Modeling Data Flow Execution in a Parallel Environment

Georgia Kougka[1], Anastasios Gounaris[1], and Ulf Leser[2]

[1] Department of Informatics
Aristotle University of Thessaloniki, Greece
{georkoug,gounaria}@csd.auth.gr
[2] Institute for Computer Science
Humboldt-Universität zu Berlin, Germany
leser@informatik.hu-berlin.de

Abstract. Although the modern data flows are executed in parallel and distributed environments, e.g. on a multi-core machine or on the cloud, current cost models, e.g., those considered by state-of-the-art data flow optimization techniques, do not accurately reflect the *response time* of real data flow execution in these execution environments. This is mainly due to the fact that the impact of parallelism, and more specifically, the impact of concurrent task execution on the running time is not adequately modeled. In this work, we propose a cost modeling solution that aims to accurately reflect the *response time* of a data flow that is executed in parallel. We focus on the single multi-core machine environment provided by modern business intelligence tools, such as Pentaho Kettle, but our approach can be extended to massively parallel and distributed settings. The distinctive features of our proposal is that we model both time overlaps and the impact of concurrency on task running times in a combined manner; the latter is appropriately quantified and its significance is exemplified.

1 Introduction

Nowadays, data flows constitute an integral part of data analysis. The modern data flows are complex and executed in parallel systems, such as multi-core machines or clusters employing a wide range of diverse platforms platforms like Pentaho Kettle[3], Spark[4] and Stratosphere[5] to name a few. These platforms operate in a manner that involves significant time overlapping and interplay between the constituent tasks in a flow. However, there are no cost models that provide analytic formulas for estimating the *response time (wallclock time)* of a flow in such platforms. Cost models, apart from being useful in their own right, are encapsulated in cost-based optimizers; currently, cost-based optimization solutions for task ordering in data flows employ simple cost models that may

[3] http://community.pentaho.com/projects/data-integration
[4] http://spark.apache.org/
[5] http://stratosphere.eu/

not capture the flow execution running time accurately, as shown in this work. This results in an execution cost computation that may deviate from the real execution time, and the corresponding optimizations may not be reflected on response time.

Typically, cost models rely on the existence of appropriate metadata regarding each task, which are combined using simple algebraic formulas with the sum and max operations. Most often, task metadata consider the cost of each task, which is commensurate with the task running time if executed in a stand-alone manner. The main challenges in devising a cost model for running time that is appropriate for modern data flow execution stem from the following factors: (i) many tasks are executed in parallel employing all three main forms of parallelism, namely, partitioned, pipelined and independent, and the resulting time overlaps, which entail that certain task executions do not contribute to the overall running time, need to be reflected in the cost model; and (ii) computation resources are shared among multiple tasks, and the concurrent execution of tasks using the same resource pool impacts on their execution costs.

In this work, we focus on devising a cost model that can be used to estimate the response time, when the dataflows are executed in parallel and distributed execution environments. To this end, we extend existing cost modeling techniques that tend to consider time overlapping (e.g., [11, 4, 18, 1, 2, 16]) but not the interplay between task costs. In order to achieve this, we propose a solution in which the cost of each task is weighted according to the number of concurrent tasks taking into account constraints of execution machines, such as capacity in terms of number of cores. More specifically, we initially focus on a single multicore machine environment, such as Pentaho Data Integration (PDI, aka Kettle), and the contribution is as follows:

1. We explain and provide experimental evidence on why the existing cost models provide estimates that widely deviate from the real execution time of modern workflows.
2. We propose a model that not only considers overlapping task executions but also quantifies the correlation between task costs due to concurrent allocation to the same processing unit. The model is execution engine software- and data flow type-independent.
3. We show how our model applies to example flows in PDI, where inaccuracies of up to 50% are observed if the impact of concurrency is not considered.

In the remainder of this section we provide background on flow parallelization, the assumptions regarding the execution environment that we consider and a discussion about the inadequacy of cost models employed in data flow optimization. We continue the discussion of related work in Sec. 2. In Sec. 3 we introduce the notation. Our modeling proposal is presented in detail in Sec. 4 and we conclude in Sec. 5.

