Schedule Optimization for Data Processing Flows on the Cloud

Herald Kllapi
herald@di.uoa.gr

Eva Sitaridi [*]
evas@di.uoa.gr

Manolis M. Tsangaris
mmt@di.uoa.gr

Yannis Ioannidis
yannis@di.uoa.gr

MaDgIK Lab, Dept. of Informatics and Telecommunications
University of Athens, Ilissia GR15784, Greece.

ABSTRACT

Scheduling data processing workflows (dataflows) on the cloud is a very complex and challenging task. It is essentially an optimization problem, very similar to query optimization, that is characteristically different from traditional problems in two aspects: Its space of alternative schedules is very rich, due to various optimization opportunities that cloud computing offers; its optimization criterion is at least two-dimensional, with monetary cost of using the cloud being at least as important as query completion time. In this paper, we study scheduling of dataflows that involve arbitrary data processing operators in the context of three different problems: 1) minimize completion time given a fixed budget, 2) minimize monetary cost given a deadline, and 3) find trade-offs between completion time and monetary cost without any a-priori constraints. We formulate these problems and present an approximate optimization framework to address them that uses resource elasticity in the cloud. To investigate the effectiveness of our approach, we incorporate the devised framework into a prototype system for dataflow evaluation and instantiate it with several greedy, probabilistic, and exhaustive search algorithms. Finally, through several experiments that we have conducted with the prototype elastic optimizer on numerous scientific and synthetic dataflows, we identify several interesting general characteristics of the space of alternative schedules as well as the advantages and disadvantages of the various search algorithms. The overall results are quite promising and indicate the effectiveness of our approach.

Categories and Subject Descriptors

G.1.6 [**Mathematics of Computing**]: Optimization—*Constrained optimization, Simulated annealing*; H.2.4 [**Database Management**]: Systems—*Distributed databases, Query processing*; H.2.8 [**Database Management**]: Database Applications; H.4 [**Database Management**]: Information Systems Applications

[*] Present address: Columbia University, New York, NY 10027.

SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

General Terms

Algorithms, Design, Economics, Experimentation, Performance

Keywords

Cloud computing, Dataflows, Query optimization, Scheduling

1. INTRODUCTION

Complex on-demand data retrieval and processing is a characteristic of several applications and combines the notions of querying & search, information filtering & retrieval, data transformation & analysis, and other data manipulations. Such rich tasks are typically represented by data processing graphs, having arbitrary data operators as nodes and their producer-consumer interactions as edges. Suppose a user wants to find the names and images of authors of ACM publications. This could be expressed in SQL as follows:

```
SELECT UNIQUE auth.name, img.image
FROM AuthorDB auth, ImageBD img
WHERE auth.pub="ACM" AND auth.name=face(img.image)
```

Function `face()` detects the face of the person in a particular image and returns her name. The SQL query is optimized [18] and transformed into an execution plan, represented as a dataflow graph. In a distributed environment, the optimizer must decide, among others, where each node of the plan will be executed. This level of optimization is called scheduling. Scheduling the processing nodes of a dataflow graph onto a set of available machines is a well-known NP-complete problem, even in its simplest form [13] [25]. Traditionally, the only criterion to optimize is the completion time or makespan of the dataflow, and many heuristic scheduling algorithms have been proposed for that problem [19].

Recently, *cloud computing* has attracted much attention from the research community [1]. Thanks to virtualization, it has evolved over the years from a paradigm of basic IT infrastructures used for a specific purpose (Ad-Hoc Clusters), to Grid computing [10], and finally, to several paradigms of resource provisioning services: depending on the particular needs, infrastructures (IaaS), platforms (PaaS), and software (SaaS) can be provided as services [12].

One of the differences between Ad-Hoc Clusters and Clouds under the Infrastructure as a Service (IaaS) paradigm, is the cost model of resources. Ad-Hoc Clusters represent a fixed capital investment made up-front and relatively small operational cost paid over time. On the other hand, Clouds are characterized by elasticity and offer their users the ability to lease resources only for as long as needed, based on a per quantum pricing scheme, e.g. one hour [2]. Together with the lack of any up-front cost, this represents a major benefit of Clouds over Ad-Hoc Clusters. The ability to use computational resources that are available on demand, challenges the way we implement algorithms and systems. Execution of dataflows can now be *elastic*, providing several choices of price-to-performance ratio and making the optimization problem of dataflow scheduling at least two dimensional.

To illustrate the complexity of this task, we use the example of dataflow graph in Fig. 1. Nodes represent data processing operations (operators) and edges represent producer-consumer relationships. Assume that the execution time of A and C is 1 hour and the execution time of each B is 10 minutes. Also assume that a pricing scheme of charging the use of a machine per hour is employed by the Cloud. Consider two possible schedules for this dataflow: *Schedule 1*: execute all operators in one virtual host.

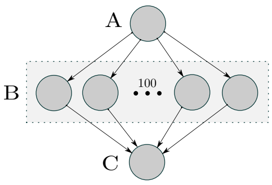

Figure 1: A split (A), compute(B), and merge(C) dataflow.

First A is executed, then 100 Bs, and finally C. The time required for the dataflow to complete is the sum of the execution times of all operators: $60 + 10 \cdot 100 + 60 = 1120$ minutes or 18.6 hours. Since there is only one host involved, the cost for this schedule corresponds to 19 hours of use. *Schedule 2*: execute each operator in a different virtual host. First, A is executed and the data produced is transferred to the hosts that execute B. All Bs are executed in parallel and their data is transferred to one host, which executes C. Assuming that the data transfer time is negligible, the completion time is only $60+10+60 = 130$ minutes or 2 hours and 10 minutes. However, the cost for this schedule is huge and corresponds to 102 hours of resource usage. All hours will be charged for all virtual hosts. *Schedule 2* will run about 9 times faster than *Schedule 1*, but will cost 5 times more.□

The number, nature, and temporal & monetary costs of available schedules depend on many parameters, such as the dataflow characteristics (execution time of operators, amount of data generated, etc.), the cloud pricing scheme (quantum length and price), the network bandwidth, and more. The choice of how much parallelism to use or, equivalently, the optimal trade-off between completion time and money spent, depends on the needs of the particular user concerned. For example, a user may have budget constraints but be relaxed about completion time; another user may have a strict deadline but no concern about money; finally, a third user may have no a-priori constraints but wants to choose herself the best trade-off after having been shown all choices.

In this work, we make the following contributions:

- We model the scheduling of dataflows on the cloud as a 2D optimization problem between time and money, which is mostly ignored until now.
- We study the space of alternative schedules that arise from that model and investigate the time-money trade-off these provide for different types of dataflows and cloud environments.
- We propose several greedy, probabilistic, and exhaustive optimization algorithms to explore the space of alternative schedules.
- We present the results of several experiments and draw various interesting insights on the trade-offs available and the effectiveness of all algorithms in exploring these trade-offs.

The remainder of the paper is organized as follows. In Section 2, 3, 4, and 5, we formulate the problem of dataflow scheduling in the context of ADP [30], a prototype dataflow evaluation system, which has served as our experimental testbed in this work. In Section 6, we present our approach along with different optimization algorithms that we use. In Section 7, we present the results of the experimental evaluation. In Section 8, we present an overview of the related work and finally, in Section 9, we conclude and present some future directions.

2. PROBLEM FORMULATION

User requests take the form of queries in some high-level language (e.g. SQL, Hive [28], etc.). In principle, optimization could proceed in one giant step, examining all execution plans that could answer the original query and choosing the one that is optimal and satisfies the required constraints. Alternatively, given the size of the alternatives space, optimization could proceed in multiple smaller steps, each one operating at some level and making assumptions about the levels below. This is in analogy to query optimization and execution in traditional databases but with the following differences: operators may represent arbitrary operations on data with unknown semantics, algebraic properties, and performance characteristics, and are not restricted to come from a well-known fixed set of operators (e.g., those of relational algebra); optimality may be subject to QoS or other constraints and may be based on multiple diverse relevant criteria, e.g., monetary cost of resources, staleness of data, etc., and not just solely on performance; the resources available for the execution of a data processing graph are flexible and reservable on demand and are not fixed a-priori. These differences make dataflow optimization essentially a new challenging problem; they also generate the need for run-time mechanisms that are not usually available. Our optimization process, introduced in [30], has three different layers of abstractions described below.

