example

A real-world data flow, which has the role of analyzing emerging temporal trends, is illustrated in Figure 1. The data flow builds a taxonomy of current trends for a specific region, which are extracted from Twitter messages (tweets), and its purpose is to categorize the trends and derive key representative features.

This example flow comprises 14 activities for deriving the timestamp from tweets (*Extract timpestamp*), deriving the textual content (*Extract textual content*), performing look-up operations on auxiliary datasources (*LookupRegion*, *LookupTrends*), executing tasks that correspond to ordinary relational database operators (*Select, Join, Aggregation*), and performing analysis operators (*Qualitative Analysis*, *Label trends*, *Quantitative Analysis*).

In this example, we assume that 6 of the activities can execute on 2 candidate engines and 5 of the activities can execute on 3 candidate engines. The engines can be MapReduce engines, GPU accelerators and O-RDBMSs.

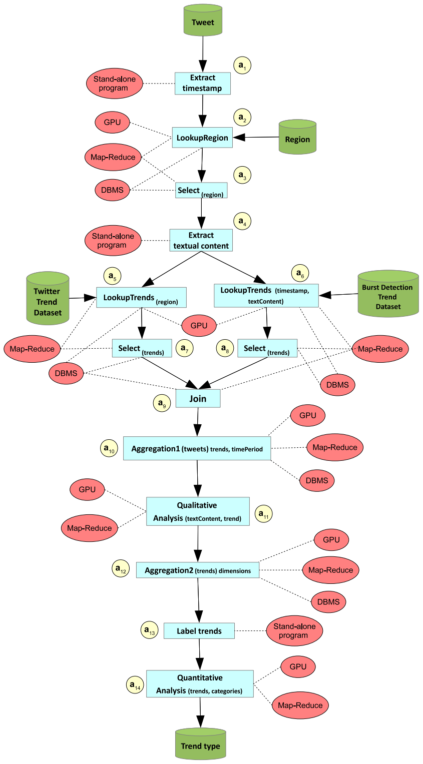


Figure 1: A real-world data flow for interpreting emerging temporal trends.

The remainder 3 activities are performed using stand-alone scripts. In such a setting, the total number of different engine allocations in the figure is $2^6 3^5 = 15,552$. However, for each engine, there may multiple engine instantiations (not shown in the figure). If, for example, there are 3 different instances per execution engine and stand-alone programs, there are more than $7.4 \cdot 10^{10} < 3^{14}(2^6 3^5)$ possible allocations. It is easy to see that this number increases exponentially in the number of available engines. Furthermore, moving data from one engine to another, e.g., from a Hadoop cluster to a database is associated with a time overhead. Also, not all activities can run on any engine; for example, the last activity can run only with the help of a GPU-based implementation or as a Map-Reduce program in a specific cluster. Our goal is to devise a concrete mapping of each flow activity to an execution engine in a small amount of time.

In the remainder of this section, we present first, an exhaustive solution, upon which we later propose improvements, and second, heuristics that are fast albeit not very efficient in improving performance.

Algorithm 1 Exhaustive Search

```
Require: $G(A, E), ENG, \mathbf{C}, \mathbf{CE}, \mathbf{CONSTR}$
f ← ∅
$mincost \leftarrow \infty$
for all $i = 0 \dots m^n - 1$ do
    $f_{candidate} \leftarrow mapNumberToAllocation(i)$
    if $f_{candidate}$ satisfies constraints in CONSTR then
        $cost_{candidate} \leftarrow calculateCost(f_{candidate}, \mathbf{C}, \mathbf{CE})$
        if $cost_{candidate} < mincost$ then
            $mincost \leftarrow cost_{candidate}$
            $f \leftarrow f_{candidate}$
        end if
    end if
end for
return an engine allocation plan $f$
```

2.2. An exhaustive solution

The rationale of an exhaustive methodology is to estimate all the possible combinations of engine allocations with regards to all flow activities. For a flow with n activities and m available execution engines, we have the following auxiliary matrices: (i) the $n \times m$ $\mathbf{C}$ matrix, where the element in the i-th row and j-th column is the $c_{i,j}$ execution time cost defined earlier; (ii) the $n \times m \times m$ $\mathbf{CE}$ matrix, where the element with the (k, i, j) coordinates is the $ce^{a_k}_{i \to j}$ inter-engine data shipping and engine switching cost; and (iii) the $n \times m$ $\mathbf{CONSTR}$ matrix, where the element in the i-th row and j-th column is set to 1 if the activity a_i can be mapped to the engine e_j.

The exhaustive algorithm iterates over all possible m^n allocations of execution engines to nodes. Due to this exponential complexity, it can be only applied to tiny flows, e.g., flows with very few nodes and candidate execution engines. In the exhaustive algorithm, each allocation is mapped to a distinct number in the range $[0, m^n - 1]$ with the help of the *mapNumberToAllocation* function. We can imagine each allocation plan as a number with n digits of base m. The value v of the i-th digit from right to left denotes that the i-th activity is allocated to the engine $v + 1$. For example, the allocation number $(3521)_6$ denotes a mapping of a 4-node flow to engines, where $m = 6, f(1) = 2, f(2) = 3, f(3) = 6$ and $f(4) = 4$. Since the allocations with the smallest and the largest numbers are $(0000)_6$ and $(5555)_6$, respectively, all possible allocations are in the range of $[0, 6^4 - 1]$.

2.3. Heuristics

Another approach of allocation is to apply simple heuristics in order to avoid the complexity of estimating all the possible allocation combinations. For the purposes of this paper, we investigate two different heuristics, similar to those mentioned in [4]:

H1: this is a 2-step heuristic. First, we rank all engines based on their average execution cost for all flow activities in increasing order, i.e., the value for e_j is $\frac{1}{n}\sum_{i=1}^{n} c_{i,j}$. Then, we allocate each activity to the engine with the highest rank that is capable of executing that activity.

H2: this is also a 2-step heuristic. First, we rank all engines based on their execution cost for each flow activity separately. Then, we allocate each activity to the engine with the highest rank that is capable of executing that activity.

In Figure 2, we show an example for a linear and a non-linear flow with $n = 5$ and $m = 3$. Because of the constraints between the engines, some allocations are not considered and the corresponding cells in C are shaded. Additionally, in this example we consider a CE matrix; for simplicity this matrix is $m \times m$ assuming that the values are the same for all n. For both flows, *H1* and *H2* provide a single allocation plan as shown in the figure; this is because the two flows differ only in their edges, which are not considered by naive heuristics.

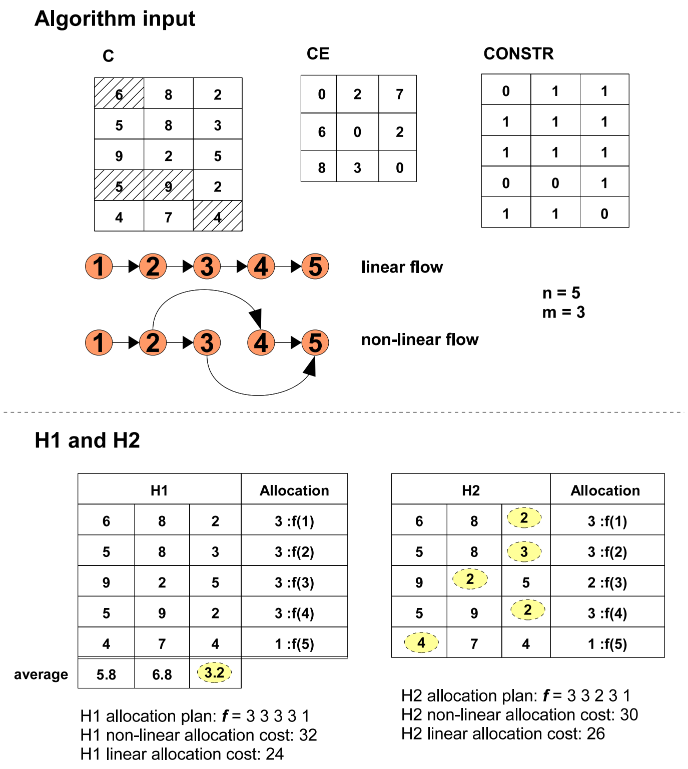
In the remainder of this work, we will refer to those solutions as simple or naive heuristics to distinguish them from our proposals in the next two sections.

3. Anytime algorithms

We now introduce anytime algorithms that can be stopped at any point and they are guaranteed to move closer to the optimal allocation the longer they are allowed to run. In a sense, the exhaustive algorithm can be classified as an anytime algorithm, too. But here, we present three types of solutions that are both efficient and effective, as verified by our experiments.

3.1. A branch and bound solution

A branch-and-bound (**BB**) approach can improve upon the naive exhaustive algorithm of the previous section. More specifically, we can perform the



Algorithm input

C

6	8	2
5	8	3
9	2	5
5	9	2
4	7	4

CE

0	2	7
6	0	2
8	3	0

CONSTR

0	1	1
1	1	1
1	1	1
0	0	1
1	1	0

n = 5
m = 3

H1 and H2

	H1			Allocation
	6	8	2	3 :f(1)
	5	8	3	3 :f(2)
	9	2	5	3 :f(3)
	5	9	2	3 :f(4)
	4	7	4	1 :f(5)
average	5.8	6.8	3.2	

H1 allocation plan: ***f*** = 3 3 3 3 1
H1 non-linear allocation cost: 32
H1 linear allocation cost: 24

H2			Allocation
6	8	2	3 :f(1)
5	8	3	3 :f(2)
9	2	5	2 :f(3)
5	9	2	3 :f(4)
4	7	4	1 :f(5)

H2 allocation plan: ***f*** = 3 3 2 3 1
H2 non-linear allocation cost: 30
H2 linear allocation cost: 26

Figure 2: An application of *H1* and *H2*.

following two main improvements. First, we can calculate the flow execution cost after each activity gets allocated and to abandon an intermediate allocation plan as soon as it exceeds the current minimum cost, which is the minimum of *H1* and *H2* algorithms for the first time. Second, when an allocation of an activity to a node is found not to satisfy the engine constraints, we move the allocation id counter as many steps as required in order not to examine a similarly invalid allocation of the same engine to the same node. For example, let us suppose that the $(3300)_6$ allocation is invalid because the 3rd node cannot run on engine 4. Instead of examining $(3301)_6$, $(3302)_6$, and so on (which are bound to be invalid as well), we move directly to $(3400)_6$.

