An Algorithmic Framework for Geo-Distributed Analytics

Srikanth Kandula, Ishai Menache, Joseph (Seffi) Naor, Erez Timnat
Microsoft Research and Technion – Israel Institute of Technology
{srikanth, ishai}@microsoft.com, {naor, ereztimn}@cs.technion.ac.il

Abstract Large scale cloud enterprises operate tens to hundreds of datacenters, running a variety of services that produce enormous amounts of data, such as search clicks and infrastructure operation logs. A recent research direction in both academia and industry is to attempt to process the "big data" in multiple datacenters, as the alternative of centralized processing might be too slow and costly (e.g., due to transferring all the data to a single location). Running such *geo-distributed analytics* jobs at scale gives rise to key resource management decisions: Where should each of the computations take place? Accordingly, which data should be moved to which location, and when? Which network paths should be used for moving the data, etc. These decisions are complicated not only because they involve the scheduling of multiple types of resources (e.g., compute and network), but also due to the complicated internal data flow of the jobs – typically structured as a DAG of tens of stages, each of which with up to thousands of tasks. Recent work [17, 22, 25] has dealt with the resource management problem by abstracting away certain aspects of the problem, such as the physical network connecting the datacenters, the DAG structure of the jobs and/or the compute capacity constraints at the (possibly heterogeneous) datacenters. In this paper, we provide the first analytical model that includes all aspects of the problem, with the objective of minimizing the makespan of multiple geo-distributed jobs. We provide exact and approximate algorithms for certain practical scenarios, and suggest principled heuristics for other scenarios of interest.

1 Introduction

Many enterprises have data and computation clusters spread across the world. Due to efficiency or privacy considerations, the data may only be available in distributed locations. For example, datacenter server logs [1, 19, 23] and surveillance videos [2, 18] are massive datasets that are accessed infrequently. It is efficient to store such datasets *in-place*, that is at or close to where the data is generated. As another example, due to privacy considerations, the EU and China now require user and enterprise data to be stored within their borders. Our goal in this paper is to consider algorithmic questions that arise in executing data-parallel queries [10, 24, 26] on such geographically distributed datasets.

We identify three key aspects of this problem. First, distributed sites where data is generated have limited amount of compute and storage capacity (e.g., a Starbucks store recording video). Hence, it may not be efficient to run all of the computation at the site having the data. Further, capacities can vary across sites by several orders of magnitude; an organization may have rented hundreds of VMs at Azure or EC2 locations and have thousands or more servers at various on-premise locations.

Second, when large amounts of data have to be analyzed, data-parallel queries can exhaust the network bandwidth available to-and-from the various sites. Hence, task scheduling has to more carefully account for network usage. However, the topology of the physical network that interconnects the various sites is complex: (a) when data is present at *off-net* locations such as Starbucks stores or airports, the network paths cross many ISPs leading to capacity bottlenecks within the network; and (b) even when data is distributed among *on-net* sites, such as datacenters connected with a private wide-area network [15, 20], there can be bottlenecks within the network. Furthermore, the round trip delays in the wide-area network can be several orders of magnitude larger than those within a cluster; the delays and the available capacity on the network paths can also vary over time.

Third, the queries that need to be supported are not necessarily a single map stage followed by a single reduce stage. Rather, decision support queries tend to be quite complex. For example, the benchmarks in TPC-H [3] and TPC-DS [4] lead to directed acyclic graphs (DAGs) containing many stages, in various popular frameworks such as Hive [24], Spark-SQL [8] or SCOPE [10]. Furthermore, most production systems have a cadence that offers a predictable set of *recurring* queries. Such queries can, for example, *digest* raw logs and video streams into structured datasets so as to speed up the processing of subsequent user queries.

We are unaware of any prior work that takes into account these three aspects. Many systems ignore all three issues. For example, Iridium [22] supports map-reduce queries (i.e. DAG of depth 2), assumes that the only network bottlenecks are at the in- and out-links of sites (“congestion-free core”) and does not account for limits on compute capacity. Many other works such as SWAG [17], Tetrium [16], Carbyne [14] make identical assumptions. Geode [25] supports more general queries, but only considers minimizing the total amount of data crossing sites; a measure that may or may not translate to fast query execution.