Operator Graphs: Their nodes are data operators and their (directed) edges are operator interactions in the form of producing and consuming data. Operators encapsulate data processing algorithms and may be custom-made by end users. At this abstraction layer, of great importance are algebraic equivalences that operators satisfy. These include typical algebraic transformations, e.g., associativity, commutativity, or distributivity, (de)compositions, i.e., operators being abstractions of whole operator graphs that involve compositions, aggregations, and other interactions of more specific operators, and partitions, i.e., operators being amenable to replication and parallel processing by each replica of part of the original input, in conjunction with some pre- and post-processing operators.

Concrete Operator Graphs: These are similar to operator graphs but their nodes are concrete operators, i.e., software components that implement operators in a particular way and carry all necessary details for their execution. At this layer, capturing an operator's available implementation(s) is the critical information. In general, there may be multiple concrete operators implementing an operator, e.g., a low-memory but expensive version and a high-memory but fast one; a multi-threaded version and a single-threaded one; or two totally different but logically equivalent implementations of the same operator. An example from databases is the $join$ operator, which has multiple implementations: $nested - loops$ join has low memory consumption but long execution time; $hash$-join uses more memory but has short execution time.

Execution Plans: These are similar to concrete operator graphs, but their nodes are concrete operators that have been allocated resources for execution and have all their parameters set. At this layer, the assignment of concrete operators to containers is performed. The crucial information here is the resources needed to execute the operators, e.g., CPU and memory. This paper presents the modeling and a methodology for this stage of optimization. The statistics needed are assumed to be known, calculated either by mathematical formulas or by historical data.

A **dataflow** is represented as a directed acyclic graph (DAG) $graph(ops, flows)$. Nodes (ops) correspond to arbitrary concrete operators and edges ($flows$) correspond to data transferred between them. An **operator** in ops is modeled as $op(time, cpu, memory, behavior)$ where $time$ is the execution time of the operator & cpu is its average CPU utilization measured as a percentage of the host CPU power when executed in isolation (without the presence of other operators), $memory$ is the maximum memory required for the effective execution of the operator, and $behavior$ is a flag that is equal to either *pipeline* (PL) or *store-and-forward* (S&F). If behavior is equal to S&F, all inputs to the operator must be available before execution; if it is equal to PL, execution can start as soon as some input is available. Two typical examples from databases are $sort$ and $select$ operators: $sort$ is S&F and $select$ is PL. We assume that each operator has a uniform resource consumption during its execution (cpu, $memory$, and $behavior$ do not change). A **flow** between two operators, producer and consumer, is modeled as $flow(producer, consumer, data)$, where $data$ is the size of the data transferred.

The **container** is the abstraction of the host, virtual or physical, encapsulating the resources provided by the underlying infrastructure. Containers are responsible for supervising operators and providing the necessary context for executing them. A container is described by its CPU, its available memory, and its network bandwidth: $cont(cpu, memory, network)$.

A **schedule** S_G of a dataflow graph G is an assignment of its operators into containers $schedule(assigns)$. An individual operator **assignment** is modeled as: $assign(op, cont, start, end)$ where $start$ and end are the start and end time of the operator correspondingly, executed in the presence of other operators.

Time $t(S_G)$ and **money** $m(S_G)$ costs are the completion time and the monetary cost of a schedule S_G of a dataflow graph G. The following two sections define the way we model and calculate the time and money costs of a schedule.

The **cloud** is a provider of virtual hosts (containers). We model only the compute service of the cloud and not the storage service. Assuming that the storage service is not used as temporary space, a particular dataflow G will read and write the same amount of data for any schedule S_G of operators into containers. So the cost of reading the input and writing the output is the same.

In the rest of the paper, we use the "." notation to denote a property, e.g., the running time of an operator A is denoted as $A.time$. Indifferent values are denoted by "$-$", e.g., $assign(op, cont, -, -)$.

3. TIME MODELING

To calculate the completion time of a schedule executed over a set of containers, several aspects of operator execution must be modeled. This is an issue for any distributed system and our particular approaches do not depend on having a Cloud underneath or any other architecture. In this section we present the approach adapted in our work, inspired by or taken from earlier works.

There are two types of temporal constraints, those implied by the dataflow graph and those imposed by the execution environment. The dataflow graph implies constraints based on the inter-operator dependencies captured by its edges, but also by the nature of the operators themselves. An S&F operator cannot be executed until all its inputs are available, while a PL operator must wait for all inputs produced by S&F operators. This raises an important issue especially when the two operators are in different containers and introduce network costs. We describe our approach to model the various cases in Section 3.1.

The execution environment constraints are due to resource limitations when multiple operators use them concurrently. In that respect, we categorize container resources as time-shared and space-shared [11]. Time-shared resources can be used by multiple operators concurrently at very low overhead. Concurrent use of space-shared resources, however, implies high overheads beyond container limits of resources. We consider memory as the only space-shared resource, whereas CPU and network as time-shared resources. Constraints are imposed only by space-shared resources: in every container, at any given moment, memory must be sufficient for the execution of the running operators. On the other hand, CPU and network requires particular treatment. We describe our approach to model CPU and network overloading in Section 3.2.

3.1 Network Cost

Two issues must be addressed regarding network cost: handling of S&F and handling of PL operators. The network communication is modeled with special operators that perform data transferring (dt). These operators are injected between the operators of a flow $flow(producer, consumer, data)$, if $producer$ and $consumer$ are assigned to different containers. Always two dt operators are injected, one attached after the producer and one attached before the consumer. If the operator is S&F (Fig. 2), the dt operator creates a new node in the dataflow graph. On the other hand, if the operator is PL (Fig. 3), the functionality of the dt operator is injected into the operator itself without changing the dataflow graph. Fig. 3A illustrates the actual execution of the pipeline operator A intermixed with the dt operator and Fig. 3B illustrates the modeling of A. The execution time is increased and the CPU

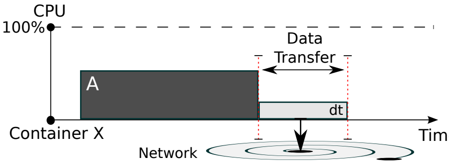

Figure 2: An S&F operator with a data transfer operator attached.

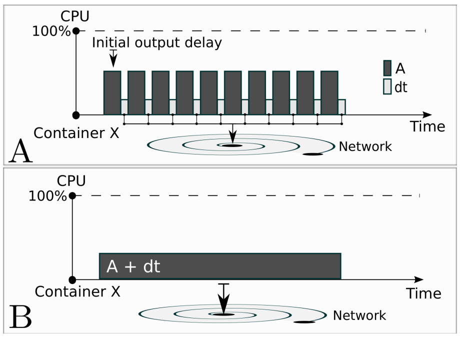

Figure 3: PL operator A intermixed with data transfer operator dt (A) and the modeling of the operator (B)

utilization is decreased. For simplicity, we assume that the initial output delay is 0. In the transformed graph, all network communication is performed by PL operators, no matter what the initial nodes were.

Store-and-Forward: Let A be an S&F operator that belongs to dataflow G defined as $A(-,-,-,S\&F)$ with $assign(A,X,-,-)$. For every operator B with $flow(A,B,D_{A\to B})$ and $assign(B,Y,-,-)$, if $X \neq Y$ insert in G a data transfer operator node dt between A and B as follows: remove $flow(A,B,D_{A\to B})$; insert $flow(A,dt,D_{A\to B})$, $flow(dt,B,D_{A\to B})$, and $assign(dt,X,-,-)$. The execution time of dt is as follows:

$$dt.time = \frac{D_{A\to B}}{\min(X.network, Y.network)} \tag{1}$$

$$dt(time, DT_{CPU}, DT_{MEM}, PL)$$

with DT_{CPU} and DT_{MEM} being fixed system-wide values for the CPU utilization and memory requirements of data transfer operators. The same method is followed for all operators C $flow(C,A,D_{C\to A})$, whose outputs are consumed by A.

Pipeline: Let A and B be two connected pipeline operators that belong to dataflow graph G with $flow(A,B,D_{A\to B})$. We assume that the execution time of both operators is fully overlapped. Let the assignments of A and B be $assign(A,X,-,-)$ and $assign(B,Y,-,-)$ correspondingly, with $X \neq Y$. Using Eq. 1, the running time is:

$$T = \max(A.time + dt.time, B.time + dt.time)$$

The new properties of operator A will be:

$$A(T, \frac{A.time * A.cpu + dt.time * DT_{CPU}}{T}, -, -)$$

The *memory* and *behavior* of A will not change. The new properties of B are calculated in a similar way. The same technique is applied on all connected PL operators.