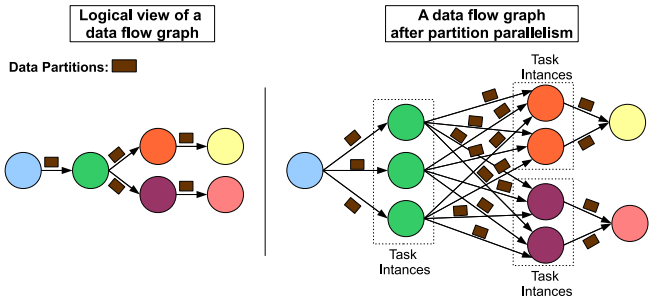

Fig. 1. A data flow graph before and after partitioned parallelism; circles with the same color correspond to partitioned instances of the same flow task.

1.1 Parallelizing Data Flows

The parallel execution of a data flow exploits three types of parallelism, namely *inter-operator*, *intra-operator* and *independent* parallelism. Here, we use the terms *task*, *activity* and *operator* interchangeably. These types of parallelization are well-known in query optimization [6], and used to decrease the *response time* of a data flow execution.

The *intra-operation parallelism* considers the parallelization of a single task of a data flow. This type of parallelization is defined by the instantiation of a logical task as a set of multiple physical instances, each operating on a different portion of data, i.e. each task can be executed by several processors and data is partitioned. An example of partitioned parallelism is depicted in Figure 1. There is a set of different methods of partitioning, such as round-robin and hash-partitioning. In this work, we assume that the degree of *intra-operation* parallelism is fixed; e.g., in Figure 1, the degree for the green task is set to 3.

The *independent* parallelism is achieved when the tasks of the same data flow may be executed in parallel because there are no dependency constraints or communication between them. An example is the two branches at the right part of the flow in Figure 1(left).

The *pipeline* parallelism takes place when multiple tasks are executed in parallel with a producer-consumer link and each producer sends part of its output, which is a collection of records, as soon as this output is produced without waiting the processing of its input to complete and, therefore, the whole output to be produced.

In this work, we present a cost model for data flow execution plans that accurately estimates the *response time* considering the *pipeline parallelism* and *independent* types of parallelism, which are relevant to a single machine Kettle execution. However, it is straightforward to extend our work to cover partitioned parallelism as well, as briefly discussed in Sec. 4.

1.2 Assumptions regarding a single multi-core machine execution environment

Our main assumptions are summarized as follows:

- Data flows utilize all the available machine cores. The number of cores depends on the execution machine.
- The execution machine is exclusively dedicated to the data flow execution. I.e., we assume that an execution machine executes only one data flow and the execution of the next flow can be started only after the completion of the previous flow. So, the available machine executes tasks and stores data for a single data flow at a time.
- Multiple tasks of a data flow are executed simultaneously. Each task spawns a separate thread, running on a core decided by the underlying operating system scheduler. Obviously, if two task threads share the same core, they are executed concurrently but not simultaneously.
- The execution engine exploits pipeline and independent parallelism to the largest possible extent; i.e., the default engine configuration regarding task execution operates in a mode, according to which flow tasks are aggressively scheduled as soon as their input data is available.

The assumptions above hold also for massive parallel settings. The main difference is that, in massive parallelism settings, partitioned parallelism typically applies.

1.3 Motivation for devising a new cost model

A main application of cost models is in cost-based optimization. One of forms of data flow optimization that has been largely explored in the data management literature is task re-ordering. Taking this type of optimization as case study, we can observe from the survey in [9] that the corresponding techniques target one of the following optimization objectives:

1. *Sum Cost Metric of the Full plan (SCM-F)*: minimize the sum of the task and communication costs of a data flow [7, 13, 20, 10, 15, 8, 16].
2. *Sum Cost Metric of the Critical Path (SCM-CP)*: minimize the sum of the task and communication costs along the flow's critical path [1, 2].
3. *Bottleneck*: minimize the highest task cost [18, 1, 2, 16].
4. *Throughput*: maximize the throughput (number of records processed per time unit) [5].

The first three metrics and the associated cost models can capture the *response time* under specific assumptions only. The *response time* represents the wall-clock time from the beginning until the end of the flow execution. *SCM-F* defines the *response time* when the tasks of a data flow are executed sequentially; for example when all tasks are blocking. Another case is when tasks are pipelined but are executed on the same CPU core (processor). In that case, the *SCM-F*

may serve as a good approximation of the *response time*. *SCM-CP* reflects the *response time* when the data flow branches are executed independently and the tasks of each branch are executed sequentially. Finally, *bottleneck* represents the *response time* when all the tasks of the flow are executed in a pipelined manner and each task is executed on a different processor assuming enough cores are available.