The reason behind these rather inadequate solutions is that the underlying algorithmic problem – how to schedule a DAG of stages (each consisting of many tasks) across a network of sites – is difficult. Scheduling a dependent set of tasks on a pool of processors is itself challenging [13] and has received attention in the scheduling literature, e.g., [12, 21, 9] and references therein. In addition, the problem at hand has to also route the network traffic resulting from task placement. That is, the demands on the network depend on task placement. Furthermore, the way that the network routes these demands affects the finish time of the tasks. Many prior works model less general versions of the problem described above [22, 25, 14, 16, 17] and motivate their heuristics by arguing that the general problem is impractical to solve.

1.1 Our Results

The primary contribution of our paper is a rigorous algorithmic model for studying the geo-distributed analytics scheduling problem, which captures the aspects highlighted above: (1) a general directed acyclic graph (DAG) of dependencies between stages, (2) compute capacity and other limits per site and (3) a general topology for the physical network. In our model, we are given a DAG representing the jobs and their inter-stage dependencies. Our goal is to minimize the time to complete all of the jobs, i.e., the makespan. We consider different types of dependencies between stages (of a job), both of which arise in practice: *soft-precedence* and *strict-precedence*. With strict-precedences, each stage has to wait for all stages preceding it in the dependency DAG to complete their work entirely, before it can start its processing. With soft-precedences, dependent stages can overlap in execution, though the advancement of a stage is bounded by the advancement of the stages on which it depends. We also model different dependency types between any pair of stages u and v: *all-to-all*, where all tasks in u communicate with all tasks in v, and *scatter-gather*, a much sparser and targeted communication pattern (see §2.2). We choose these precedences and dependency types because they effectively capture practice (see §2).

We make several important modeling decisions allowing us to formulate the problem in a tractable way. In particular, we operate at a *stage* granularity, which makes the problem-size orders of magnitude smaller, and allows us to bypass excess rounding of variables. After defining the general model (§2), we derive exact and approximate algorithms for several scenarios of interest. In particular, we present an optimal solution for soft-precedence with scatter-gather dependency (§3). This solution is achieved using a linear programming formulation. Appealingly, the linear program for soft-precedence directly dictates a feasible schedule. For the strict-precedence case, we formulate a different linear program, that unfortunately does not directly imply a schedule. However, it does indicate where to execute every stage, but not when to do that. Thus, we apply an algorithm on top of the LP solution to construct a practical schedule. We show an ω-approximation for strict-precedence with all-to-all dependency, under a bus network topology, where ω is the width of the logical DAG (§4). We note that this result implies, as an important special case, an *optimal* algorithm for a single geo-distributed *map-reduce* [11] job. For the remaining practical scenarios, we have designed principled heuristics (§5) – all-to-all dependencies with both soft- and strict-precedence, under a general network topology. Notably, we show how to address the multiplicative constraints that arise when modeling all-to-all dependencies by heuristically linearizing them using Taylor expansion. The linear programs are solved iteratively, using values from previous iterations as the points around which we expand the Taylor series. Our simulations show that this heuristic is nearly-optimal, offering a practical solution for the remaining scenarios, although we do not have formal guarantees for it.

2 The Model

In this section, we present the geo-distributed analytics model. In §2.1, we describe the basic properties and assumptions of the model. In §2.2 and §2.3, we define the dependency types and the precedence models across stages, respectively. Finally, in §2.4, we describe the different network topologies that we consider.