3.2 Operator Overlap

Two or more operators assigned to the same container share container resources. A particular problem arises with the CPU when overlapping operators require at some point more than 100% utilization together. For example consider Fig. 4 with the three overlapping operators A, B, and C. Given the assumption of uniform resource consumption by operators, the regions between those thresholds have uniform CPU utilization. There is a problem in region X, where the total CPU requirements are higher than the container's capacity. This is addressed by uniformly extending the length of region X and, therefore, the overall execution time of all operators by reducing their CPU utilization within X.

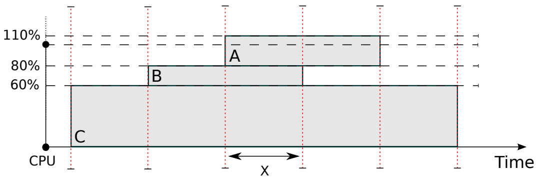

Figure 4: Operator A, B, and C executing in the same container. At region X the overall CPU utilization exceeds 100% and as a result the operators will get a fraction of the processor and their duration will stretch by 10% during that region.

More specifically, let G be a dataflow graph and C a container. Let $OP = \{op_j\}$ the set of operators assigned to C and $R = \{r_i\}$ the set of time regions formed by the beginning and endings of operators (in order of i). Let $time(r_i)$ and $cpu(r_i)$ be the duration and CPU utilization of region r_i. Based on the above, the following holds:

$$cpu(r_i) = \sum_{j=1}^{|OP|} (op_j).cpu * \delta(op_j, r_i, C)$$

with

$$\delta(op_j, r_i, C) = \begin{cases} 1, & \text{if } op_j \text{ active in } r_i \text{ in } C \\ 0, & \text{otherwise} \end{cases}$$

The duration of r_i is calculated as follows:

$$time(r_i) = \begin{cases} time(t_i), & \text{if } cpu(t_i) \leq 1 \\ time(t_i) * cpu(t_i), & \text{otherwise} \end{cases}$$

The scaling of region r_i affects the duration of all operators in OP that are active in r_i. The same technique used for CPU sharing is applied for network sharing as well.

4. MONEY MODELING

Cloud providers lease computing resources that are typically charged based on a per time quantum pricing scheme. This quantum is typically one hour although recently other alternatives seem to appear as well [2]. The minimum monetary cost $m(S_G)_{min}$ of executing a dataflow graph G in a cloud environment given a schedule S_G and the cloud pricing scheme (quantum time Q_t and cost Q_m of leasing a container for Q_t) is a challenging task. In this work we employ an approximation. On each container, we slice time into windows of length Q_t starting from the first operator of the schedule. The financial cost is then a simple count of the time-windows that have at least one operator running, multiplied by Q_m.

$$m(S_G) = Q_m * (\sum_{i=1}^{|C|} \sum_{j=1}^{|W|} \epsilon(c_i, w_j))$$

with $C = \{c_i\}$ being the set of containers, $W = \{w_j\}$ being the set of time-windows, and

$$\epsilon(c_i, w_j) = \begin{cases} 1, & \text{if at least one operator is active in } w_j \text{ in } c_i \\ 0, & \text{otherwise} \end{cases}$$

The fragmentation $f(S_G)$ of a schedule S_G is the time during which the resources are not being used but are charged for.

$$f(S_G) = Q_t * \sum_{i=1}^{|C|} \sum_{j=1}^{|W|} (\epsilon(c_i, w_j) - \tau(c_i, w_j))$$

with $\tau(c, w)$ being the fraction of time that operators are active in the container c_i during time window w_j. It is easily proved that $\lim_{Q_t \to 0} f(S_G) = 0$ for any given dataflow. That is that if cloud resources are charged for exactly the time being used, there is no fragmentation overhead. In Fig. 5 we show a plan with four operators. The monetary cost is $5 * Q_m$. There are three quanta not fully used. The fragmentation of the schedule is $5 * Q_t - (1 + \frac{1}{4} + \frac{1}{4} + 1 + \frac{1}{2}) * Q_t = 2 * Q_t$. □

5. SCHEDULING PROBLEMS

As mentioned earlier, the space of solutions is two dimensional. The solutions that belong to the skyline (Pareto curve) [24] represent trade-offs between time and money. We define two types of optimization problems: constrained and skyline. All problems are illustrated in Fig. 6. The **constrained** problems are (C1) *find*

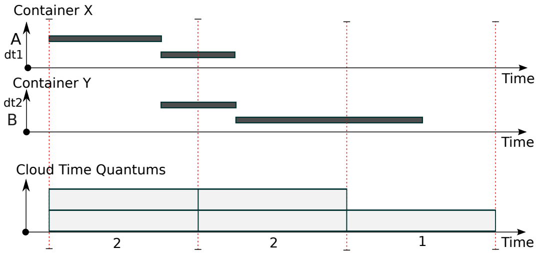

Figure 5: Financial cost of a scheduling plan. Operators A and B are assigned to different containers. The time is sliced into 3 time-windows and 5 quanta are leased during the execution.

the fastest plan within a pre-specified financial budget (Fig. 6A) and (C2) *find the cheapest plan within a pre-specified deadline*(Fig. 6B). Those problems are symmetric and are essentially one-dimensional optimization problems, only under constraints on the other dimension. In the **skyline** problem (SK), no constraints are specified a-priori. Several solutions are proposed that are on the skyline of the time/money space and the one with the best trade-off between time and money is chosen after optimization. In Fig. 6C, skyline optimization will return all the Pareto optimal solutions and the relevant user will choose the one the user thinks best.

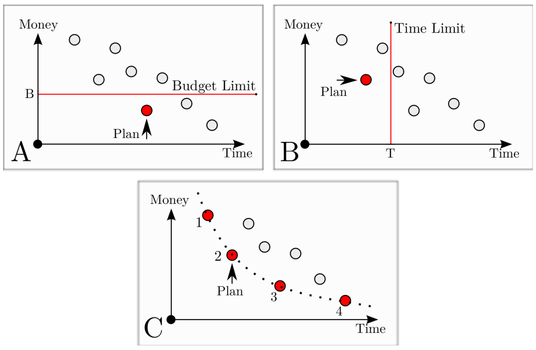

Figure 6: The three optimization problems: constraints (A and B) and skyline (C). The chosen plans are shown with an arrow.

The above skyline shape can be captured by a metric we propose, called *Dataflow Elasticity* E:

$$E = \frac{(T_{max} - T_{min})/T_{max}}{(M_{max} - M_{min})/M_{max}}$$

with T_{min} and T_{max} being the min and max values in the time dimension of the skyline of schedules. Likewise, M_{min} and M_{max} are the min and max values in the money dimension. This metric expresses the acceleration of the completion time when paying more money [31]. For dataflows with high elasticity, a small amount of money makes a great difference in time (i.e. is worth paying for completion time). On the other hand, paying more does not have a significant impact on completion time for low elasticity dataflows. For them, it is worth paying, say, 50% less to lose only 10% in completion time!

6. SCHEDULING ALGORITHMS

Independent of the particular type of optimization problem solved, essentially the same algorithms can be used to search the space of alternatives and find the optimal one(s). We propose a family of algorithms that follow a nested loop approach. The outer loop calls the inner optimizer several times, with different upper bounds on the number of parallel containers to be used in the schedule and stops when the optimality criterion doesn't improve much after a particular number of consecutive iterations.

The degree of parallelism P_G of a dataflow G is an important predictor of performance. It captures the maximum number of concurrent running operators, when the computation has infinite resources. Assume the number of containers and the network bandwidth to be infinite. Satisfying only the constraints implied by the dataflow, an approximation of P_G is the maximum number of operators whose execution overlap.

The inner loop, optimizes on either time or money under a constraint on the other. We use a suite of greedy and probabilistic local search algorithms for the inner loop that are presented in the following subsections.