So, why do we need another cost model? PDI, Flink, Spark and similar environments aggressively employ pipeline parallelism potentially on multiple processors. Consequently, the *SCM-F* and *SCM-CP* cost metrics do not correspond to the *response time* of the flow execution. *Bottleneck* cost metric is not appropriate either. This is because there are pipelined tasks that are executed on the same processor, but also there are tasks that are blocking, e.g., sort. So, for estimating *response time*, we need to employ a cost metric that explicitly considers parallelism and the corresponding overlaps in task execution. A cost model that computes this cost metric is therefore needed.

Furthermore, a more accurate cost model for describing the response time is significant in its own right even when not used to drive optimizations. It allows us to better understand the flow execution and provides better insights into the details involved. Moreover, as will be shown in the subsequent section, merely considering time overlaps does not suffice, because the task costs are correlated during concurrent task execution.

2 Other Related Work

The main limitation of existing cost models is that, even if they consider overlapped execution, they assume that the cost of each task remains fixed independently of whether other tasks are executing concurrently sharing CPU, memory and other resources. Examples that fall in this category are the work in [11], which targets a cloud environment for scientific workflow execution, and in [4]. The cost model in the latter considers that the flow is represented by a graph with multiple branches (or paths), where the tasks in each path are executed sequentially and multiple branches are executed in parallel. In constrast, we cover more generic cases.

Additionally, several proposals based on the traditional cost models have been presented in order to capture the execution of MapReduce jobs. For example, a performance model that estimates the job completion time is presented for ARIA Framework in [19]; this solution accounts for the fact that the map and reduce phases are executed sequentially employing partitioned parallelism but do not take into account the effect of allocation of multiple map/reduce tasks on the same core. The same rationale is also adapted by cost models introduced in proposals, such as [21] and [17]. Nevertheless, an interesting feature of these models is that they model the real-world phenomenon of imbalanced task partition running times. In the MapReduce setting, the authors in [14] propose the Produce-Transporter-Consumer model to define the parallel execution of MapReduce tasks. The key idea is to provide a cost model that describes the

tradeoffs of four factors (namely, map and reduce waves, output compression of map tasks and copy speed during data shuffling) considering any overlaps. As previously, the impact of concurrency is neglected. Other works for MapReduce, such as [3], suffer from the same limitations.

3 Preliminaries

A data flow is represented as a *Directed Acyclic Graph (DAG)*, where each vertex corresponds to a task of the flow and the edges between vertices represent the communication among tasks (intermediate data shipping among tasks). In data flows, the exchange of data between tasks is explicitly represented through edges. We assume that the data flows can have multiple *sources* and multiple *sinks*. A *source* of a data flow corresponds to a task with no incoming edges, while a *sink* corresponds to a task with no outgoing edges. The main notation and assumptions are as follows:

Let $G = (V, E)$ be a *Directed Acyclic Graph (DAG)*, where $V = v_1, v_2, ..., v_n$ denotes the vertices of the graph (data flow tasks) and E represents the edges (flow of data among the tasks); n is the total number of vertices. Each vertex corresponds to a data flow task and is responsible for one or both of the following: (i) reading or storing data, and (ii) manipulating data. The tasks of a data flow may be complex data analysis tasks, but may also execute traditional relational operations, such as union and join. Each edge equals to an ordered pair (v_j, v_k), which means that task v_j sends data to task v_k.

Each data flow is characterized by the following metadata:

- *Cost* (c_i), which applies to each task. The c_i corresponds to the cost for processing all the input records that the v_i task receives taking into consideration the required CPU cycles and disk I/Os. In distributed systems, the cost of network traffic needs to be considered as well, and may be the most important factor. An essentially similar consideration is c_i to denote the cost per single input record; in that case the *selectivity* information of all tasks is needed in order to derive the task cost for its entire input.
- *Communication Cost* ($cc_{i \to j}$), which may apply to edges. The communication cost of data shipping between the v_i and v_j depends on either the forward local pipelined data transfer between tasks or the data shuffling between parallel instances of the same data flow. It does not include any communication-related cost included in c_i; it includes only the cost that is dependent on both v_i and v_j rather than only on v_i.
- *Parallelism Type of Task* (pt_i), which describes the type of parallelism of a task i, when the task is executed. More specifically, the parallelism type characterizes if a data flow task is executed in a pipelined, denoted as p or no pipelined manner (*blocking* task), denoted as *np*. A *blocking* task requires all the tuples of the input data in order to start producing results; i.e., the parallelism type of a task reflects the way a task process the input data and produces its output.

4 Our cost model

First, we describe the main formula template of our model, then we explain how it applies to a single-machine setting and finally, we generalize to distributed settings.