Although, the two afore-mentioned improvements can yield speedups in the decision taking time, the computational complexity remains exponential, which renders the algorithm unsuitable for use in large flows. A remedy to this complexity problem is to cap the number of the allowed iterations.

More specifically, we derive the **BB-IC** (Branch-n-Bound-Iteration Capping) algorithm, which, in addition to the two previous improvements allows only a pre-specified number of iterations (termed as *noi*). Given that threshold, the algorithm first estimates the maximum number n_{max} of nodes the allocation of which can be examined without exceeding the iteration threshold: $n_{max} = \lfloor log_m(noi) \rfloor$. Then, it runs the *H1* and *H2* and it keeps the best performing one. From the allocation plan of the best performing heuristic, it detects the n_{max} most expensive nodes, and investigates all their possible allocations using the branch-and-bound approach. The rationale behind this is to investigate other allocations for the parts of the flow that contribute the most to the total cost. The remaining nodes are allocated according to the allocation of the best performing heuristic.

Compared to the Algorithm 1, the main changes are in three places: (i) the iteration of i is up to $m^{n_{max}} - 1$ in line 3; (ii) $f_{candidate}$ corresponds to a plan with a subset of activities where the allocation of the remainder activities is defined by the best performing simple heuristic; as such, the cost estimation in line 6 needs to take this into account, and (iii) after line 11, we insert an else statement to increase the value of i, as explained above.

3.2. Random-walk solutions

Our second approach to coping with the complexity of the problem is to explore the search space with random walks. We examine three main variants:

RW: Starting from the allocation derived from the best performing heuristic between *H1* and *H2*, we make random perturbations for a pre-specified number of times; this number is the *length* of the walk. In each iteration, we choose an activity in a round-robin fashion and we randomly alter its allocation.

RWR-r: This flavour extends the previous one by restarting the random walk r times. Each time, the starting point is a randomly selected allocation of all activities.

RWR-b: This flavour also employs restarts, but the starting point is the best performing allocation detected thus far (see Algorithm 2).

3.3. Dealing with large flows

For flows with large sets of activities and candidate engines, *BB-IC* and *RW* can explore only a very small part of the search space. E.g., in *BB-IC*, if $n = m = 100$ and $noi = 10000$, then n_{max} is only 2. The two algorithms

Algorithm 2 RWR-b

```
Require: $G(A, E), ENG, \mathbf{C}, \mathbf{CE}, \mathbf{CONSTR}, length, r$
$f \leftarrow best(H1, H2)$
$mincost \leftarrow calculateCost(f)$
for all $i = 1 \dots r$ do
    $f_{candidate} \leftarrow f$
    for all $j = 1 \dots length$ do
        make a random change in $f_{candidate}$ that satisfies constraints in CONSTR
        $cost_{candidate} \leftarrow calculateCost(f_{candidate}, \mathbf{C}, \mathbf{CE})$
        if $cost_{candidate} < mincost$ then
            $mincost \leftarrow cost_{candidate}$, $f \leftarrow f_{candidate}$
        end if
    end for
end for
return an engine allocation plan $f$
```

presented below, employ set-cover approaches in order to prune the search space; more specifically, they preprocess the candidate engines, and select a subset of *ENG* before applying the *BB-IC* and *RW* solutions. The intuition is that if, for large flows, we can derive a much smaller engine candidate set than the initial one, *BB-IC* and *RW* can improve their performance.

SC1: This set-cover based approach reduces the *ENG* set as follows. In each iteration, we count the number of activities each engine is allowed to execute, and we select the engine that is capable of processing the most activities. Conflicts are resolved arbitrarily. Then, we remove the activities supported by that engine and we proceed to the next iteration, unless there are no activities left. After we have selected the subset of engines, we apply both *BB-IC* and *RWR-b* (which run in very short time) and we choose the allocation with the lowest cost.

SC2: The $SC2$ is another set-cover flavour, which takes into account the inter-engine cost. More specifically, it performs the first iteration exactly as *SC1* does. Then, in each subsequent iteration, it chooses the engine with the lowest average inter-engine cost with respect to the last added engine across all activities. Similarly to *SC1*, this procedure continues until all the activities can be executed on at least one engine, and the final allocation is found after applying both *BB-IC* and *RWR-b* to the reduced engine set.

Algorithm 3 DP-cost

```
Require: $G(A,E), ENG, \mathbf{C}, \mathbf{CE}, \mathbf{CONSTR}$
for all $j = 1 \dots m$ do
    if $f(1) = j$ satisfies constraints in CONSTR then
        $DP_{cost}(1,j) \leftarrow C(1,j)$
    end if
end for
for all $i = 2 \dots n$ do
    for all $j = 1 \dots m$ do
        if $f(i) = j$ satisfies constraints in CONSTR then
            $k_{min} \leftarrow min_{1 \leq k \leq m}\{DP_{cost}(i-1,k) + CE(i,k,j)\}$
            $DP_{cost}(i,j) \leftarrow C(i,j) + DP_{cost}(i-1,k_{min})$
            $DP_{nodes}(i,j) \leftarrow k_{min}$
        end if
    end for
end for
```

The maximum number of iterations in the pre-processing engine selection phase for both set-cover approaches is m, but in practice, it is a small fraction of m and thus, the pre-processing step runs in a few milliseconds on a simple modern machine. Additionally, the number of iterations or restarts of BB-IC and RWR-b algorithms define the actual execution of the *SC* flavours that can be classified as anytime, too.

3.4. A hybrid solution

According to our experience, each of the previous anytime solutions may exhibit the best performance in different settings. Since all of them are lightweight and explore the search space in different ways, it is both possible and effective to run all of them and choose the best each time. Therefore, we introduce the **BEST** meta-heuristic that, after executing all *BB-IC*, *RWR-b*, *SC1* and *SC2*, chooses the one that yields the allocation plan with the lowest execution cost. As shown in Section 6, we can further increase the performance benefits by more than 10% because of that.

4. Dynamic programming

The previous proposals put emphasis on improving the naive heuristics without significantly raising the optimization overhead. Here, we propose a dynamic programming proposal that can find the optimal solution for linear flows and can act as an approximate solver for arbitrary flows.

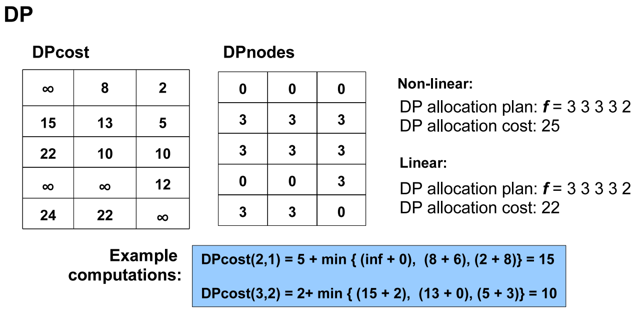
4.1. Detailed Description

The rationale of the **DP** algorithm is to calculate the cost of increasingly larger portions of A, i.e., it starts of flows containing only a_1, then it examines flows containing (a_1, a_2), and so on, until it examines the complete flow. When examining flows with the first i activities, we consider the allocation costs of the flow consisting with the first $i-1$ activities. We employ a DP_{cost} matrix of size $n \times m$, where each cell (i, j) denotes the optimal cost of the plan with the first i activities when $f(i) = j$. The first row is initialized with the activity costs in $C[1, *]$. For the other rows, we have $DP_{cost}(i, j) = C(i, j) + min_{k \in [1,m]}\{DP_{cost}(i-1, k) + CE(k, j)\}$. We also employ an auxiliary matrix, DP_{nodes}, which, in each cell (i, j), stores the engine for which the last part of the sum expression in the recursive formula is minimized. Overall, the last row of the DP_{cost} contains the costs when all activities are considered for all possible allocations of the last activity. In Algorithm 3, we show how the matrices are populated. The exact allocation is found by recursively examining the rows of the DP_{nodes} matrix from bottom to top (not included in the pseudocode).

When the flow is linear, the *DP* algorithm finds the optimal cost of an allocation; that cost is the minimum cost in the last row of DP_{cost}. To find the allocation function f, we start from the minimum value of the last row of DP_{cost}, the column of which denotes the allocation of the last activity $f(n)$; then, with the help of DP_{nodes}, we can recursively find the allocations $f(n-1), f(n-2), \ldots, f(1)$. Interestingly, the algorithm can be employed as an approximate solver for arbitrary flows. In that case, we can run the algorithm as previously and build the allocation plan, but such an allocation is not guaranteed to be optimal.

In Figure 3, the allocation plan of *DP* for the metadata of example in Figure 2 is the same for both flows as well, i.e., $f = (3, 3, 3, 3, 2)$. We can see that for both flows, *DP* yields a better solution than *H1* and *H2*. The allocation of *DP* is optimal for the linear case, but it is sub-optimal for the non-linear case.

Additionally, we should mention that for reasonable values of *noi*, *r* and *length*, *BB*, *BB-IC* and the random walk flavours find the optimal solution for both flows. The optimal solution of the non-linear case is $f = (3, 3, 2, 3, 2)$ with total cost 22 instead of 25.



DP

DPcost

∞	8	2
15	13	5
22	10	10
∞	∞	12
24	22	∞

DPnodes

0	0	0
3	3	3
3	3	3
0	0	3
3	3	0

Non-linear:
DP allocation plan: ***f*** = 3 3 3 3 2
DP allocation cost: 25

Linear:
DP allocation plan: ***f*** = 3 3 3 3 2
DP allocation cost: 22

Example computations:

DPcost(2,1) = 5 + min { (inf + 0), (8 + 6), (2 + 8)} = 15

DPcost(3,2) = 2+ min { (15 + 2), (13 + 0), (5 + 3)} = 10

Figure 3: An application of *DP*.