6.1 Outer Loop Algorithm

We have implemented one generic nested loop optimizer (Alg. 1) that solves any of the constrained or skyline problems, depending on the values of its parameters. The parameters of Alg. 1 are:

Algorithm 1 Nested Loop Optimizer

```
Input:
    $G$: The dataflow graph
    CONST: Solution constraints
    FILTER: Solution space filter
    LIMIT: Container limit sequence generator
    STOP: Stopping condition
    OPT: Lower level optimizer
Output:
    $space$: The space of solutions

space ← ∅
while LIMIT.$hasNext()$ and STOP.$continue()$ do
    $limit$ ← LIMIT.$getNext()$
    $next$ ← OPT($G$, $limit$, CONST)
    $space$ ← FILTER($space \cup \{next\}$)
    STOP.$addFeedback(next)$
    LIMIT.$addFeedback(next)$
end while
return $space$
```

1) The dataflow graph G as defined in Section 2. 2) $CONST$ is a boolean routine returning whether or not a schedule is satisfying the constraints. If $CONST$ returns always true ($time < \infty$ & $money < \infty$) is enough to express the lack of constraints used in the skyline problem. 3) $FILTER$ is a routine that is applied on a set of schedules and returns a reduced set, removing the schedules that are dominated by others, according to time, money, or both (skyline). 4) $LIMIT$ is a generator of container limits. In its simplest form, it just has to generate limits up to the total number of operators in the dataflow. However, in most cases, this is a very loose upper bound. 5) $STOP$ is a boolean routine determining whether or not to stop the exploration based on some of many possible criteria. The simplest way is to call the lower level sequentially for all limits generated by $LIMIT$. Another possibility is to end the exploration when the difference between the values of the OPT parameter for a specified number of consecutive schedules is below a particular threshold. Finally, OPT is a single-objective optimizer that tries to optimally assign the operators to containers, minimizing either the time or the money related to the schedule. Below we outline how some of the parameters of Alg. 1 are instantiated for each of the three optimization problems defined above.

Problem C1: $CONST$ checks whether or not a schedule is covered by the given financial budget. $FILTER$ returns only the fastest schedule. $LIMIT$ generates at most 20 container limits splitting equally the [1, P_G] range. Condition $STOP$ is the last five schedules to not differ significantly with respect to completion time. Using linear regression, we compute the line in the time/iteration space using the five last iterations. If the slope of that line is less than 0.1 the exploration stops. OPT minimizes completion time having a budget limitation as constraint.

Problem C2: $CONST$ checks whether or not a schedule is within the given time limit. $FILTER$ return only the cheapest schedule. $LIMIT$ is the same as in Problem C1. $STOP$ is calculated in the same way as in C1 but the line is computed on the money/iteration space. OPT minimizes money having the time limitation as constraint.

Problem SK: $CONST$ accepts any schedule. $FILTER$ returns the skyline of solutions. $LIMIT$ is the same as C1 and the $STOP$ condition is always false. OPT can be any algorithm described in the following subsections. The size of the skyline is at most the number of container limits produced by $LIMIT$.

In the following subsections we define the various algorithms we have explored with respect to OPT.

6.2 Greedy Scheduling Algorithms

We have implemented several greedy scheduling algorithms [17] using different heuristics. In Alg. 2, we present the generic greedy algorithm. Only two routines have to be defined: $NEXT$ returns the next operator to be assigned choosing from a set of operators ready for that and $ASSIGN$ returns the container where the next operator will be assigned to. Each operator having no inputs, is a candidate for assignment. An S&F operator is a candidate, as soon as all of its inputs are available. A PL operator is a candidate, as soon as all of its inputs come from PL or from completed S&F operators.

Algorithm 2 Generic Greedy (GG)

```
Input:
    $G$: The dataflow graph
    $C$: The maximum number of parallel containers to use
    CONST: Solution constraints
    NEXT: Next operator to assign
    ASSIGN: Container the next operator is assigned to
Output:
    $S_G$: The schedule of $G$ with at most $C$ containers

$S_G.assigns \leftarrow \oslash$
$ready \leftarrow$ {operators in $G$ that has no dependencies}
while $ready \neq \oslash$ do
    $n \leftarrow \mathtt{NEXT}(ready)$
    $candidates \leftarrow$ {containers that assignment of $n$ satisfy CONST}
    if $candidates = \oslash$ then
        return $ERROR$
    end if
    $c \leftarrow \mathtt{ASSIGN}(n, candidates)$
    $ready \leftarrow ready - \{n\}$
    $ready \leftarrow ready +$ {operators in $G$ that constraints no longer exist}
    $S_G.assigns \leftarrow S_G.assigns + \{assign(n, c, -, -)\}$
end while
return $S_G$
```

We define four greedy algorithms: **G-BRT** balances container utilization, **G-MNT** minimizes network traffic, **G-MPT** minimizes completion time, and **G-MPM** minimizes monetary cost. In particular, at every step, G-BRT assigns the operator with maximum running time to the container that will minimize the standard deviation of the utilization of the containers in the schedule. The latter is the summation of the execution time of the operators assigned to the container. G-MNT assigns the operator with the maximum output size to the container that minimizes the data transferred through the network. G-MPT assigns the operator with the maximum execution time to the container that minimizes completion time. Finally, G-MPM assigns the operator with the maximum output size to the container that minimizes monetary cost.

6.3 Local Search Scheduling Algorithms

The local search method we use is simulated annealing [14]. We implemented a generic simulated annealing that requires the definition of only three routines: $INIT$ specifies the initial schedule from which the search begin, $COST$ returns the value of the optimization parameter for a particular schedule, and $NEIGHBOR$ returns a neighbor of a particular schedule. We do not accept neighbors that do not satisfy the constraints.

We define the following algorithms: **SA-MPT** begins with a random assignment and the $COST$ routine returns the completion time of a particular schedule, **SA-MPT2** begins with the output of G-MPT as its initial state and has the same $COST$ routine as SA-MPT, **SA-MPM** begins with a random assignment and the $COST$ routine returns the money of a particular schedule, and **SA-MPM2** begins with the output of G-MPM as its initial state and has the same $COST$ routine as SA-MPM. All algorithms produce a random neighbor by assigning a randomly chosen operator to a different container, also chosen randomly. More sophisticated methods of choosing neighbors are left for future work.

6.4 Schedule Duration Estimation

Alg. 3 is in the heart of all algorithms described. Given any (partial) schedule of operators, this algorithm estimates when and for how long every operator will run. Thus, the completion time and monetary cost is estimated for the whole schedule.

Algorithm 3 Schedule Duration Estimation (SDE)

```
Input:
    $G$: The dataflow graph
    $S_G$: A (partial) $schedule(assigns)$
Output:
    $S_G$: The schedule with calculated $start$ and $end$ for all ops in $S_G$

$ready \leftarrow \oslash$
$time \leftarrow 0$
$G \leftarrow G$ + {data transfer ops before & after S&F operators if needed}
$queued \leftarrow$ {ops in $G$ that have no dependencies}
for all operators $A$ in $queued$ that satisfy memory constraints do
    $ready \leftarrow ready + \{assign(A, -, time, -)\}$
end for
while $ready \neq \oslash$ do
    $next\_event \leftarrow \mathtt{find\_next\_event}(ready)$
    $terminated \leftarrow \mathtt{forward\_in\_time}(next\_event, ready)$
    $time \leftarrow next\_event$
    for all operators $A$ in $terminated$ do
        $S_G \leftarrow S_G + \{assign(A, -, -, time)\}$
    end for
    $ready \leftarrow ready - terminated$
    $queued \leftarrow queued$ + {ops in $G$ that constraints no longer exist}
    for all ops $A$ in $queued$ that satisfy memory constraints do
        $ready \leftarrow ready + \{assign(A, -, time, -)\}$
    end for
end while
return $S_G$
```

Routine `find_next_event` returns the time-stamp when first termination of operator occurs assuming uniform behavior. The `forward_in_time` routine simulates the execution of all operators in *ready* until the time-stamp computed by `find_next_event`, and returns the ones that have terminated.

7. EXPERIMENTAL EVALUATION

In this section, we describe the overall setup of our experimentation effort and the results we have obtained from it. Although we have experimented with all three optimization problems, we only present the results of the one on skyline optimization, i.e., with no budget or deadline constraints, as it is the most challenging. Similar things hold for all parameters influencing our experiments, where only the most interesting or characteristic results are discussed, the rest offering not much additional insight.

7.1 Experimental Setup

The experiments conducted are characterized by four elements:

Execution Environment: In our experiments, we have realistically assumed that all containers are identical, i.e., they have the same resources (cpu, memory, and network). In particular, we have assumed containers with one CPU and total memory and maximum network bandwidth equal to 1.0.

Dataflow Graph Structure: We examine four families of dataflows: Montage [16] (Fig. 7A), Ligo [8] (Fig. 7B), Cybershake [7] (Fig. 7C), and Lattice (Fig. 8A). The first three are abstractions of actual dataflows that are used in real applications: Montage is used by NASA to generate custom mosaics of the sky, Ligo is used by the Laser Interferometer Gravitational-wave Observatory to analyze galactic binary systems, and Cybershake is used by the Southern California Earthquake Center to characterize earthquakes.