4.1 A Generalized Cost Model for Response Time

We define the following cost model for estimating the response time:

$$\textit{Response Time (RT)} = \sum z_i w^c c_i + \sum z_{ij} w^{cc} cc_{i \to j} \tag{1}$$

where variable $z_i = \{0, 1\}$ is binary and defined as 1 only for tasks that determine the RT. The c_i factor denotes the cost of the i^{th} task, where $i = \{1, \ldots, n\}$. The w^c and w^{cc} weights cover a set of different factors that are responsible for the increase/decrease of RT during the task execution and communication between two tasks (data shipping), respectively. The z variables capture the time overlapping of different tasks, whereas w^c and w^{cc} quantify the impact of the execution of one task on all the other tasks that are concurrently executed, i.e., they capture the correlation between the execution of multiple concurrent tasks.

In general, the weights aim to abstract the impact of multi-threading in a single metric. Multi-threading may lead to performance overhead due to several factors, such as the *context switching* between threads, as the flow tasks are executed concurrently and need to switch from one thread to another multiple times. An additional factor for response time increase is due to the *locks* that temporarily restrict tasks sharing memory to write to the same memory location. Finally, the most significant factor in the terms of affecting the response time is the *contention* that captures the interference of the multiple interactions of each data flow task with memory and disk. Specifically, when there are multiple requests to memory, this may result in exceeding memory bandwidth and consequently, to RT increase. Finally, allocating and scheduling threads incurs some overhead, which, however, is negligible in most cases.

Nevertheless, multi-threading execution leads to execution cost improvement because of the parallel task execution. So, we may observe RT minimization, when all or more of the available cores are exploited by the data flow tasks and one copy of data is used by multiple threads at the same time. Also, the delays occurred by transferring data from memory and disk are overlapped by the task execution, when the number of tasks is higher than the available execution units.

The cost model in Eq. (1) generalizes the traditional ones. For example, based on the proposed formula, if we consider w^c and w^{cc} set to 1 and that all the tasks have $z_i = 1$, then the cost model actually corresponds to *SCM-F* and defines the RT under the specific assumptions discussed previously. If only the tasks that belong to the critical path have $z_i = 1$, then the cost model corresponds to *SCM-CP*. Similarly, if we want to consider the bottleneck cost metric, we set $z_i = 1$ for the most expensive task and $z_i = 0$ for all the other tasks.

4.2 Models without considering the communication cost

Firstly, we examine simple flows and we gradually extend our observations to larger and more complicated ones. In all cases, given that we target single-machine environments, it is reasonable to consider that the communication cost $cc_{i \to j}$ is set to 0.

A linear flow with a single pipelined segment of n tasks

A pipeline segment is defined by a sequence of n tasks in a chain, where the first task is a child of either a source or a blocking task, and the last task is either a sink or a blocking task; additionally, the tasks in between are all of p type. Also, pipeline segments do not overlap with regards to the vertices they cover.The key point of our approach is to account for the fact that there is non-negligible interference between tasks, captured by the variable α. Let us suppose that our machine has m cores. In the case where $n \leq m$, each task thread can execute on a separate core exclusively. The cost model that estimates the *response time (RT)* of a data flow execution is defined as follows, which aims to capture the fact that the running times of tasks overlap. So, we set (i) $z_i = 0$ for all tasks, apart from the task with the maximum cost, for which z is set to 1, since it determines the RT; and (ii) $w^c = \alpha$:

$$\textit{Response Time (RT)} = \alpha \max\{c_1, ..., c_n\} \tag{2}$$

Let us consider now the case where $n > m$ and the task threads need to share the available cores in order to be executed. In this case, each core may execute more than one task and the RT is determined by all the flow tasks, so $z_i = 1$ for all the flow tasks. An exception is when there is a single task with cost higher than the sum of all the other costs (similarly to the modeling in [19]):

$$\textit{Response Time (RT)} = \alpha \max\{\max\{c_1, .., c_n\}, \frac{\sum\{c_1, ..., c_n\}}{m}\} \tag{3}$$

In Eq. (3), w^c equals either to α, as in Eq. (2), or to α/m.