2.1 Preliminaries

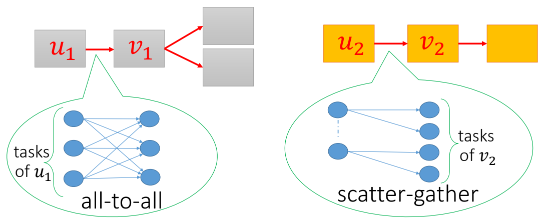
We are given a set of jobs. Each job is modeled as a DAG whose nodes correspond to *stages*; a stage consists of tasks that perform similar computation on different subsets of the data in parallel. An edge (u,v) in the DAG indicates that stage v depends on stage u, in the sense that it needs some data from u to complete its own computation. We will often use the terminology "logical" edge for (u,v), to distinguish such edges from the "physical" links that connect datacenters.
Stages and tasks. Each stage v contains n_v tasks. We assume that n_v is large which lets us use the fractional output of a Linear Program (LP) and incur small error. E.g., a fraction 0.1 of a stage translates to 10% of its tasks; we convert this to an integer and handle the (small) rounding errors with heuristics. In an examined data-parallel cluster at Microsoft consisting of tens of thousands of servers that ran SCOPE [10] jobs, the median stage had 50 tasks and 25% of the stages had over 250 tasks.
Data flow. We assume that each task reads (writes) some fraction of the input (output) of the stage. We make a simplifying assumption that these fractions are equal for tasks within a stage. For every stage v, we denote the ratio between the size of the output and the size of the input by s_v (its selectivity). Most stages output less data than their input; however about 10% generate up to $10\times$ more output [6]. Also given is c_v, the progress rate of a stage per core, per unit time. We assume that the progress rate is linearly proportional to the number of cores that the stage is given, up to a limit. For every edge $e=(u,v)$, we denote by D_e the total amount of data to be transferred between the stages u and v. For every stage v, we denote by $D_{\mathsf{IN},v}$ the total amount of input data for that stage; its total output data $D_{\mathsf{OUT},v}$ equals $s_v \times D_{\mathsf{IN},v}$. Further, $D_{\mathsf{IN},v} = \sum_{uv\in E} D_{uv}$, and $D_{\mathsf{OUT},u} = \sum_{uv\in E} D_{uv}$.
Compute datacenters. We assume that there are n datacenters numbered $1,2,3...,n$. For datacenter (DC) i, we denote the compute capacity by C_i, i.e., the number of cores. The DCs are connected via a physical network for which we will consider different possible topologies and link capacities in §2.4. Since the number of machines in a DC is large, we ignore potential machine fragmentation issues.

Dataset locations. For every source stage $v \in V$, its input data can be distributed across different DCs. The amount of input data for the stage v at DC i is denoted by $I_{i,v}$. For every sink node $v \in V$, it may be required that its output data be distributed across DCs. The amount of output data of node v that should reside in DC i is denoted by $O_{i,v}$, and is part of the input to the problem.

2.2 Stage dependency types

Consider two stages u and v connected via a logical edge $e = (u,v)$. Each stage is composed of many tasks. The modeling question is which tasks of stage v receive input data from which tasks of stage u. We define two types of dependencies: all-to-all and scatter-gather. The same DAG can have both types of dependencies.

Fig. 1: We illustrate here two DAG jobs. The left job has four stages with an *all-to-all* dependency type between u_1 and v_1. The right job is a chain of three stages with a *scatter-gather* dependency type between stages u_2 and v_2.



All-to-all dependency. A logical edge $e = (u,v)$ means that every task of stage u sends data to every task of stage v (see left part of Figure 1 for an illustration). Moreover, we assume that the same amount of data is sent between each pair of tasks (i.e., $D_e/n_u n_v$ per pair of tasks). This dependency typically arises during a *shuffle*; for example, in map-reduce, each task in the reduce stage is responsible for a partition and receives data corresponding to keys in that partition from every map task. In the examined production cluster, roughly 50% of the edges are all-to-all.
Scatter-gather dependency. A logical edge $e = (u,v)$ means that each task in stage u sends input to exactly one task in stage v or vice-versa (see right part of Figure 1). The former happens when $n_u \leq n_v$ and the latter happens when $n_u > n_v$. Note that this generalizes the One-to-One dependency. This dependency occurs when aggregations or joins are performed over partitioned data. In the examined production cluster, 50% of the edges are scatter-gather; about 16% are one-to-one.