If we observe the time behavior of the workloads, we can identify phases of high and low parallelism. Parallelism is reduced when operators collect or distribute data from a lot of other operators (bottlenecks) or when an increased amount of data is to be transferred. Montage experiences a phase with the highest amount of parallelism, while it has several bottleneck operators. Ligo has more evenly distributed phases of parallelism and moderate number of bottleneck operators. Cybershake has two high parallelism phases and just four bottleneck operators, while it transfers large data volumes. Lattice, by design, provides us with a choice of parallelism and bottlenecks. Lattice is a purely synthetic dataflow family that we have designed generalizing the typical Map-Reduce dataflow (such as that of Fig. 1). Specific Lattice dataflows have a certain height (H) and branching factor (B) and are denoted as H-B Lattice. For example, Fig. 8B shows the 5-2 Lattice.

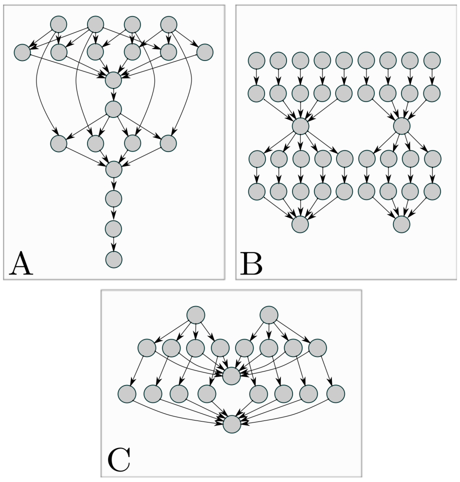

Figure 7: Montage(A), Ligo(B), and Cybershake(C) dataflows.

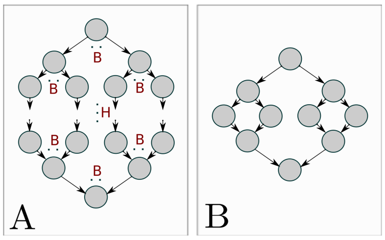

Figure 8: Lattice - general dataflow graph (A) and an example with $H = 5$ and $B = 2$ (B).

Table 1: Lattice Dataflows

H-B:	500-1	11-3	9-4	7-7	5-21	3-498
Size:	500	485	426	457	485	500

Table 2: Lattice Operator Properties

Property	**Values**
time	0.2, 0.4, 0.6, 0.8, 1.0
cpu	0.4, 0.45, 0.5, 0.55, 0.6
memory	0.05, 0.1, 0.15, 0.2, 0.25
data	0.2, 0.4, 0.6, 0.8, 1.0

We have experimented with several sizes of dataflow graphs, from 5 to approximately 500 operators. Here we present the results of the largest graphs (500 operators) as they are the most challenging. The nature of the results for smaller graphs has been similar. Montage, Ligo, and Cybershake dataflows have been produced by the Pegasus dataflow generator [4]. Lattice dataflows have been handcrafted in several forms according to the H-B parameters and are shown in Table 1.

Operator Types: In our experiments, we have indicated the values of operator properties as percentages of the corresponding parameters of container resources. For example, an operator having memory needs equal to 0.4 uses 40% of a container's memory. Furthermore, execution times are given as percentages of the cloud's time quantum and so are data sizes (inputs/outputs of operators), taking into account the network speed. For example, an execution time of 0.5 indicates that the operator requires half of a time quantum to complete its execution (say, 30 minutes according to the predominant Amazon cost model). Likewise, an output of size 0.2 requires one fifth of a time quantum to be transferred through the network if needed. This way, the output data size is in inverse correlation with network speed. Money is measured as described in Section 4.

We have used synthetic workloads based on Montage, Ligo, and Cybershake dataflows as defined in [4]. Since our intention was to study data intensive workloads, we scaled up the specified run times and operator output sizes by a factor of 50 and 1000 respectively; run time to output size ratio was increased by 20. We also set operator memory to 10% of the container capacity, since they were not specified by the original benchmarks. The properties of operators in Lattice are chosen with uniform probability from the corresponding sets of values shown in Table 2.

Data transfer CPU utilization is $DT_{CPU} = 0.05$ and memory needs $DT_{MEM} = 0.001$ (defined in Section 2). We experimented with changing the output by a factor of $\frac{1}{0.64}$, $\frac{1}{2.56}$, $\frac{1}{40.96}$, and $\frac{1}{10485.76}$. We have used dataflows with all operators being S&F and all being PL.

Schedule Optimization Algorithms: For the outer loop of our Nested Loops Optimizer (Alg. 1), the numbers of available containers examined have been 10, 30, 50, 70, 90, 110, 130, and 150.

For the inner loop, we have experimented with all algorithms de-

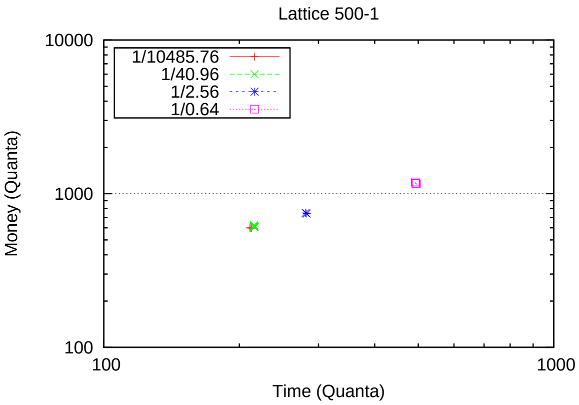

Figure 10: Time/Money skylines of S&F 500-1 Lattice with different output sizes (log-log scale).

scribed in Section 6: greedy (G-BRT, G-MNT, G-MPT, G-MPM) and simulated annealing (SA-MPT, SA-MPT2, SA-MPM, SA-MPM2). In addition, we have also run experiments with a random schedule generator to explore the time/money space of solutions and gain insights into the nature of the optimization problem at hand. The next subsection presents the findings of exactly that experiment, whereas the following one those with the actual optimization algorithms.

7.2 Time/Money Space Exploration

The random schedule generator used for this experiment produces 10,000 (almost) random schedules for each dataflow studied. The details of the algorithm are omitted due to lack of space. Below we analyze how the time/money space is affected by various characteristics of the dataflow optimized or the execution environment.

Output Data Size: In Fig. 9, we show the skylines for various output data sizes of two Lattice dataflows (3-498 and 7-7) and in Fig. 12 we show the skylines of Ligo, Montage, and Cybershake. The key observation is that, for a particular dataflow graph G, in general, the elasticity metric E^G increases as the output size decreases. Also, for small output, elasticity is greater for high degree of parallelism ($E^{3-498}_{small} > E^{7-7}_{small}$). In some cases the output size does not affect the elasticity too much (Lattice 7-7 in Fig. 9 and Ligo in Fig. 12).

The elasticity metric is almost undefined ($E^{3-498}_{small} \approx \frac{0}{0}$) for 3-498 Lattice dataflow with large output. This is also the case for all outputs in 500-1 Lattice dataflow shown in Fig. 10. This means that there is no time/money trade-off, consequently, the fastest schedule is also the cheapest one.

In Fig. 11, we plot entire solution spaces (not just the skylines), for the same dataflow graphs but only for the smallest ($\frac{1}{10485.76}$) and largest ($\frac{1}{0.64}$) outputs. For small outputs, the more containers we use, the faster the schedules run in general, i.e., time and money are anti-correlated and true elasticity is present. For large outputs, in the extreme case, they are correlated and the fastest schedule is also the cheapest. We also observe that, for large outputs, sometimes the solution space is not continuous. This happens because when the memory is not sufficient, the operator execution is delayed. For large outputs, the data transfer operators are long running, so their delay adds significant delay in the whole dataflow.

In Fig. 12, we show the skylines of Ligo, Montage, and Cybershake. We observe that $E^{Ligo} > E^{Montage} > E^{Cybershake}$, i.e., elasticity is analogous to the amount of parallelism exhibited by the dataflow structure, where Ligo has more parallelism than Montage, which has more than Cybershake.

Multi-processing vs. Uni-processing: Our model allows multiple operators to run concurrently in the same container. The skylines of Lattice dataflows produced with multi- and uni-processing are shown in Fig. 13. Schedules produced under uni-processing are slower and more expensive than those produced under multi-processing for small output sizes while not differing much for large output sizes. The main reason for this is under-utilization of the container when running one operator at a time. Domination of the network usage time over the processing time for large outputs explains the fact that the skylines do not differ much.