Experiments in PDI

In the following, we present a set of experiments that we conducted in order to understand the role of α in RT estimation. We consider synthetic flows in PDI with $n = 1, \ldots, 26$ tasks and an additional source task. The input ranges from 2.4M to 21.8M records. Two machines are used, with (i) a 4-core/4-thread i5 processor; and (ii) a 4-core/8-thread i7 processor, respectively. Finally, the task types are two, either homogenous or heterogeneous. In the former case, all tasks have the same cost (denoted as *equal*). In the latter case (denoted as *mixed*), half of the tasks have the same cost as in the *equal* case, and the other tasks have another cost, which is lower by an order of magnitude. All the tasks apply filters to the input data, but these filters are not selective in the sense that they produce the same data that they receive. The data input is according to the TPC-DI Benchmark[12] and we consider records taken from the implementation in `http://www.`

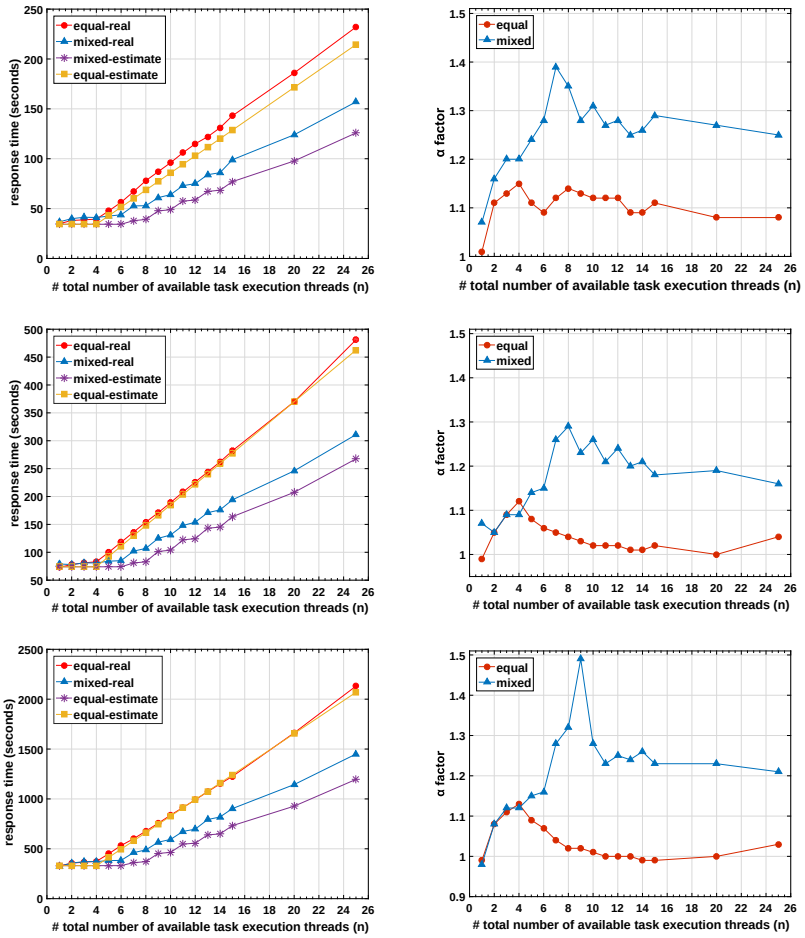

Fig. 2. Response Time (RT) and the α factor of linear flows with same and different task costs for $n \in [1, 25]$ executed by the 4-core/4-thread i5 machine for 2.4(top), 4.8(middle) and 21.8M(bottom) input records.

`essi.upc.edu/dtim/blog/post/tpc-di-etls-using-pdi-aka-kettle`. Each experiment run was repeated 5 times and the median times are reported; in all experiments the standard deviation was negligible.

The left column of Figures 2 and 3 shows how the response time of the two different types of data flows evolves as the number of tasks, and consequently the number of execution threads, increases. It also shows what the cost model estimates would be if no weights were considered. The main observation is twofold. First, the response time, as expected from Eqs. (2) and (3), stays approximately stable when $n \leq m$, and then, grows linearly when $n > m$. This behavior does not change with the increase in the data size. Second, estimates with no weights