2.3 Precedence models

Another important modeling aspect relates to the *precedence* between tasks in stages connected by an edge. We offer two models: soft-precedence and strict-precedence.
Soft-precedence. Here, we assume that each task can make fractional progress (say 1%) as long as every input-generating parent of that task has made at least an equivalent amount (1%) of progress. Such dependent tasks can execute simultaneously.
Strict-precedence. Here, we assume that a task can make no progress until all of its input-generating parent tasks in the DAG have finished completely.

We consider the above two models since they are extreme points of the design space. Note that a solution under the strict-precedence model is also valid under the soft-precedence model but the converse case does not hold. A typical setting in Hadoop launches reduce tasks after 80% of the map tasks have finished. In general, data-parallel frameworks allow overlapping to *pipeline* network transfers with task execution; however, overlapping adds to cost since both tasks simultaneously hold resources, such as memory.

2.4 Network topologies

In this work, we consider two types of network topologies.
General network topology. The most general topology can be modeled as follows. The n DCs are connected via a physical network of m nodes, where $m \geq n$. Nodes numbered $n+1 \dots m$ are relay nodes. Edges in the physical network (k, ℓ) have a corresponding maximum data transfer rate denoted by $B_{k,\ell}$. We assume that intra-DC transfer rate is unlimited, thus $B_{k,k} = \infty$.
Bus (or star) topology. Here, all of the DCs are connected to a bus (or to one hypothetical relay node). The bus has unbounded capacity and the bottlenecks are only in uploading and downloading data from a DC to the bus. For DC i, we denote the maximum upload and download data transfer rates by u_i and d_i respectively. With the advent of full-bisection datacenter backplanes [5, 7], this topology matches the network within a DC; the only bottlenecks are at the servers or in and out of *racks* of servers while the core is congestion-free. On the wide-area network, it remains a useful simplification; for example an actual physical topology can be approximated by choosing appropriate values for $\{u_i, d_i\}$.

3 Algorithms for Soft-Precedence Constraints

In this section we construct an LP representing the soft-precedence model. Then, we obtain an optimal solution for soft-precedence with scatter-gather stage dependency. Finally, we add multiplicative constraints to incorporate all-to-all stage dependency. In Section 5 we show a heuristic approach for dealing with these constraints. Given a value of time T as input, the LP examines whether the problem can complete within time T. The optimal value of T can be found, e.g., via binary search. The number of variables in the LP depends on T. To have data flow from input locations to the DCs where the first computation stages execute, we pad the graph with dummy stages for the input and for the output; we omit the specific details due to lack of space.

3.1 Computation and logical flow constraints

We define variables $x_{i,u,t}$ to denote the number of cores given to stage u on DC i, during time frame $[t,t+1)$. Since DC i has C_i cores, we have the following constraint: $\forall i \in [n], t \in [T] : \sum_{u \in V} x_{i,u,t} \leq C_i$. Note that the tasks in a stage may be constrained if needed by the maximum number of cores that they can use: $\forall i \in [n], t \in [T] : x_{i,u,t} \leq C_u$.

We define variables $r_{i,j,e,t}$ to represent the rate of data transfer from DC i to DC j, on edge $e=(u,v)$ during time frame $[t,t+1)$. Note that this definition refers to the *logical traffic demand* on the network due to a logical edge between stages; the data can transferred using any routes on the physical network.

We define variables $\mathsf{IN}_{j,v,t}$ to be the total amount of input data for stage v that reached DC j by time t. This amount is equal to the sum of all input data for v that was transferred into DC j by time t, including all logical edges entering v, which are denoted by $In(v)$. Data transfers should also include data transfers from the DC j to itself - if stage u finished on DC j, it can transfer its data to itself for processing stage v on the same data center j. These data transfers are unlimited in rate, $B_{j,j} = \infty$, but they should still be accounted for. As for the data that originates at j, it is denoted by $I_{j,v}$. So the total input data $\mathsf{IN}_{j,v,t}$ is given by:

$$\forall j \in [n], v \in V, t \in [T] \, : \, \mathsf{IN}_{j,v,t} = \sum_{i \in [n], e \in In(v), t' \leq t} r_{i,j,e,t'}.$$