Cloud Time Quantum: We have also experimented with the cloud time quantum size. In Fig. 14, we show the skyline of Lattice dataflows for quantum size 0 and 1 using multi- and uni-processing. We observe that the cloud quantum time affects the fastest schedules (many containers) more than the slowest (few containers). This was expected since, in general, the more containers we use, the higher the fragmentation we have. In Fig. 14, we also observe that, under uni-processing, elasticity is very high ($E^{7-7}_{small} \approx \frac{T_{max}-T_{min}}{0}$) so, the fastest and slowest schedule cost the same. Hence, for these specific cases, the optimization problem becomes one dimensional.

Pipeline vs. Store-&-Forward: In Fig. 15, we observe that the 500-1 Lattice dataflow with PL operators has some elasticity, while that with S&F operators does not. Furthermore, the schedules of the former are faster and some of them cheaper. On the other hand, the schedules for the 7-7 Lattice dataflow with PL operators are in general slower and more expensive than for that with S&F operators. This has two explanations: 1) due to PL operator execution overlapping, all are stretched in time to reach the one with the maximum execution time and 2) due to data transfer overlapping with processing, the space-shared resources (memory) consumed by PL operators are occupied for a longer period of time. This also explains the fact that for large number of containers, the 7-7 Lattice dataflows schedules are faster than S&F because more memory is available. Two connected PL operators assigned to the same container will be executed faster than two S&F with the same functionality, but when are assigned to different containers, this may not be the case.

General conclusions: Based on the above discussion, one can draw the following interesting conclusions regarding the time/money space of alternative schedules:

1. There is correlation between an operator's completion time and the time spent on data transfer. For large values of $\frac{time}{data}$ completion time is anti-correlated with monetary cost and the level of elasticity is high. On the other hand, for small values, time is correlated to money and there is limited elasticity, so solving a one-dimensional optimization problem is enough. For example, linear dataflows with S&F operators are not elastic while those with PL are.

2. Multi-processing, in general, produces faster and cheaper plans than uni-processing.

3. Pipelining should be used with caution, since it can add a significant overhead in both completion time and money.

4. Fragmentation of resources causes significant increases in monetary costs, especially when using many containers.

5. Different types of elasticity exist for different types of dataflows. Some dataflows are more elastic than others, and some are not elastic at all.

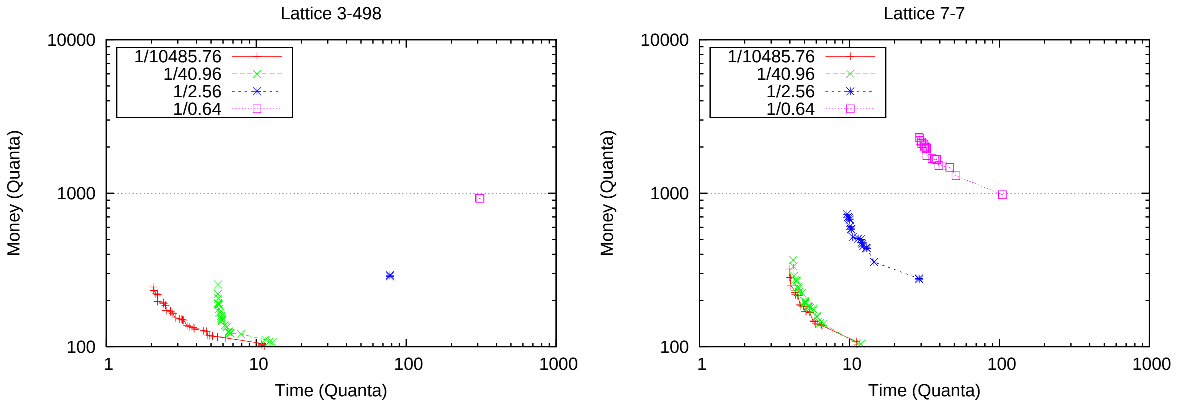

Figure 9: Time/Money skylines of S&F Lattice dataflows with different output sizes (log-log scale).

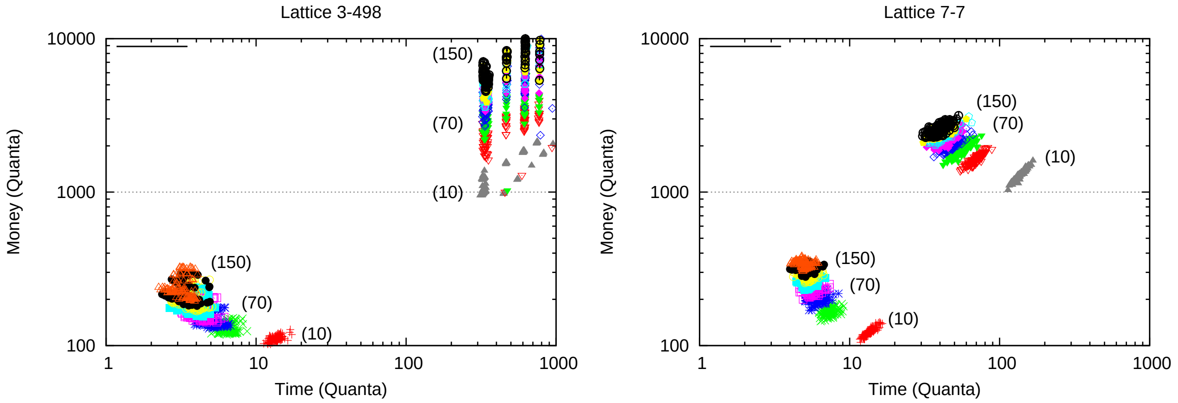

Figure 11: Time/Money of S&F Lattice dataflows with small ($\frac{1}{10485.76}$) and large ($\frac{1}{0.64}$) output size using different numbers of available containers.

7.3 Schedule Optimization

In Fig. 16, we present the schedules produced by the subsequent invocations of the algorithms against Montage with S&F operators and small output. We see that there is elasticity between time and money. It is clear that the time decreases when the number of parallel containers increases. Money tends to increase with the number of containers but not strictly. There are two different reasons for this: First, availability of a number of containers does not imply use of all of them in the schedule obtained; the schedule's actual maximum parallelism may be less than the number of available containers or the schedule identified by the optimization algorithms as optimal may have not used all the available parallelism. The number of available containers is just an upper limit. Second, due to the quantized pricing policy, the average usage of quanta is better and the produced schedule more compact.

In Fig. 17, we show the time/money space of Montage and the skylines produced by various algorithms. Interestingly, when these skylines are merged together, the overall skyline has schedules from different algorithms. Schedules given by SA-MPM give cheap schedules, SA-MPT give solutions faster than SA-MPM but more expensive, and G-BRT give the fastest and most expensive solutions. It appears that one size does not fit all. We also observe that G-MNT (minimize network traffic) gives solutions similar to G-MPM using only a small number of containers. Algorithm SA-MPT starts with a random assignment and the solution it finds is very close to the one produced by the greedy G-MPT. Algorithms G-MPM and SA-MPM2 do not provide trade-offs. This happens because all operators are S&F and thus, the cheapest schedule is always the one using a single container. On the other hand, SA-MPM, starting with a random schedule, give elasticity.

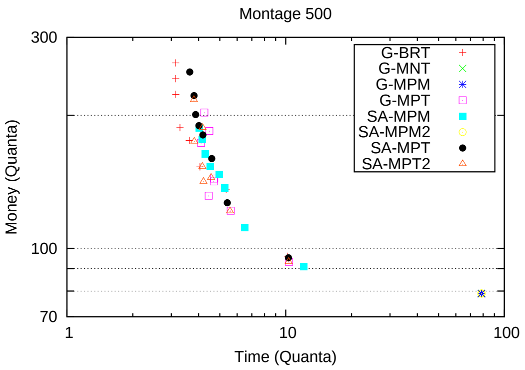

Figure 17: Time/Money by different algorithms for Montage dataflow with small output (log-log scale).