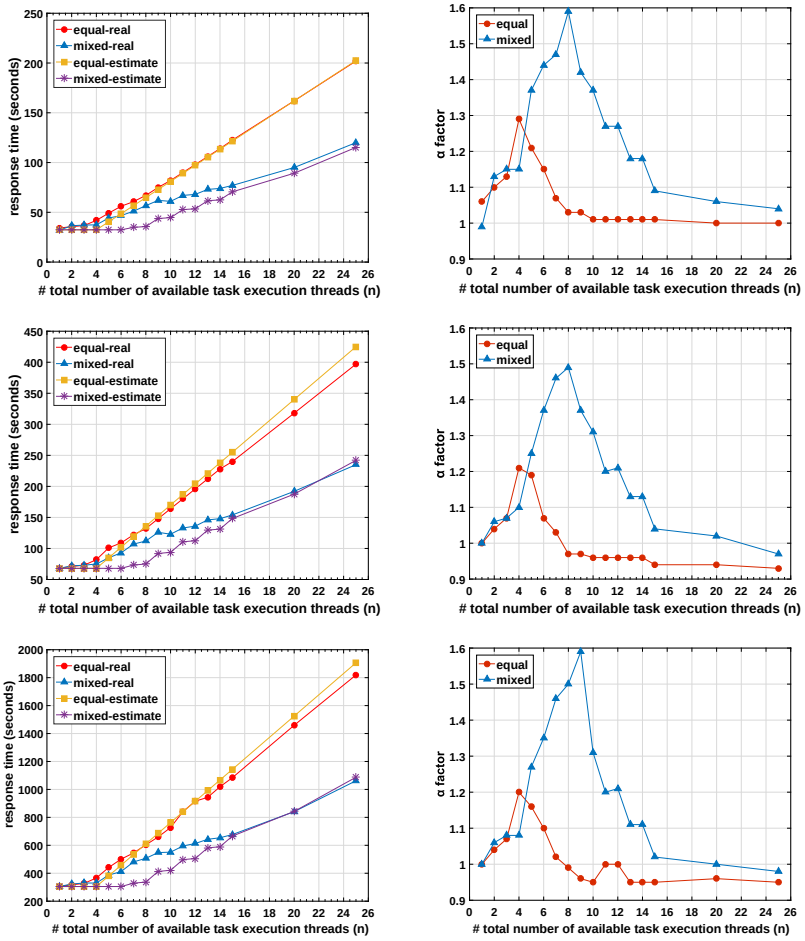

Fig. 3. Response Time (RT) and the α factor of linear flows with same and different task costs for $n \in [1, 25]$ executed by the 4-core/8-thread i7 machine for 2.4(top), 4.8(middle) and 21.8M(bottom) input records.

can underestimate the running time by up to 50%, whereas there are also cases when they overestimate the running times by a smaller factor (approx. 5%). More importantly, the main inaccuracies are observed in the mixed-cost case, which is more common in practice.

The α factor is shown in the right column of Figures 2 and 3. Values both lower and higher than 1 are observed. Although α captures the combination of overhead and improvement causes described in the previous section, the importance of each cause varies. In values greater than 1, resource contention is dominating; whereas, in values lower than 1, the fact that waits for resources are hidden outweighs any overheads. The main observations are as follows: (i)

	2.4 million records				4.8 million records				21.8 million records			
	4cores-4threads		*4cores-8threads*		*4cores-4threads*		*4cores-8threads*		*4cores-4threads*		*4cores-8threads*	
n	*2 paths*	*1 path*	*2 path*	*1 path*	*2 paths*	*1 path*	*2 paths*	*1 path*	*2 paths*	*1 path*	*2 path*	*1 path*
2	38.5	38.1	35.1	35.6	78	78	69	71	346	356	315	317
4	42	39.4	42.1	41.9	84	83	82	82	369	374	370	377
6	59.3	56.3	56	56.1	118	118	109	109	530	533	497	499
8	78	78	67	67	155	154	134	132	694	677	606	594
10	96	96	83	82	192	189	166	164	851	837	735	727
12	115	115	99	98	229	226	197	196	1040	991	879	916
14	134	131	115	114	265	262	229	228	1210	1151	1006	1019
16	153	152	131	129	304	305	260	256	1382	1311	1146	1152
18	164	170	145	145	327	323	287	289	1486	1493	1296	1299
20	187	186	160	162	380	370	323	318	1720	1662	1454	1437

Table 1. Comparison of running times between flows with the same number of tasks but a) with 2 independent and b) a single segment (in seconds).

the α factor varies significantly for the same dataset when the number of tasks is modified; (ii) α can be of significant magnitude corresponding to more than 50% increase in the task costs; (iii) for flows that consist of up to 4 tasks with equal cost, the α factor continuously grows (i.e., contention is dominating) and then, when the number of tasks further increases, the behavior differs between cases; and (iv) for data flows with different task costs and $n > m$, the α factor increases sharply for flows with up to 7-9 tasks depending on the input data size.

A linear flow with multiple independent pipelined segments

In Table 1, we show the running times of flows with the same number of tasks when all tasks belong to a single pipelined segment and when there are two segments belonging to two different paths originating from the same source. We can observe that the running times are similar. From this observation, we can draw the conclusion that the magnitude of the weights (i.e., the w^c and the corresponding α factors) depend on the number of concurrent tasks and need not be segment-specific; that is, it is safely to assume that all concurrent tasks share the same factors.