4.2. Analysis

The time complexity of DP is $O(nm^2)$ because the size of the DP_{cost} matrix is $n \times m$ and in order to fill in each cell, the algorithm examines all m values of the cells in the previous row. As shown in the experiments, for a few hundreds of nodes and engines, the algorithm terminates in a few seconds. The space complexity is $O(nm)$, because of the $n \times m$ size of the DP_{nodes} matrix, which stores intermediate allocation. Note that, although we assume an $n \times m$ DP_{cost} matrix, we only need to keep two rows each time, thus the space complexity depends on DP_{nodes}. Below, we provide a sketch of the proof that DP is correct for linear plans; due to space limitations, we prove only that the cost found by DP is optimal.

Theorem 1. DP *finds the minimum cost of a linear flow.*

Proof. A sketch using induction on the size of the set A is as follows. If $n = 1$, the optimal solution is trivial and is found by the algorithm. Let the algorithm find the optimal solution $OPT(n, j)$ for $n = x$ and all engines $1 \leq j \leq m$. Assume now that $n = x + 1$. For the cost of the $(x + 1)^{th}$ running on e_j, the DP algorithm examines, for all possible allocations of the x^{th} activity, the sum of the allocation cost of the first x activities and the inter-engine cost $ce^x_{f(x)\to j}$. The first part of that sum is optimal. The second part of the sum corresponds to the cost incurred by an edge $(x, x+1)$, which is the only real edge that exists between the first x activities and the $(x+1)^{th}$ one. Thus, for $n = x+1$, DP examines the whole set of valid combinations of optimal allocations of the first x activities plus the inter-engine cost between the first x activities and the $(x + 1)^{th}$ one, i.e., it does not miss any valid solution. □

5. Theoretical Analysis

In the following we prove that the FAA (Flow Activity Allocation) problem is not only $\mathcal{NP}$-hard (the corresponding decision problem is $\mathcal{NP}$-complete) but at the same time it is impossible (unless $\mathcal{P} = \mathcal{NP}$) to approximate it within a small constant factor. The proof concerns a simplification of FAA, where $c_{i,j} = 1, 1 \leq i \leq n, 1 \leq j \leq m$ (uniform engines and activities with unit-processing times) while $ce^{a_k}_{i \to j} = 0$ when $i = j$ and $ce^{a_k}_{i \to j} = 1$, when $i \neq j$. We consider the case where the number of engines m can be arbitrary. In case where the number of engines is restricted we can get similar but slightly better results with respect to the approximation ratio bound. The proof is based on a transformation of the scheduling problem $P\infty|prec, c = 1, p_j = 1|C_{max}$ (we use the notation introduced in [9] to denote scheduling problems). In this scheduling problem, the number of engines is arbitrary ($P\infty$), there are precedence constraints ($prec$) with unit-processing times for the activities ($p_j = 1$), there is a unit-time communication cost among engines ($c = 1$) and the goal is to minimize the makespan, that is the total length of the schedule. Note, that the simplified FAA problem could be represented as $P\infty|prec, c = 1, p_j = 1| \sum C_j$, since the goal is to minimize the total activity completion time. The following theorem is stated without a proof in [10], which we provide here for the sake of completeness.

Theorem 2. *The simplified FAA is $\mathcal{NP}$-hard and cannot be approximated by a polynomial-time algorithm with approximation error bound less than* $8/7$.

Proof. [11] provides a polynomial-time transformation to prove that the decision scheduling problem $P\infty|prec, c = 1, p_j = 1|C_{max}$ is $\mathcal{NP}$-complete. In particular, given an instance S of the 3-SAT problem, we construct an instance J for the scheduling problem. In a nutshell, for each variable in S, 6 activities are constructed and for each clause in S, 13 activities are constructed. Appropriate precedence constraints between these activities are enforced so that S has a truth assignment if and only if there is a schedule of J with makespan 6. This means, than in the case where S is a YES-instance of 3-SAT, then all activities in J are processed in the time interval $[0, 6]$, while in the case where S is a NO-instance then in every possible feasible schedule of instance J there is at least one activity that completes at time 7 or later.

Let there be a polynomial-time approximation algorithm for the problem in the current paper with approximation $8/7 - \epsilon$, $\epsilon > \frac{19}{114k}$. We construct

k copies of the instance J, $J_1, J_2, \ldots, J_k$, adding the precedence constraint that all activities in instance J_i are predecessors to all activities in J_{i+1}, $1 \leq i \leq k-1$. Let the resulting schedule be denoted by J^*. If S is a YES-instance of 3-SAT, then instance J^* has a schedule with total activity completion time equal to $57mk^2$. This quantity is computed based on the precedence graph of the activities related to variables and clauses (see [11]). If S is a NO-instance of 3-SAT then the earliest possible time that J_i can start is $7i-7$. Based on the precedence graph we get that at each integer time in the range $[7i-6, 7i-3]$ at most $4m$ activities can be completed. At time $7i-2$ at most m activities can be completed and finally at time $7i-1$ at most $2m$ activities can be completed. As a result, the total completion time of J_i is at least $133mi - 76m$, which when summed for all i, gives that the minimum total completion time for all activities in J^* is $66.5mk^2 - 9.5mk$.

From the above discussion we get that a polynomial-time approximation algorithm for our restricted problem with approximation ratio strictly less than $8/7 - \frac{19}{114k}$ is impossible unless $\mathcal{P} = \mathcal{NP}$, since this algorithm could be used to distinguish between the YES- and NO-instances of 3-SAT. This also proves the fact that the FAA problem is $\mathcal{NP}$-hard. □

6. Evaluation

In this section, we conduct a thorough evaluation of the solutions presented in the previous sections. We use both synthetic and real-world flows. First, we examine real-world flows, where the focus is on the performance (Section 6.1.1) and testing under “real-world” conditions. For the latter, we examine two main aspects: the impact of inaccuracy in the statistical metadata (Section 6.1.2) and behaviour under settings where, intuitively, employing multiple engines does not seem promising; e.g., when all tasks can be executed on any engine, and the inter-engine costs are an order of magnitude higher than task processing costs (Section 6.1.3). The real-world flows are taken from [12]; they correspond to data-intensive scientific scenarios from several disciplines including astronomy, earthquake hazard characterization, biology and physics, and they are commonly used in the evaluation of techniques for data flows (e.g., [13]).

The purpose of the synthetic flows' experiments is to unveil the strengths and weaknesses of each allocation algorithm in random flow instances. In the synthetic flows, we focus on the following dimensions: (i) performance of the alternatives presented in terms of the estimated flow execution cost

(Section 6.2.1); (ii) associated overhead of each solution in terms of real time spent in reaching allocation decisions (Section 6.2.2); (iii) accuracy, which refers to the deviation of the approximate flow execution costs compared to the optimal ones and is examined only when the optimal solution can be found in reasonable time (Section 6.2.3); and (iv) sensitivity analysis, which investigates the impact of different flavours and parameter values for the random walk proposals (Section 6.2.4).

In order to cover a wide range of settings, the flows vary in terms of the number of activities, engines, execution time of activities, transfer and switching costs between engines, and density of the DAG representing the flow. The probability of an engine to be capable of executing a specific activity is set to 50% unless otherwise stated. The activity and the inter-engine cost values are uniformly distributed in the range [1,100]. To generate the inter-engine cost values we take into account the engine-independent amount of data outputed by each activity. The default settings of the random walk solutions are: number of restarts $r = 50$ and length of walk $l = 10^3$. The default setting of the number of iterations (noi) for the *BB-IC* solution is 10^4. These values are set in such a way that the anytime algorithms complete in a few seconds at most.

All the algorithms are implemented in MATLAB and the experiments were executed on a machine with an Intel Core i5 660 CPU and 6GB of RAM. All experiments were repeated 50 times and we report the average values (except in Table 5).

6.1. Real-World Flows

We experimented with 4 real-world flow structures, described in [12]. In particular, we created instances of the following flow types: Montage, Epigenomics, LIGO and CyberShake (see Figure 4). For those flows, we experimented with $n = m = 100$ and the rest of the settings as in the introduction of this section. Initially, we assume that each activity processes the same amount of data and the inter-engine connection speed is the same for all pairs of engines; this implies that the inter-engine costs are activity-independent but we relax this assumption later.

6.1.1. Performance

In this experiment, we evaluate the relative performance in terms of execution time TET of the different policies (see Table 1). The numbers are

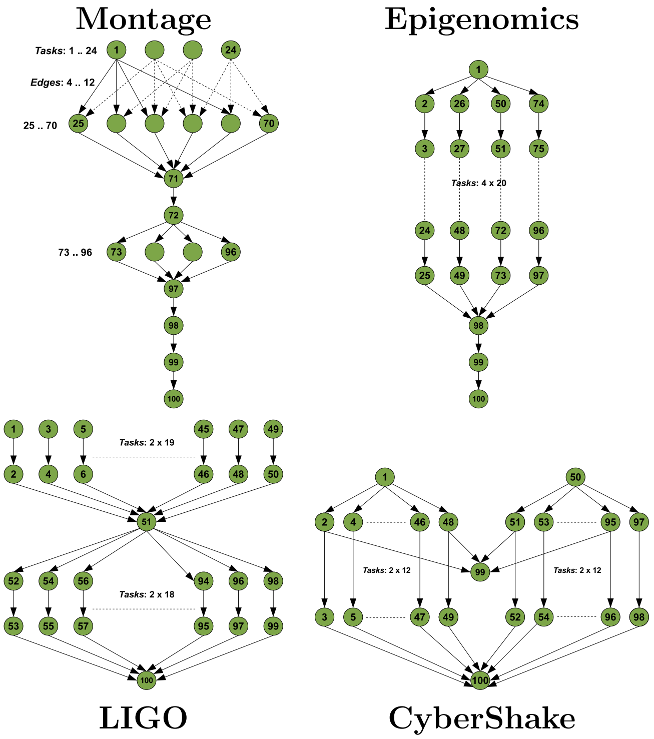


Figure 4: The structure of the real workflows used in our experiments.