In Fig. 18, we show the running time of the different algorithms for several types of S&F Lattice dataflows with small output. Clearly, the level of parallelism afforded by the dataflow graph affects the running time of the algorithms, some more (e.g., G-MPT,

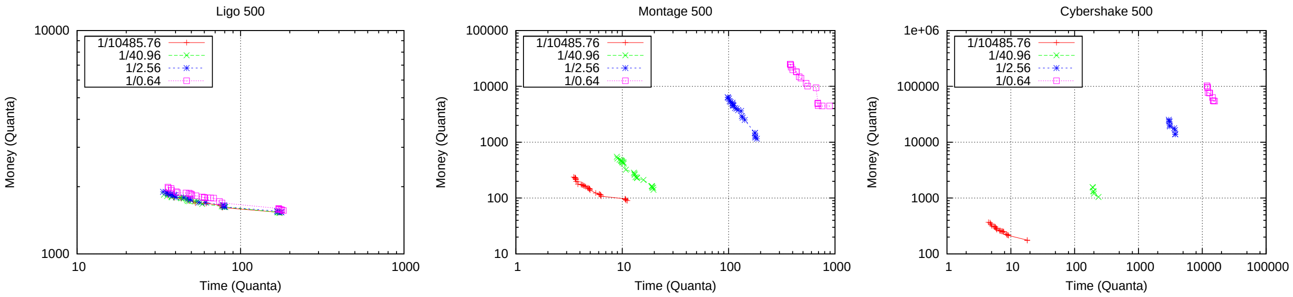

Figure 12: Time/Money skylines of Ligo, Montage, and Cybershake dataflows using different output sizes (log-log scale).

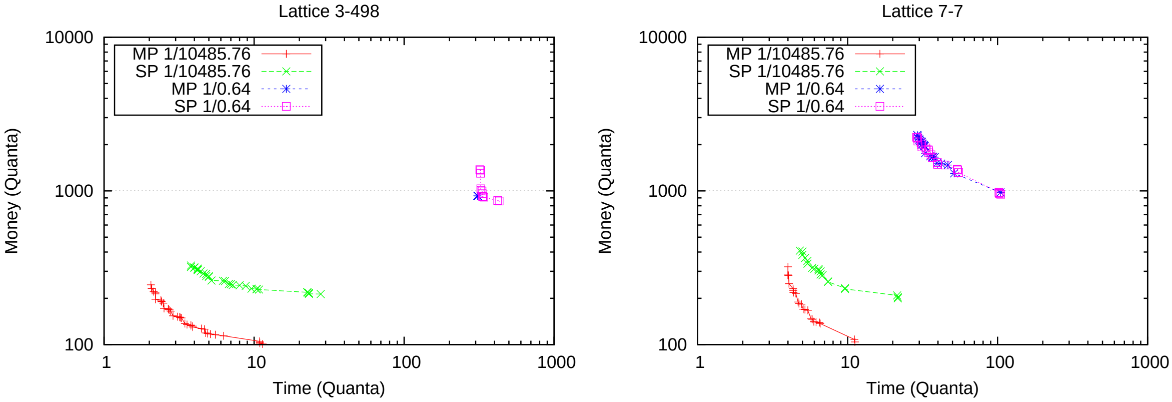

Figure 13: Uni-processing vs. multi-processing skylines of S&F Lattice dataflows (log-log scale).

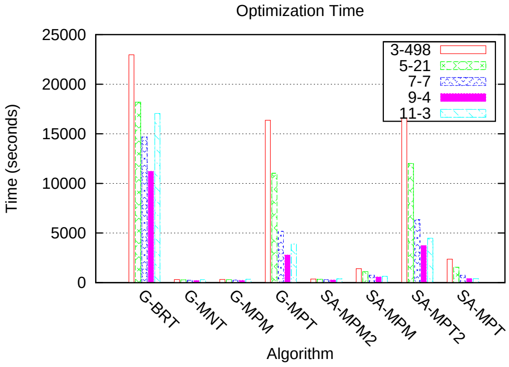

Figure 18: Optimization time of all algorithms against the Lattice dataflows.

SA-MPT2) than others (e.g., G-BRT, SA-MPT). Since, G-BRT uniformly distributes the operators into containers, the expected number of containers used is near to those available. Naturally, this algorithm is near the worst case among all greedy algorithms.

General conclusions: In summary, the following are the key observations on the behavior of the various optimization algorithms:

1. Different optimization algorithms tend to explore different areas of the time/money space, some of them generating skyline schedules that exhibit interesting trade-offs that no other algorithm does. This raises a meta question of choosing the optimal optimization algorithm depending on the user needs, which is left for future work.
2. Sophisticated search methods, such as simulated annealing, do not seem to improve significantly the schedules produced by greedy algorithms, while they require much more time to execute. The space of alternatives is huge and more study need to be done in finding ways of exploring the solution space efficiently. For large dataflows, greedy algorithms appear to be the right choice.

8. RELATED WORK

Typically, some middleware is used to execute user-defined code in distributed environments. The Condor/DAGMan/Stork [20] set is the state-of-the-art technology of High Performance Computing. Nevertheless, Condor was designed to harvest CPU cycles on idle machines; running data intensive workflows with DAGMan is very inefficient [26]. Many systems use DAGMan as middleware, like Pegasus [6] and GridDB [21]. Proposals for extensions of Condor to deal with data intensive scientific workflows do exist [26], but to the best of our knowledge, they have not been materialized yet. In [9] is presented a case study of executing the Montage dataflow on the cloud examining the trade-offs of different dataflow execution modes and provisioning plans for cloud resources.

Hadoop is a platform that follows the Map-Reduce [5] paradigm to achieve fault tolerance and massive parallelism. Several high-lever query languages have been developed on top of Hadoop, e.g., PigLatin [22] and Hive [28] (used by Facebook), and the same holds for several applications, e.g., Mahout [3], a platform for large-scale machine learning. The dataflow graphs used in Map-Reduce are relatively restricted, as they are mostly Lattice 3-N dataflows, for some N, and this reduces opportunities for optimization.

Dryad [15] is a commercial middleware by Microsoft that has a more general architecture than Map-Reduce since it can parallelize

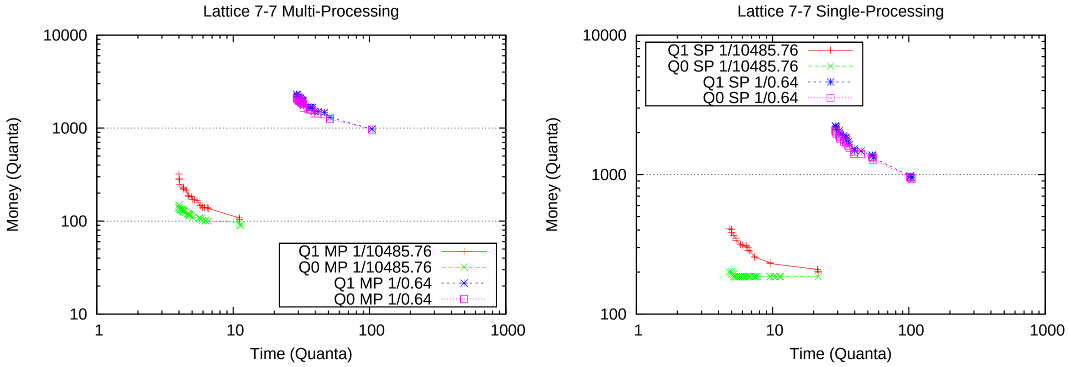

Figure 14: Time/Money skylines of S&F Lattice dataflows with quantum size equal to 0 and 1 (log-log scale).

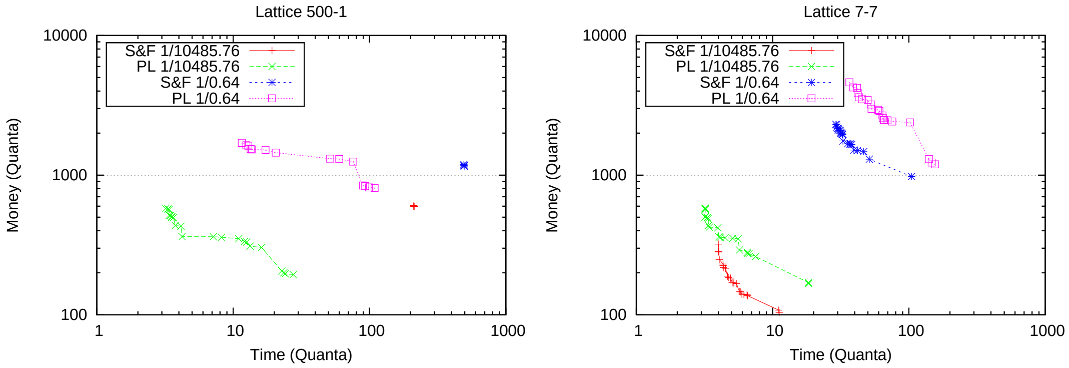

Figure 15: Time/Money store and forward vs. pipeline for different dataflows and outputs (log-log scale).

any dataflow. Its schedule optimization, however, relies heavily on hints requiring knowledge of node proximity, which are generally not available in a cloud environment. It also deals with job migration by instantiating another copy of a job and not by moving the job to another machine. This might be acceptable when optimizing solely time but not when the financial cost of allocating additional containers matters.