Estimating the response time of a flow: the complete case

In the previous sections, we showed how we can estimate the response time of a single pipelined segment in data flows. Now, we leverage our proposal to more generic data flows with multiple pipeline segments, in order to estimate the response time of flows that consist of multiple pipeline segments. To this end, we employ a simple list scheduling simulator. The steps of this methodology are described, as follows:

1. Receive as input the flow DAG, the cost (c_i) of all the tasks of a dataflow, the number of available cores, and the α factors.
2. Isolate all the single-pipeline segments of the flow with the help of the parallelism type task metadata.
3. Split the input in blocks of a fixed size B.
4. Create a copy for the first block for each task directly connected to a source and insert it in a FCFS (First Come First Serve) queue.
5. Schedule blocks arbitrarily to cores until there are no blocks in the queue under the condition that a task can process at most one block at time at any core:

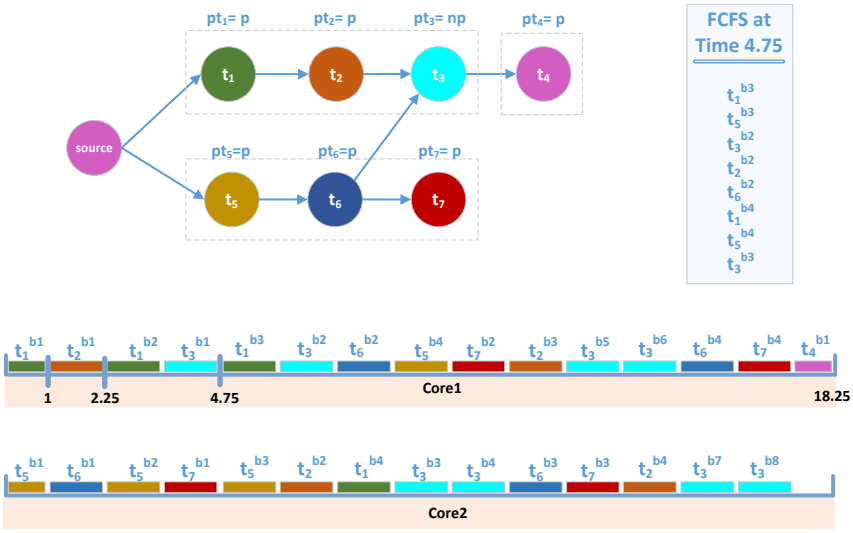

Fig. 4. Example of running a generic flow on 2 cores; the dotted borders denote pipeline segments.

(a) when a block finishes its execution, re-insert it in the queue annotated by the subsequent tasks in the DAG;
(b) if the task is a child of a source, insert its next block in the queue ;
(c) if a blocking task has received its entire input, start scheduling the corresponding blocks for the segment initiating from this task.

6. The response time is defined by the longest sequence of block allocations to a core.

In Figure 4, we present an example with a flow running on 2 cores, where all task costs per block are 1, each task receives as input 4 blocks and emits 4 other processed blocks except t_3, which outputs a single block. The α factor is 1, when there are up to two concurrent tasks, and 1.25 otherwise. Concurrent tasks are those for which there is at least 1 block either in the queue or being executed. In this example, the running time is when the work on the first is completed.

4.3 Considering communication costs

We need to consider communication only in settings where multiple machines are employed. Broadly, we can distinguish among the following three cases:

1. On each sender, there is a single thread for computation and transmission. In this case, both z_i and $z_{i,j}$ in Eq. (1) are 1 to denote that computation and transmission occur sequentially.

2. On each sender, there is a separate thread for data transmission, regardless of the number of outgoing edges. In this case, depending on which type of cost dominates, only one of z_i and $z_{i,j}$ is set to 1, since computation and transmission overlap in time.
3. On each sender or receiver, there is a separate thread for each edge. If all edges share the same network, then we can follow the same approach as in the case of multiple pipelined tasks sharing a single core.

The first two cases assume a push based data communication model, whereas the third one applies to both push and pull models.

4.4 Considering partitioned parallelism

Paritioned flows running on multiple machines can be covered by our model as well. More specifically, we can model and estimate the DAG flow instance on each machnine independently using the same approach, and then take the maximum running time as the final one. The factors may differ between partitioned tasks. Finally, if a DAG instance does not start its execution immediately, we need to add the time to receive its first input (which kicks-off its execution) to its estimated running time.