Table 1: Normalized performance for real-world flows with 50% engine constraints

ACCURATE STATISTICS								
flow/alg	***H1***	***H2***	***DP***	***BB-IC***	***RWR-b***	***SC1***	***SC2***	***BEST***
Montage	1.3355	1.4083	1.4043	1.2555	1.2362	1.1578	1.0815	**1**
Epigenomics	1.5147	1.0282	**0.3208**	1.0057	1.0118	1.3652	1.1720	1
LIGO	1.3559	1.0512	**0.7601**	1.0245	1.0358	1.2604	1.0843	1
CyberShake	1.2858	1.1806	**0.9751**	1.1267	1.1304	1.1577	1.0489	1

Table 2: Normalized performance for real-world flows with 50% engine constraints and inter-engine cost activity-dependent.

ACCURATE STATISTICS								
flow/alg	***H1***	***H2***	***DP***	***BB-IC***	***RWR-b***	***SC1***	***SC2***	***BEST***
Montage	1.3463	1.2507	1.2589	1.1427	1.0984	1.1443	1.0984	***1***
Epigenomics	1.9009	1.2167	***0.4664***	1.1390	1.0911	1.5464	1.3353	1
LIGO	2.0327	1.4644	***0.9433***	1.3526	1.2668	1.5341	1.3177	1
CyberShake	1.5200	1.3433	1.1649	1.1749	1.1308	1.2738	1.1899	***1***

normalized according to the execution cost yielded by *BEST*. For the montage flow, the *BEST* meta-heuristic is the best performing policy, while the

Table 3: Normalized performance for real-world flows with 50% engine constraints

INACCURATE STATISTICS								
10% inaccurate statistics								
flow/alg	***H1***	***H2***	***DP***	***BB-IC***	***RWR-b***	***SC1***	***SC2***	***BEST***
Montage	1.0660	1.1054	1.1035	1.0373	1.4565	1.1331	1.2793	**1**
Epigenomics	1.4548	0.9874	**0.3107**	1.0000	1.8556	1.6620	1.7467	1
LIGO	1.3206	0.9844	**0.7026**	1.0000	1.7263	1.4803	1.5855	1
CyberShake	1.1728	1.0381	**0.8469**	1.0215	1.5245	1.2302	1.3635	1
30% inaccurate statistics								
flow/alg	***H1***	***H2***	***DP***	***BB-IC***	***RWR-b***	***SC1***	***SC2***	***BEST***
Montage	1.1082	1.1542	1.1512	1.0649	1.5024	1.2354	1.3023	**1**
Epigenomics	1.4429	0.9789	**0.3034**	1.0000	1.8527	1.6091	1.7258	1
LIGO	1.2789	0.9890	**0.7188**	1.0000	1.7321	1.4334	1.5649	1
CyberShake	1.1614	1.0558	**0.8744**	1.0390	1.5639	1.2124	1.3491	1

Table 4: Normalized performance for real-world flows with no engine constraints

Inter-engine cost ∈ [1,100] and no engine constraints								
flow/alg	***H1***	***H2***	***DP***	***BB-IC***	***RWR-b***	***SC1***	***SC2***	***BEST***
Montage	1.0502	3.4899	3.42259	1.0307	1.0453	1.1453	1.0196	**1**
Epigenomics	1.0293	1.2458	**0.2980**	1.0018	1.0248	1.1788	1.0359	1
LIGO	1.0328	1.4764	**0.9932**	1.0023	1.0265	1.1653	1.0396	1
CyberShake	1.0645	2.2775	1.8019	1.0387	1.0615	1.1628	1.0288	**1**
Inter-engine cost ∈ [1,1000] and no engine constraints								
flow/alg	***H1***	***H2***	***DP***	***BB-IC***	***RWR-b***	***SC1***	***SC2***	***BEST***
Montage	1.4840	32.4686	30.3408	1.4739	1.4761	1.1429	1.0012	**1**
Epigenomics	7.0332	1.0136	**0.5915**	1.0012	1.0099	8.1907	7.1936	1
LIGO	1.2568	13.6752	7.7084	1.2489	1.2514	1.1293	1.0037	**1**
CyberShake	1.2459	21.2047	15.6062	1.2424	1.2429	1.1289	1.0002	**1**

naive heuristics $H1$ and $H2$ yield 33% and 40% higher execution cost, respectively. However, the pattern changes for the rest of the real-world flow types. Those flows are not linear but they comprise linear subflows. So, *DP* outperforms the other policies. For Epigenomics, *DP*'s execution time is more than 3 times lower than those from the best performing heuristic, which is *H2*, and *BEST*. For LIGO and Cybershake, the DP's performance benefits are higher than 20% compared to the simple heuristics.

We now relax the assumption regarding the homogeneity of inter-engine network and the volume of data processed by each activity. More specifically, we experiment with scenarios where the inter-engine cost is a linear function of the data volume, and each activity may alter this volume by a factor uniformly drawn from 0.5 to 1.5 (denoting the pruning of half of the data and generating half as much additional data, respectively). The results are shown in Table 2. We observe that the heuristics yield relatively better performance for the montage flow, but the performance degradation with regards to *BEST* remains significant (25%). For the other three types of real-world flow structures, we observe that both *H2* and *DP* exhibit worse performance than the one reported in Table 1 and difference between our best performing proposal and the best performing heuristic widens. This is attributed to the fact that the more the heterogeneity in the cost associated with the graph edges, the more the need to consider these costs carefully, something that *H2* does not perform and *DP* performs only partially (since it considers only some of the existing edges). In the remainder and in order to keep the evaluation concise, we will discuss mostly the case, where inter-engine costs are assumed to be activity-independent showing that even for that setting our solutions manage to yield improvements.

Regarding the time overhead, this is a couple of seconds even for the most time consuming techniques, such as the dynamic programming and random walk. Detailed experiments for the decision making overhead are presented later.

6.1.2. Imprecise Statistical Metadata

So far, we have assumed that the statistics in **C** and **CE** (the execution time and inter-engine shipping and switching costs) are accurately known. Here, we relax this assumption and we allow for imprecise statistics. We repeat the first part of the experiment in Section 6.1.1, but after we determine the allocation, we perturb the values in **C** and **CE** and we re-evaluate the total cost. In particular, we multiply each element in the two cost matrices with a scalar value $\alpha \in [0.9, 1.1]$ (denoting inaccuracies of ±10%) or $\alpha \in [0.7, 1.3]$ (denoting inaccuracies of ±30%). In this way, we emulate a situation, where the actual costs differ from those used during decision taking.

The results are shown in Table 3. For smaller inaccuracies of up to 10%, *DP* stills outperforms the other policies for the last 3 flow types. The performance improvements vary from 15% up to more than 3 times. For Montage flows, *BEST* performs better, as in the case with no inaccuracies. However, the difference of *BEST* from the naive heuristics drops to 6.6%. When the inaccuracies grow larger, this difference is up to 10% for Montage flows. For such inaccuracies, *DP* clearly outperforms all the other policies for the other flow types.

6.1.3. Settings discouraging multiple execution engines

Intuitively, one might expect that when allowing any engine to run the complete flow (i.e., not having engine constraints), then not switching between engines, as in *H1*, yields the highest performance. However, as shown in the top part of Table 4, for the Epigenomics flow, *DP* still achieves more than 3 times lower execution cost. For the rest of the real-world flows, the improvements are significantly lower (between 3.2% and 6.4%). *H2* produces much worse results.

In addition, in real world, one might expect the inter-engine data transfer and switching costs to dominate. So, we perform another experiment, where the **CE** values are an order of magnitude higher than the values in **C**. The results are presented in the lower part of Table 4. Our solutions in that case improve the performance from 24.5% to 42%.

6.2. Synthetic Flows

The results regarding the real-flows provide strong insights into the strengths of our solutions but they are tailored to the specific flows examined. To complement the evaluation, we randomly generate DAGs. The flows considered consist of n=10, 20, 50, 100, 200 activities. We categorize the flows depending on their number of activities, as *small* (10 or 20 activities), *medium* (50 activities), *large* (100 activities) and *very large* (200 activities), based on the categorization in [14]. Regarding the exact shape of the flow graph G, we consider *dense* flows, where the probability of two activities to be connected with an edge is 50% (i.e., there exist $\frac{n(n-1)}{4}$ edges), *sparse* flows, where the edge probability is 20% (i.e., there exist $\frac{n(n-1)}{10}$ edges), and *linear* flows, where activity a_i is connected only with a_{i+1}.

The number of the available engines is m=10, 20, 50, 100, 200. The Montage flow is the one closer to the random flow instances tested below. Also, the sparse flows are less sparse than the rest of the flows in Section 6.1.

As in the experimental setting of real-world flows, by default we assess the performance improvement where the inter-engine cost is activity-independent, but we later relax this to show that the our results hold for a wide range of inter-engine cost values.