Nefeli [29] is a cloud gateway that uses hints for efficient execution of workloads. It uses the cloud at a lower level than our work, being aware of the physical resources and the actual locations of the virtual machines. This information may not be generally available, however, especially in commercial clouds.

There are also several efforts that move in the same direction as our work but try to solve simpler versions of the problem. Examples include a particle swarm optimization of general dataflows having a single-dimensional weighted average parameter of several metrics as the optimization criterion [23] and a heuristic optimization of independent tasks (no dependencies) having the number of machines that should be allocated to maximize speedup as the optimization criterion given a predefined budget[27].

Finally, at the foundational level, we have followed a parallelism and resource-sharing model for optimal scheduling of relational operators of query execution plans with time and space-shared resources [11] and have generalized it to arbitrary operators.

In summary, we capitalize on the elasticity of clouds and produce multiple schedules, enabling the user to select the desired trade-off. To the best of our knowledge, no dataflow processing system deals with the concept of elasticity or two-dimensional time/space optimization, which constitute our key novelties.

9. CONCLUSIONS & FUTURE WORK

We have presented a framework for three different problems of dataflow schedule optimization on the Cloud, for a two-dimensional optimality criterion of time and money, and a methodology to solve these problems. Experimental results have revealed interesting insights on the space of alternatives that needs to be explored, on the different levels of elasticity offered by different cases of dataflow structures, operator characteristics, and other parameters, and on the effectiveness and efficiency of several optimization algorithms.

The methodology proposed is used in the final step of the optimization of the ADP system and all the parameters are instantiated automatically using functions or statistics collected during previous executions. It can also be used separately, however, helping cloud service providers in advising consumers on choosing the appropriate time/money trade-offs.

Our plans for future work move in several directions. We intend to study the concept of elasticity further and identify its sensitivity to different workloads and environment characteristics, as well as the ability of optimization algorithms to exploit it. We also plan to evaluate the model itself on how good is in predicting the real workload. Furthermore, we plan to investigate the problem of scalability using adaptive and incremental schedule generation techniques. Finally, we intend to compare our approach with the optimization methods of other distributed computing systems such as Hadoop.

10. REFERENCES

[1] R. Agrawal et al. "The Claremont report on database research". *SIGMOD Record*, 37(3):9–19, 2008.

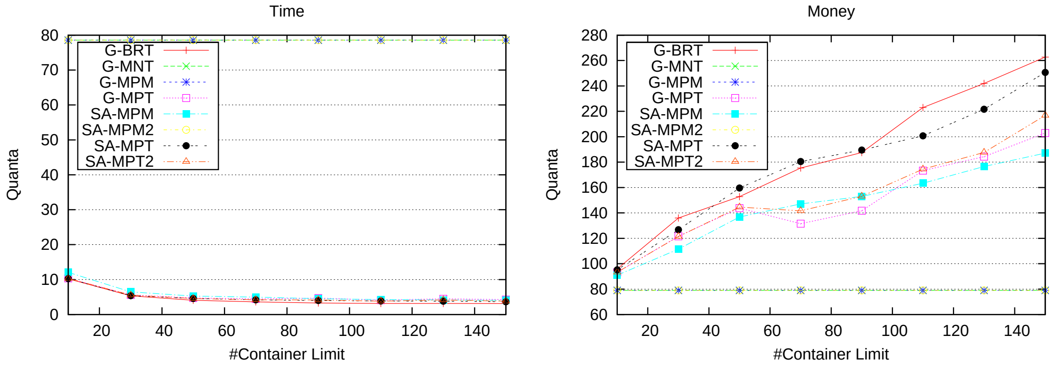

Figure 16: Time & Money as a function of the number of containers for different algorithms on the Montage dataflow for small output.

[2] Amazon. "Amazon Elastic Compute Cloud (EC2), http://aws.amazon.com/ec2/", 2010.

[3] Apache. "Mahout : Scalable machine-learning and data-mining library, http://mahout.apache.org/", 2010.

[4] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. pp. 1 –10, nov. 2008.

[5] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In *6th Symposium on Operating System Design and Implementation*, pp. 137–150, 2004.

[6] E. Deelman and e. al. "Pegasus: Mapping Large Scale Workflows to Distributed Resources in Workflows in e-Science". *Springer*, 2006.

[7] E. Deelman et al. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *IEEE e-Science*, pp. 14, 2006.

[8] E. Deelman, C. Kesselman, and more. "GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists". In *IEEE HPDC*, pp. 225–, 2002.

[9] E. Deelman, G. Singh, M. Livny, G. B. Berriman, and J. Good. The cost of doing science on the cloud: the montage example. In *IEEE/ACM SC*, pp. 50, 2008.

[10] I. T. Foster. "The Anatomy of the Grid: Enabling Scalable Virtual Organizations". In *CCGRID*, pp. 6–7, 2001.

[11] M. N. Garofalakis and Y. E. Ioannidis. "Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources". In *VLDB*, pp. 296–305, 1997.

[12] L. M. V. Gonzalez, L. R. Merino, J. Caceres, and M. Lindner. "A break in the clouds: towards a cloud definition". *Computer Communication Review*, 39(1):50–55, 2009.

[13] R. L. Graham. "Bounds on Multiprocessing Timing Anomalies". *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[14] Y. E. Ioannidis and E. Wong. "Query Optimization by Simulated Annealing". In *SIGMOD Conference*, pp. 9–22, 1987.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks". In *EuroSys*, pp. 59–72, 2007.

[16] J. C. Jacob et al. "Montage: a grid portal and software toolkit for science, grade astronomical image mosaicking". *Int. J. Comput. Sci. Eng.*, 4(2):73–87, 2009.

[17] J. Kleinberg and E. Tardos. *"Algorithm Design"*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

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

[19] Y.-K. Kwok and I. Ahmad. "Benchmarking and Comparison of the Task Graph Scheduling Algorithms". *J. Parallel Distrib. Comput.*, 59(3):381–422, 1999.

[20] M. J. Litzkow, M. Livny, and M. W. Mutka. "Condor - A Hunter of Idle Workstations". In *ICDCS*, pp. 104–111, 1988.

[21] D. T. Liu and M. J. Franklin. "The Design of GridDB: A Data-Centric Overlay for the Scientific Grid". In *VLDB*, pp. 600–611, 2004.

[22] C. Olston et al. "Pig latin: a not-so-foreign language for data processing". In *SIGMOD Conference*, pp. 1099–1110, 2008.

[23] S. Pandey, L. Wu, S. M. Guru, and R. Buyya. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *IEEE AINA*, pp. 400–407, 2010.

[24] C. H. Papadimitriou and M. Yannakakis. "Multiobjective Query Optimization". In *PODS*, 2001.

[25] G. M. R., J. D. S., and S. Ravi. "The Complexity of Flowshop and Jobshop Scheduling". *Mathematics of operations research*, 1(2):117–129, 1976.

[26] S. Shankar and D. J. DeWitt. "Data driven workflow planning in cluster management systems". In *HPDC*, pp. 127–136, 2007.

[27] J. N. Silva, L. Veiga, and P. Ferreira. Heuristic for resources allocation on utility computing infrastructures. In B. Schulze and G. Fox, editors, *MGC*, pp. 9. ACM, 2008.

[28] A. Thusoo et al. "Hive - a petabyte scale data warehouse using Hadoop". In *ICDE*, pp. 996–1005, 2010.

[29] K. Tsakalozos, M. Roussopoulos, V. Floros, and A. Delis. Nefeli: Hint-based execution of workloads in clouds. In *IEEE ICDCS*, pp. 74–85, 2010.

[30] M. M. Tsangaris and more. Dataflow processing and optimization on grid and cloud infrastructures. *IEEE Data Eng. Bull.*, 32(1):67–74, 2009.

[31] H. R. Varian. *"Intermediate Microeconomics : A Modern Approach"*, chapter 15, Market Demand. W. W. Norton and Company, 7th edition, Dec. 2005.