5 Conclusions and Future Work

In this work, we show that up to date the existing cost models do not estimate accurately the *response time* of real data flow execution, which heavily depends on parallelism. We propose a model that considers the time overlaps during the task execution, while it is capable of quantifying the impact of concurrent task execution. The latter is an aspect largely overlooked to date and may lead to significant inaccuracies if neglected, e.g., we provided simple examples of deviations up to 50%.

Our work can be extended in several ways. Applying the proposed model relies on the existence of accurate weight information; deriving efficient ways to approximate the weights before flow execution is an open issue. More thorough validating experiments (e.g., using the complete TPC-DI benchmark) are required. Also, instead of using a list scheduler simulation, one could develop analytic cost models that directly estimate the response time. Another direction for future work is to make a deep dive into the low-level resource utilization and wait measurements to establish the detailed cause of contention. Finally, there is a lack of flow cost-based optimization techniques that directly target the minimization of response time as modeled in this work.

References

1. Kunal Agrawal, Anne Benoit, Fanny Dufossé, and Yves Robert. Mapping filtering streaming applications with communication costs. In *SPAA*, pages 19–28, 2009.

2. Kunal Agrawal, Anne Benoit, Fanny Dufossé, and Yves Robert. Mapping filtering streaming applications. *Algorithmica*, 62(1-2):258–308, 2012.
3. Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas R. Burdick, and Shivakumar Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in systemml. *Proc. VLDB Endow.*, 7(7):553–564, 2014.
4. Artem M. Chirkin, A. S. Z. Belloum, Sergey V. Kovalchuk, and Marc X. Makkes. Execution time estimation for workflow scheduling. WORKS '14, pages 1–10, 2014.
5. Amol Deshpande and Lisa Hellerstein. Parallel pipelined filter ordering with precedence constraints. *ACM Transactions on Algorithms*, 8(4):41:1–41:38, 2012.
6. David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6), 1992.
7. F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11):1256–1267, 2012.
8. Georgia Kougka and Anastasios Gounaris. Optimization of data-intensive flows: Is it needed? is it solved? In *Proc.DOLAP*, pages 95–98, 2014.
9. Georgia Kougka, Anastasios Gounaris, and Alkis Simitsis. The many faces of data-centric workflow optimization: A survey. *CoRR*, abs/1701.07723, 2017.
10. Nitin Kumar and P. Sreenivasa Kumar. An efficient heuristic for logical optimization of etl workflows. In *BIRTE*, pages 68–83, 2010.
11. Ilia Pietri, Gideon Juve, Ewa Deelman, and Rizos Sakellariou. A performance model to estimate execution time of scientific workflows on the cloud. WORKS '14, pages 11–19. IEEE Press, 2014.
12. Meikel Poess, Tilmann Rabl, and Brian Caufield. TPC-DI: the first industry benchmark for data integration. *PVLDB*, 7(13):1367–1378, 2014.
13. Astrid Rheinlnder, Arvid Heise, Fabian Hueske, Ulf Leser, and Felix Naumann. Sofa: An extensible logical optimizer for udf-heavy data flows. *Information Systems*, 52:96 – 125, 2015.
14. Juwei Shi, Jia Zou, Jiaheng Lu, Zhao Cao, Shiqiang Li, and Chen Wang. Mrtuner: A toolkit to enable holistic optimization for mapreduce jobs. *Proc. VLDB Endow.*, 7(13):1319–1330, August 2014.
15. A. Simitsis, P. Vassiliadis, and T. K. Sellis. State-space optimization of ETL workflows. *IEEE Trans. Knowl. Data Eng.*, 17(10):1404–1419, 2005.
16. Alkis Simitsis, Kevin Wilkinson, Umeshwar Dayal, and Malú Castellanos. Optimizing ETL workflows for fault-tolerance. In *ICDE*, pages 385–396, 2010.
17. R. Singhal and A. Verma. Predicting job completion time in heterogeneous mapreduce environments. In *IEEE IPDPSW*, pages 17–27, 2016.
18. U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *Proc.PVLDB*, pages 355–366, 2006.
19. Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. ICAC '11, pages 235–244. ACM, 2011.
20. Ramana Yerneni, Chen Li, Jeffrey D. Ullman, and Hector Garcia-Molina. Optimizing large join queries in mediation systems. In *ICDT*, pages 348–364, 1999.
21. Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Performance modeling of mapreduce jobs in heterogeneous cloud environments. CLOUD '13, pages 839–846, 2013.