6.2.1. Performance improvement

In the first set of experiments, we evaluate the flow performance in terms of flow execution time. We compare the two heuristics *H1* and *H2* against *DP*, *BB-IC*, *RWR-b*, *SC1*, *SC2* and *BEST*, which is the best among the last four. For the random walk flavours, we choose only the best performing one, and we leave their comparison for Section 6.2.4. The average results of

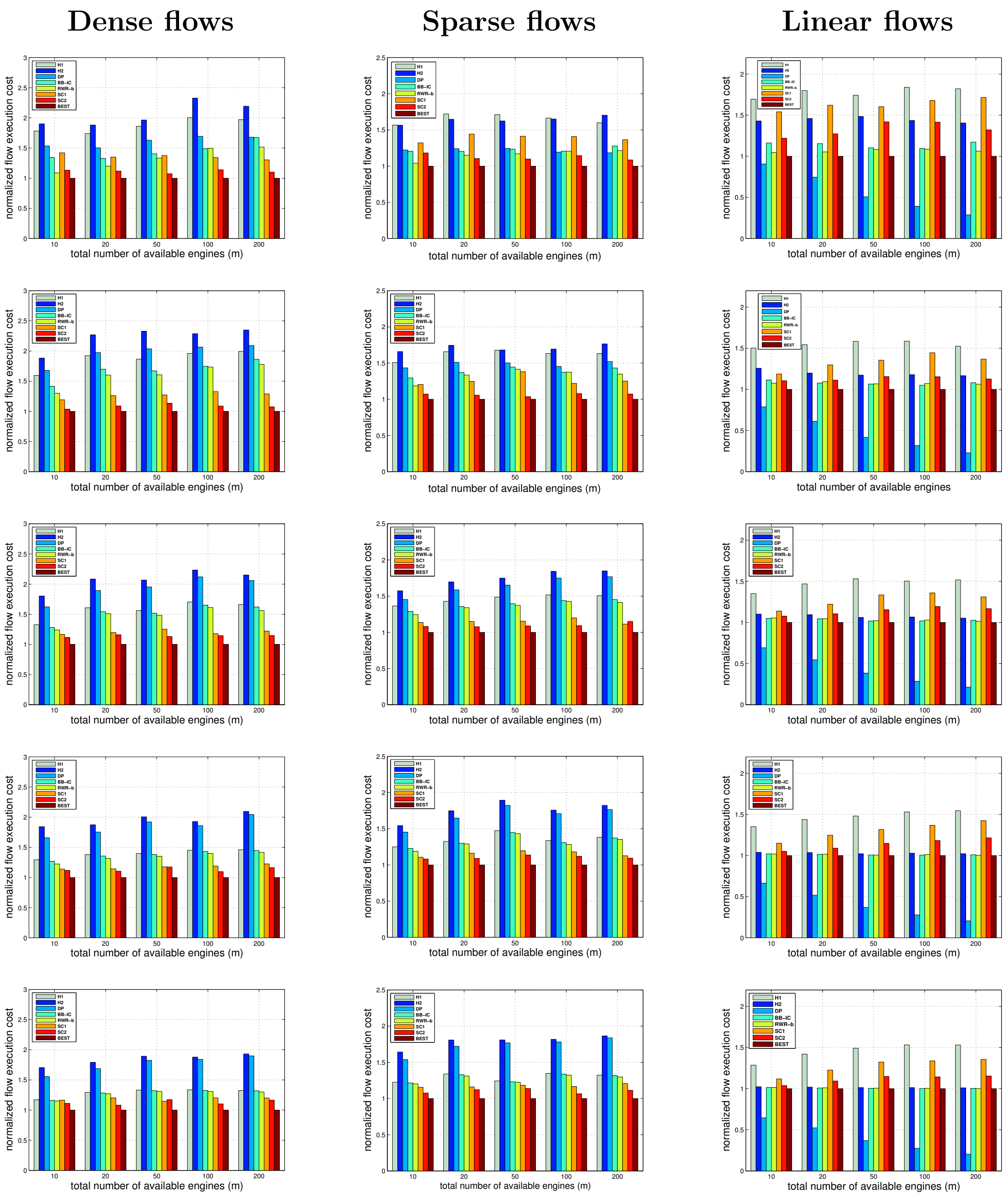
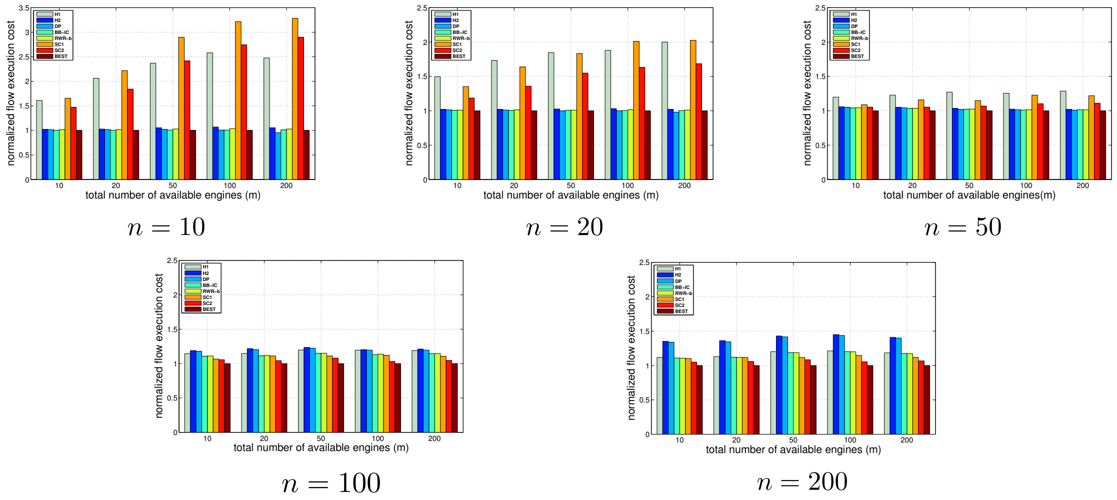


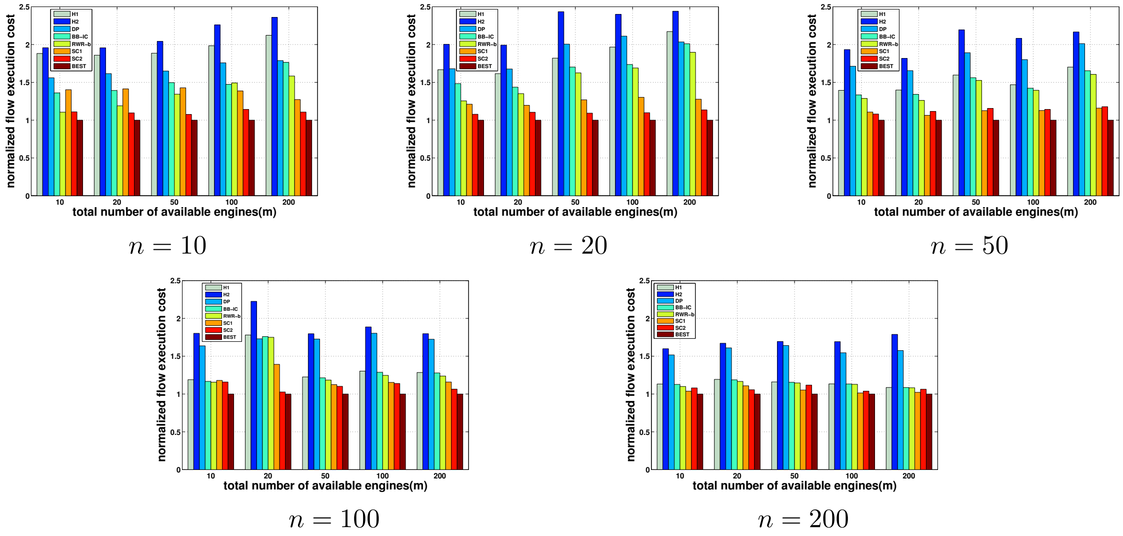
Figure 5: Performance comparison when $n = 10, 20, 50, 100$ and 200 (from top to bottom).



$n = 10$ $n = 20$ $n = 50$

$n = 100$ $n = 200$

Figure 6: Performance comparison when the inter-engine cost $\in [1, 10]$ for dense flows.



$n = 10$ $n = 20$ $n = 50$

$n = 100$ $n = 200$

Figure 7: Performance comparison when $n = 10, 20, 50, 100$ and 200 and the inter-engine cost is activity-dependent.

these experiments are presented in Figure 5. The numbers are normalized according to the execution cost yielded by *BEST* as previously.

The main observation for dense and sparse flows is that the proposed

Table 5: Maximum performance degradation of *H1* and *H2* in a single iteration compared to our solutions.

Dense flows

n / m	***10***	***20***	***50***	***100***	***200***
10	2.92	2.78	3.16	3.77	4.37
20	3.03	2.96	3.46	4.38	3.65
50	3.23	2.88	3.01	3.09	4.68
100	2.41	2.32	2.58	2.80	3.09
200	1.89	2.39	3.58	3.83	2.56

Sparse flows

n / m	***10***	***20***	***50***	***100***	***200***
10	2.10	2.67	2.24	2.31	2.80
20	2.70	2.22	2.67	3.41	2.90
50	3.15	3.01	2.67	2.62	3.10
100	2.60	3.50	2.75	2.42	2.68
200	3.27	2.68	2.42	2.25	2.24

Linear flows

n / m	***10***	***20***	***50***	***100***	***200***
10	2.15	2.70	4.36	5.64	6.99
20	1.93	2.60	3.84	4.95	6.65
50	2.09	2.47	3.51	4.29	5.84
100	1.72	2.39	3.13	4.14	5.53
200	1.82	2.10	2.94	4.02	5.40

anytime algorithms (i.e., *BB-IC*, *RWR-b*, *SC1*, *SC2* and *BEST*) consistently outperform the two simple heuristics; this is not the case for the *DP* proposal, which is proved to be optimal for linear flows and the more dense a flow is the higher the deviation of *DP*'s solution from the optimal one. Specifically, for dense flows (left column in Figure 5), when the flow size is small, the best performing simple heuristic can run on average up to 100% longer than *BEST* (for n=10 and m=100). The best performing heuristic is *H1* because it implicitly tackles edge cost minimization contrary to solutions, such as *H2* and *DP*. The relative degradation decreases but remains significant as the flow size grows. For instance, the average degradation can be up to 70% for dense flows of medium size, 45% for large flows and 33% for very large flows. Note that the maximum performance degradation in a single iteration can be much higher, as shown in the upper part of Table 5. That table presents the highest number of times the best performing heuristic cost is higher than our best performing solution, which is always *BEST* for dense and sparse flows, and *DP* for linear flows. In a single iteration, the simple heuristics' execution costs observed are up to 368% larger for medium flows and up to 283% for very large flows.

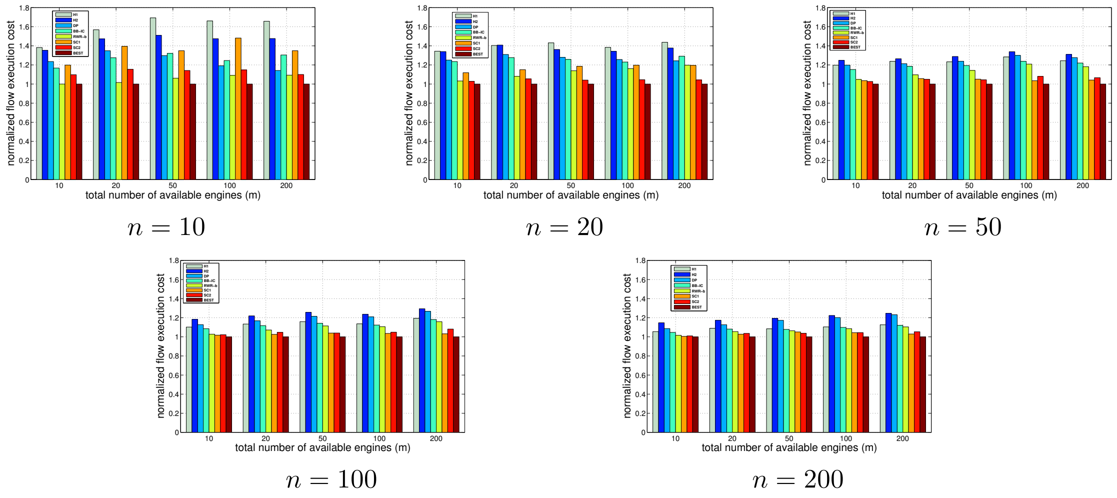
In sparse data flows, the performance improvement is lower than in dense flows, but it is still considerable and up to 66%, 51%, 47% and 34% for small, medium, large and very large flows, respectively. We always compare our best performing solution against the best performing naive heuristic. Another observation is that both in dense and sparse data flows, *H1* outperforms the other heuristic *H2*, with some exceptions for small sparse flows.

As explained in Section 3, *BEST* is a meta-heuristic, which leverages *BB-IC*, *RWR-b* and set cover solutions. In general, the set cover solutions are the ones with the highest average performance between the approaches considered by *BEST* (apart from sparse flows when $n = m = 10$). Without *BEST*, our proposal with the highest performance would be at least 10% slower. Between *BB-IC* and *RWR-b*, which are used by all *SC1*, *SC2* and *BEST*, there is no clear winner, but in the majority of the settings, *RWR-b* is superior to *BB-IC*.

As far as the linear data flows are considered, the *DP* algorithm finds the optimal solution, and as such, achieves the lowest execution times. On average, *DP* can exhibit up to 7.5 times better performance than the naive heuristics. *BB-IC* and *RWR-b* attain similar performance improvements for large and very large flows. In all cases, both the *SC1* and *SC1* algorithms are outperformed by the *BB-IC* and *RWR-b* solutions. *H2*, which does not consider edge costs, performs better than *H1*, and in some cases better than some of our proposals, such as *SC1*.

In the next experiment, we show the performance of the algorithms when the average inter-engine cost becomes an order of magnitude lower than the average activity cost. More specifically, Figure 6 depicts the execution cost of dense data flows when the inter-engine cost between engines $\in [1, 10]$. In this figure, we can see that, especially for non large flows, the naive heuristics perform very slightly worse than our solutions. This is expected since, when the inter-engine cost becomes zero, *H2* yields an optimal solution. However, even in this setting, the performance degradation for large and very large flows is significant and can reach 21%.

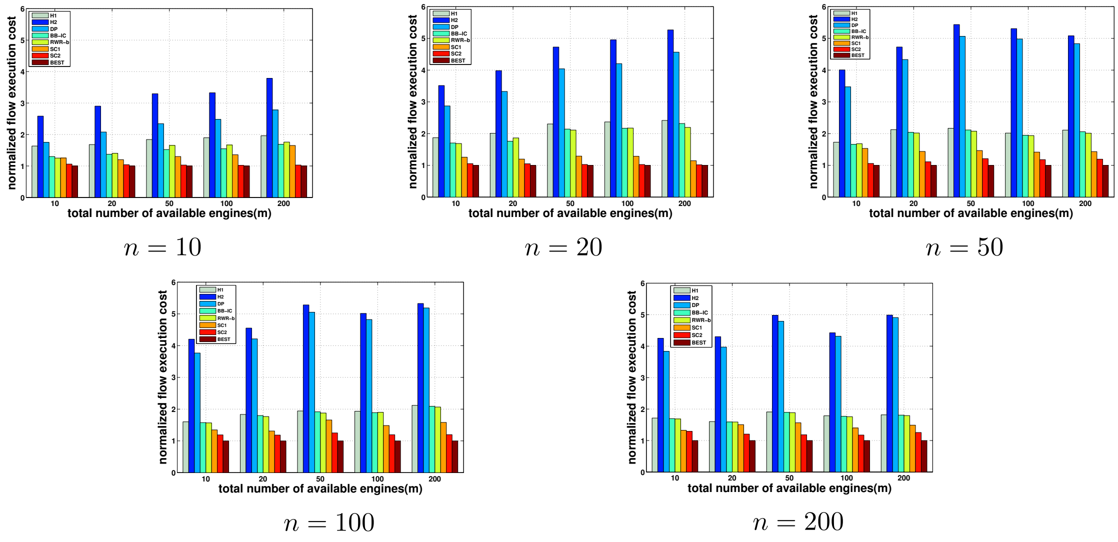
As for real flows, we relax the assumption and we consider activity-



$n = 10$ $n = 20$ $n = 50$

$n = 100$ $n = 200$

Figure 8: Performance comparison when the probability of an engine to be capable of executing an activity is 20% for dense flows.



$n = 10$ $n = 20$ $n = 50$

$n = 100$ $n = 200$

Figure 9: Performance comparison when the probability of an engine to be capable of executing an activity is 80% for dense flows.

dependent inter-engine costs as in Section 6.1.1. The results are shown in Figure 7 and confirm the conclusions that we discussed for real flows about

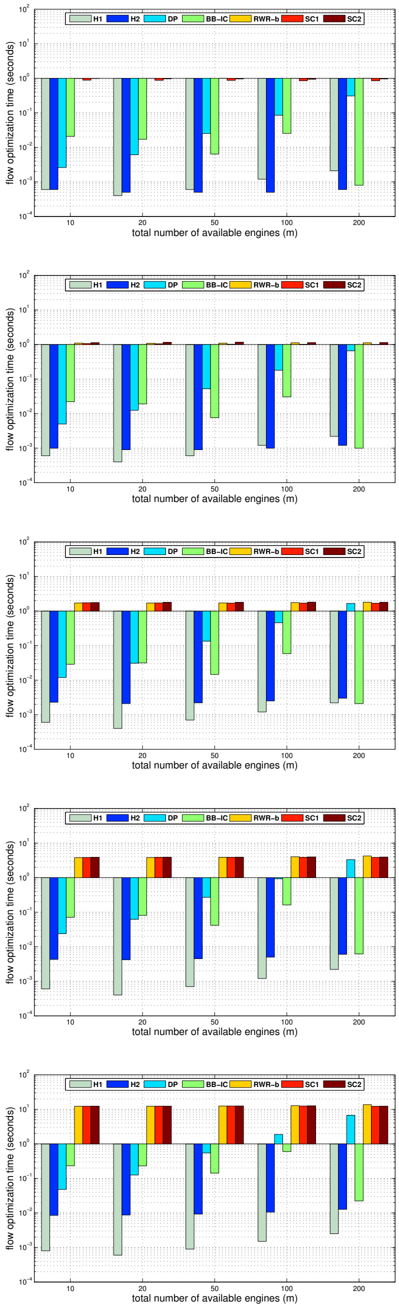


Figure 10: Decision making time for $n = 10, 20, 50, 100$ and 200 (from top to bottom).

the impact of execution engine homogeneity on the performance improvement of data flows. Specifically, we observe that the performance improvement of the data flows due to our proposals follow the same pattern for both inter-engine activity-dependent and activity-independent cases. We have observed that our solutions can be up to more than 5 times faster than the existing heuristics in isolated runs. Comparing Figure 7 against the left column of Figure 5, we see that *H1* is better than *H2* in Figure 7, but its performance against our solutions is even worse for small and medium flows. For large and very large flows, the performance gap is slightly narrower, especially for a large number of candidate execution engines. In the remainder, we will only discuss the case where the inter-engine costs are activity-independent for simplicity.

Figures 8 and 9 refer to a more and a less constrained setting, where, on average, each flow activity can run on only 20% or 80% of the engines, respectively (instead of 50%). When having 20% engine probability, the performance degradation of the naive solutions is more evident for small flows (where it can be up to 51%), but becomes smaller for very large flows (where it can be up to 12%). In the case of 80% engine probability, the performance degradation increases compared to the results of 20% or 50% engine constraints. Specifically, for small flows, the simple heuristics in the best case are 63% worse than our proposals, while, in large flows, our average performance improvements are at least 60%.

6.2.2. Decision making overhead

We show the running time of the optimization process in Figure 10. For simplicity, we discuss only dense flows, but the observations apply to all flow types. We can draw the following observations: the naive heuristics run in milliseconds for any size of flows and candidate engine sets. If the number of engines is up to 50, the *DP* and *BB-IC* algorithms run in hundreds of milliseconds. For $m = 100$, *DP* still runs in less than 1 sec, except when $n = 200$. For $m = 200$, the average time overhead of *DP* is between 0.3 secs and 6.7 secs.

RWR-b runs in 1 sec for small flows, up to 1.8 secs for medium flows; for large and very large flows, *RWR-b* does not exceed 4.2 and 13.7 secs, respectively.[3] The overhead of the set cover solutions are largely determined

[3] The overhead of RWR-b is mostly due to the estimation of the cost of each allocation

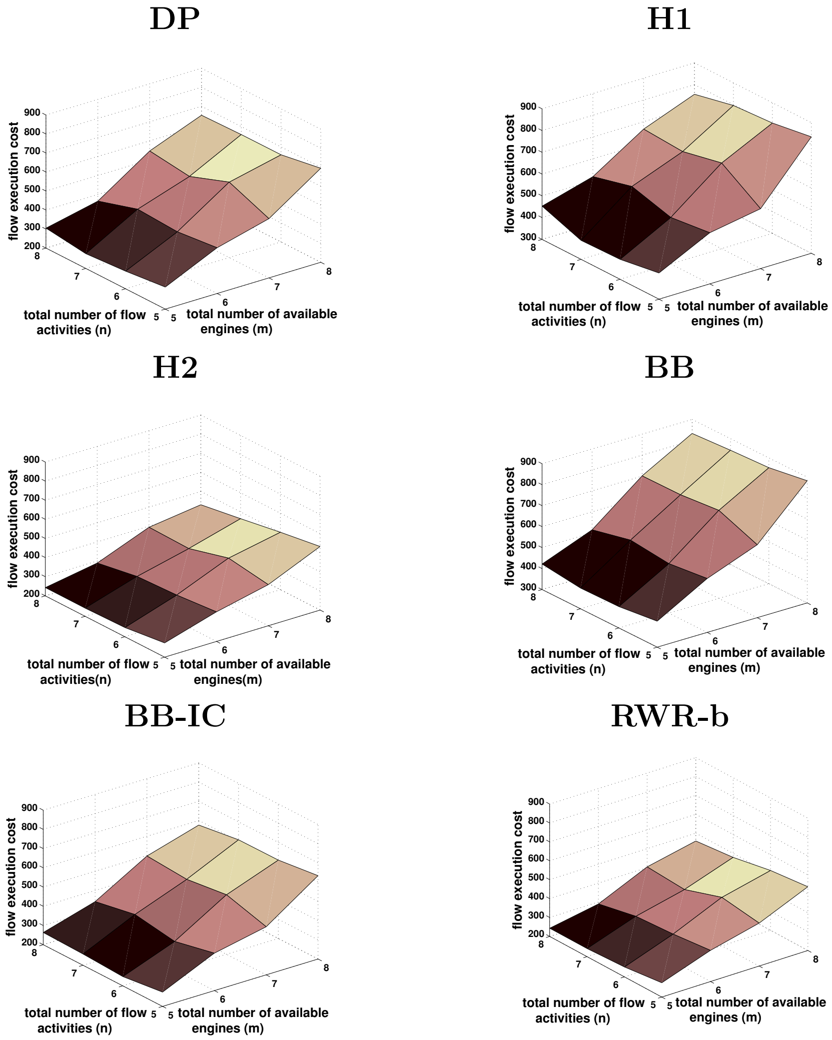


Figure 11: Performance when $m = 5, 6, 7, 8$ and $n = 5, 6, 7, 8$.

by the overhead of *RWR-b*; it is slightly smaller than that of *RWR-b* since *SC1* and *SC2* examine a smaller set of engines. Overall, the running overhead is low, which supports our claim that our proposals are practical.

plan after each random change from scratch; for large flows, more efficient cost estimation approaches that reuse previous results can be devised.

6.2.3. Accuracy

The accuracy of the algorithms can be accurately measured only if the optimal solution can be found. This can be done in reasonable time in two cases: i) when the flows are linear; and ii) when n and m are sufficiently small so that *BB* can be applied. For the first case, we can use the third column of Figure 5, which shows the average performance of the algorithms. The accuracy of the anytime algorithms degrades as n and m increase. In addition, the bottom part of Table 5 shows the maximum performance degradation observed for the two simple heuristics.

We also check the accuracy of the algorithms for very small dense flows, where $n, m = 5, 6, 7, 8$. For such flows, *RWR-b* is remarkably accurate, and, on average, it is within 2% of the optimal solution provided by *BB*. The next more accurate algorithm is *BB-IC*, the average degradation of which is 15%. *DP* is 29% slower, whereas, the best performing naive heuristic, *H1* incurs 63% higher execution costs.

6.2.4. Random Walk Flavours

In Section 3, we discussed a set of random walk flavours and here we explain why we used only *RWR-b* in the previous experiments. We present the comparison of the flavours only for a representative setting: dense data flows with $n = 50$, $m = 50$. The results of this experiment are presented in Figure 12. *RWR-b* algorithm has the best performance compared to *RW* and *RWR-r*, although the difference of performance and time overhead between *RWR-r* and *RWR-b* is negligible. Nevertheless, the optimization time of the simple *RW* is much lower than 1 sec for activities at the expense of approximately 4.2% of performance degradation.

For the same experimental setting, we investigate the impact of the random walk length for *RWR-b*. We evaluate walk lengths of $10^3, 10^4$ and 10^5, as shown in the middle row of Figure 12. The main observation is that as we increase the length of the walks, the execution cost of the algorithm is slightly increased too, whereas the optimization time increases proportionally to the length of the walk. For large lengths the optimization overhead is on the orders of minutes without significant performance benefits. Finally, in the last set experiment, we evaluate the impact of the number of restarts ($r = 10, 20, 30, 40, 50$), as shown in the bottom row of Figure 12. According to the results, the impact of restarts do not significantly affect the performance: going from 10 restarts to 50 yields approximately 2% of improvement.

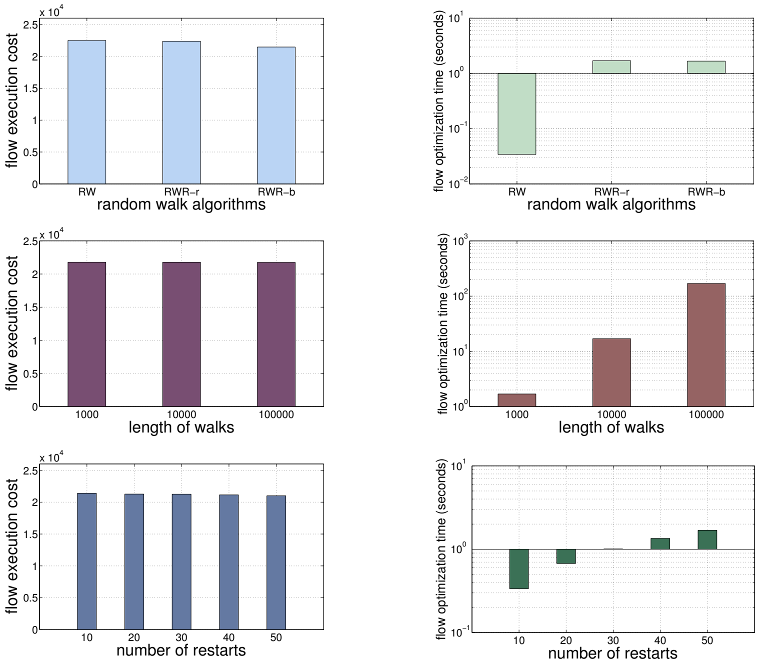


Figure 12: 1st row: performance and optimization time of random walk flavours. 2nd and 3rd row: impact of random walk length and restarts on *RWR-b*, respectively.

6.3. Summary of Lessons Learnt and Discussion

The real-world flows in scientific scenarios tend to be sparse and either close to linear ones, or comprising many linear subflows. This implies that the contribution of the inter-engine cost to the flow execution cost is less significant compared to arbitrarily random flows. In scenarios, where there are many linear subflows, *DP* exhibits clearly better performance; otherwise *BEST* yields the lowest execution times. Our solutions can also yield significant benefits when there is no obligation to switch between engines (in the sense that an engine can run the complete flow) and the inter-engine costs are an order of magnitude higher than the processing costs.

From the experiments with the synthetic data, we can draw the following conclusions for random flows: (i) our solutions can efficiently handle even very large flows and outperform naive solutions; (ii) we can declare clear winners for different types of flows: for dense and sparse flows, *BEST* is the superior algorithm; for linear flows, *DP* is optimal; (iii) the performance improvements of our proposals compared to naive solutions are significant in every type of flows; and (iv) the running time of the decision making process completes in less than a second in most of the cases, which supports our claim that our proposals are practical.

Note that for random flows, *BEST* is a practical and efficient solution, and its efficiency is largely due to the set cover algorithms. For those algorithms, several additional flavours can be devised; for example to select engines according to their average inter-engine costs. Such flavours may support better specific scenarios, e.g., flow types where some tasks play the role of a hub with high degree of incoming and outgoing edges. In this work, we mostly focus on generic flows; the development of additional flavours tailored to specific flow structures is out of our scope.

7. Related Work

The closest proposal to our work appears in [4, 3], where the authors also deal with the complexity of flows in multi-engine environments and present a concrete workflow enactment system that supports hybrid flow execution. Apart from the system presentation, in [4], an exhaustive approach for allocating the activities of a flow to different execution engines is proposed in order to meet multiple-objectives, such as performance and fault-tolerance. In addition, heuristic techniques are presented for pruning the search space; those heuristics are equivalent to *H1* and *H2*. In our work, we improve upon such allocation schemes, and, through our evaluation, we show that our approaches are both scalable and significantly better than simple heuristics, when performance is the single optimization criterion.

[15] introduces an ant colony optimization algorithm that selects service instantiations between multiple candidates, in a setting where the flows mainly consist of a series of remote service invocations. In our work, we do not employ such type of algorithms, because their optimization overhead is at least two orders of magnitude higher (see indicative running times in [16]).

A state-of-the-art approach to flow scheduling is presented in [17, 13]. Specifically, a set of optimization algorithms based on deadline and time constraints was analyzed for scheduling flows. If we consider to adapt these methodologies in order to fit in our problem keeping only the allocation part regardless of deadlines, we will come to the conclusion that these methodologies are reduced to the simple heuristics presented in Section 2; more specifically the allocation part is reduced to *H2*, to which our proposals are shown to be superior. Another family of proposals aim at finding allocations of flow nodes to processors within a cluster, when the processors are homogeneous. Apart from that difference, which renders them inapplicable to our setting, typical assumptions are that there is no notion of inter-engine cost and there are no constraints with regards to the capabilities of an engine to execute a specific flow activity. Examples of such allocation approaches are described in [18, 19, 20].

For completeness, we briefly discuss additional aspects of flow optimization, which differ from our problem setting. [21] discusses optimal time schedules given a fixed allocation of activities to engines. Scheduling issues are also considered in works such as [22], which exploit existing systems, e.g. Pegasus, for task mapping procedure and [23], in which deadline constraints are taken into account. The proposals of [24, 25, 26] focus on methodologies of re-ordering and/or merging flow activities in order to yield improved performance, while keeping the flow semantics. In [27], flow activities are transformed in order to benefit from underlying data management infrastructures. [28] and [14] discuss optimization of data flows according to multiple objectives without considering engine allocation issues. In [29], a data oriented method for workflow optimization is proposed in order to minimize execution cost. This method is based on the fact that data may be shared across several functions, and, as such, workflow performance stands to benefit from optimizations in the form of incorporating a shared database to handle common data-oriented tasks. Another proposal of flow optimization is presented in [30] based on soft deadline rescheduling in order to deal with the problem of fault tolerance in flow executions. In [31], an auction-based scheduling methodology for multi-objective flow optimization is presented; in our setting, choosing the most inexpensive engine is similar to the policy of the naive heuristic *H2*. Also, a methodology for minimizing the performance fluctuations that might occur by the resource diversity is proposed in [32]. Their proposal focuses on the delay correction during task execution. All these optimization aspects are orthogonal to our research.

The optimization of flows bears also similarities to distributed query optimization [33] and optimization of queries with user-defined functions [34]; however, in those problems, the focus is on the shape of the query plan and the ordering of the distributed operators (e.g., [34, 35]) instead of deciding the mapping of a plan node to a specific engine. Other issues that differentiate query and flow optimization include the definition of the semantics of flow nodes, algebraic re-writing of flow plans and respecting inter-task dependencies.

8. Conclusions and Future Work

In this work, we investigate the problem of allocating nodes of data-intensive flows to concrete executing engines. We prove that this problem is $\mathcal{NP}$-hard and cannot be approximated within a small constant. Due to the problem complexity, to date this problem is addressed using naive heuristics. In this work, we show that we can do significantly better without much overhead. We propose an optimal polynomial time dynamic programming solution for the specific case of flows that are linear, i.e., a chain of activities. Furthermore, we propose anytime algorithms that can handle any type and size of flows. With the help of our thorough experimentation, we declare clear winners depending on the type of the flow. Our proposals are capable of yielding solutions that are significantly better than naive approaches; actually, in real-world flows and conditions, they outperform those naive approaches by a factor of up to three. Our proposals are also easy to implement and light-weight.

This work aims to propose fast algorithms for engine selection and focuses on the generic properties of the solutions. Apart from devising tailored solutions for each type of real-world flow, in the future, it is interesting to investigate solutions without the constraint of finding an allocation in a limited time period. Two further avenues for extending this work are to consider multiple objectives (e.g., both total time and makespan) and consider the impact of co-allocating activities to the same engine on the costs of those activities.

Acknowledgments

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program “Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

We are also grateful to Alkis Simitsis for his insightful comments in early drafts of this work.

References

[1] T. Hey, S. Tansley, K. Tolle, The Fourth Paradigm: Data-Intensive Scientific Discovery, 2009.

[2] S. Chaudhuri, U. Dayal, V. Narasayya, An overview of business intelligence technology, Commun. ACM 54 (2011) 88–98.

[3] A. Simitsis, K. Wilkinson, U. Dayal, M. Hsu, Hfms: Managing the lifecycle and complexity of hybrid analytic data flows, in: ICDE, 2013.

[4] A. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal, Optimizing analytic data flows for multiple execution engines, in: SIGMOD Conference, 2012, pp. 829–840.

[5] P. Jovanovic, A. Simitsis, K. Wilkinson, Engine independence for logical analytic flows, in: Proc. ICDE, 2014.

[6] B. Huang, S. Babu, J. Yang, Cumulon: optimizing statistical data analysis in the cloud, in: SIGMOD Conference, 2013, pp. 1–12.

[7] G. Teodoro, T. D. R. Hartley, Ü. V. Çatalyürek andRenato Ferreira, Optimizing dataflow applications on heterogeneous environments, Cluster Computing 15 (2) (2012) 125–144.

[8] A. D. Popescu, D. Dash, V. Kantere, A. Ailamaki, Adaptive query execution for data management in the cloud, in: CloudDB, 2010, pp. 17–24.

[9] R. Graham, E. Lawler, J. Lenstra, A. Kan, Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey, in: Discrete Optimization II Proc. of the Advanced Research Inst. on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium, 1979, pp. 287–326.

[10] H. Hoogeveen, P. Schuurman, G. Woeginger, Non-approximability results for scheduling problems with minsum criteria, in: Integer Programming and Combinatorial Optimization, 1998, pp. 353–366.

[11] J. A. Hoogeveen, J. K. Lenstra, B. Veltman, Three, four, five, six, or the complexity of scheduling with communication delays, Oper. Res. Lett. 16 (3) (1994) 129–137.

[12] G. Juve, A. L. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, Future Generation Comp. Syst. 29 (3) (2013) 682–692.

[13] S. Abrishami, M. Naghibzadeh, D. H. Epema, Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds, Future Generation Computer Systems 29 (1) (2013) 158 – 169.

[14] A. Simitsis, K. Wilkinson, U. Dayal, M. Castellanos, Optimizing etl workflows for fault-tolerance, in: ICDE, 2010, pp. 385–396.

[15] W. N. Chen, J. Zhang, An ant colony optimization approach to a grid workflow scheduling problem with various qos requirements, IEEE Transactions on Systems Man and Cybernetics Part C Applications and Reviews 39 (1) (2009) 29–43.

[16] Z. Zhang, X. Li, Based on tsp problem the research of improved ant colony algorithms, in: Electrical Engineering and Control, Vol. 98 of Lecture Notes in Electrical Engineering, 2011, pp. 827–833.

[17] S. Abrishami, M. Naghibzadeh, D. H. J. Epema, Cost-driven scheduling of grid workflows using partial critical paths, IEEE Trans. Parallel Distrib. Syst. 23 (8) (2012) 1400–1414.

[18] E. Schikuta, H. Wanek, I. Ul Haq, Grid workflow optimization regarding dynamically changing resources and conditions, Concurr. Comput. : Pract. Exper. 20 (2008) 1837–1849.

[19] M. Rahman, M. R. Hassan, R. Ranjan, R. Buyya, Adaptive workflow scheduling for dynamic grid and cloud computing environment, Concurrency and Computation: Practice and Experience 25 (13) (2013) 1816–1842.

[20] R. Duan, R. Prodan, T. Fahringer, Performance and cost optimization for multiple large-scale grid workflow applications, in: Proc. of the ACM/IEEE Conf. on Supercomputing, 2007, pp. 1–12.

[21] K. Agrawal, A. Benoit, L. Magnan, Y. Robert, Scheduling algorithms for linear workflow optimization, in: IEEE IPDPS'2010, pp. 1–12.

[22] V. S. Kumar, P. Sadayappan, G. Mehta, K. Vahi, E. Deelman, V. Ratnakar, J. Kim, Y. Gil, M. Hall, T. Kurc, J. Saltz, An integrated framework for parameter-based optimization of scientific workflows, in: HPDC, ACM, 2009, pp. 177–186.

[23] Y. Yuan, X. Li, Q. Wang, X. Zhu, Deadline division-based heuristic for cost optimization in workflow scheduling, Inf. Sci. 179 (2009) 2562–2575.

[24] A. Simitsis, P. Vassiliadis, T. K. Sellis, State-space optimization of etl workflows, IEEE Trans. Knowl. Data Eng. 17 (10) (2005) 1404–1419.

[25] G. Kougka, A. Gounaris, Declarative expression and optimization of data-intensive flows, in: DaWaK, 2013, pp. 13–25.

[26] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, K. Tzoumas, Opening the black boxes in data flow optimization, PVLDB 5 (11) (2012) 1256–1267.

[27] M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, T. Kraft, An approach to optimize data processing in business processes, in: VLDB, 2007, pp. 615–626.

[28] U. Dayal, M. Castellanos, A. Simitsis, K. Wilkinson, Data integration flows for business intelligence, in: Proc. of EDBT, 2009, pp. 1–11.

[29] R. Minglun, Z. Weidong, Y. Shanlin, Data oriented analysis of workflow optimization, in: Proc. of the 3rd World Congress on Intelligent Control and Automation, 2000 - Volume 4, IEEE Computer Society, 2000, pp. 2564 – 2566.

[30] K. Plankensteiner, R. Prodan, Meeting soft deadlines in scientific workflows using resubmission impact, Parallel and Distributed Systems, IEEE Transactions on 23 (5) (2012) 890–901.

[31] H. Fard, R. Prodan, T. Fahringer, A truthful dynamic workflow scheduling mechanism for commercial multicloud environments, Parallel and Distributed Systems, IEEE Transactions on 24 (6) (2013) 1203–1212.

[32] R. N. Calheiros, R. Buyya, Meeting deadlines of scientific workflows in public clouds with tasks replication, IEEE Transactions on Parallel and Distributed Systems 99 (PrePrints) (2013) 1.

[33] D. Kossmann, The state of the art in distributed query processing., ACM Comput. Surv. 32 (4) (2000) 422–469.

[34] S. Chaudhuri, K. Shim, Optimization of queries with user-defined predicates., ACM Trans. Database Syst. 24 (2) (1999) 177–228.

[35] U. Srivastava, K. Munagala, J. Widom, R. Motwani, Query optimization over web services., in: VLDB, 2006, pp. 355–366.