We define variables $\mathsf{COMP}_{i,v,t}$ to be the total amount of output data for stage v that was computed in DC i by time t. This value is obtained by summing over all cores in i given to stage v over time:

$$\forall i \in [n], v \in V, t \in [T] \, : \, \mathsf{COMP}_{i,v,t} = \sum_{t' \leq t} x_{i,v,t'} \times c_v.$$

We define variables $\mathsf{OUT}_{i,v,t}$ to be the total amount of output data for stage v that was transferred away from DC i to the required destinations by time t, including all logical edges outgoing from v, which are denoted by $Out(v)$. This constraint is:

$$\forall i \in [n], v \in V, t \in [T] \, : \, \mathsf{OUT}_{i,v,t} = \sum_{j \in [n], e \in Out(v), t' \leq t} r_{i,j,e,t'}.$$

As mentioned earlier, for every stage v, DC i, time t - the amount of data computed and output COMP cannot exceed the amount of input data that has become available

IN. For instance, if only half of the input of a task has arrived by that time, then no more than half of the data could have been computed. More generally, the amount of data computed and output by v is bounded by the amount of input data that is available by time t times the selectivity of that stage (ratio of output to input), which is s_v. The constraint is therefore $\forall i \in [n], v \in V, t \in [T] : \mathsf{COMP}_{i,v,t} \leq \mathsf{IN}_{i,v,t} \times s_v$. Similarly, the amount of output data sent from the stage is bounded by the amount of data computed, yielding the constraint $\forall i \in [n], v \in V, t \in [T] : \mathsf{OUT}_{i,v,t} \leq \mathsf{COMP}_{i,v,t}$. We also demand that for every logical edge $e = (u, v)$, all necessary data be transferred. That is, the total amount of data transfers over time will be equal to D_e, yielding the constraint: $\forall e \in E : \sum_{i,j\in[n],t\in[T]} r_{i,j,e,t} = D_e$.

3.2 Physical flow constraints for general topology

We define variables $f_{i,j,e,k,l,t}$ to represent the *physical* rate of data transfer from physical node k to physical node ℓ during time frame $[t, t+1)$, to fulfill the demand of the logical flow $r_{i,j,e,t}$. That is, the *physical flow* is a way to transfer the logical traffic demands on the physical network, given network capacity constraints. Obviously, the two plans have to match. Every logical demand $r_{i,j,e,t}$ should equal the sum of the physical flows going out of i that implement it and to the sum of physical flows going into j, leading to the following constraints:

$$\forall i, j \in [n], e \in E, t \in [T] : r_{i,j,e,t} = \sum_{l\in[m]} f_{i,j,e,i,l,t} = \sum_{k\in[m]} f_{i,j,e,k,j,t}.$$

Additionally, corresponding to a logical flow from i to j, there should be no physical flow leaving j or entering i. The corresponding constraints are:

$$\forall i, j \in [n], k \in [m], e \in E, t \in [T] : f_{i,j,e,k,i,t} = f_{i,j,e,j,k,t} = 0.$$

The last constraints do not apply for $i = j = k$, in which case the only physical flow is from the DC to itself which we do allow.

For any physical node k, other than i, j, flow conservation dictates that the incoming flow to node k is equal to the outgoing flow from it. This is true for every high level flow $r_{i,j,e,t}$ separately. This leads us to the following constraints:

$$\forall i, j \in [n], k \in [m]/\{i, j\}, e \in E, t \in [T] : \sum_{\ell\in[m]} f_{i,j,e,k,\ell,t} = \sum_{\ell\in[m]} f_{i,j,e,\ell,k,t}.$$

Data transfer is also required to meet the maximum data transfer rate for every physical link $B_{k,\ell}$. This requirement implies the following constraint:

$$\forall k, \ell \in [m], t \in [T] : \sum_{i,j\in[n],e\in E} f_{i,j,e,k,\ell,t} \leq B_{k,\ell}.$$

3.3 Optimal solution for scatter-gather dependency

We now show that a solution to the LP implies an optimal schedule.

Theorem 1. *Consider the LP described in Sections 3.1–3.2 with the objective to minimize the total time. A solution to this LP is a near-optimal solution for the scatter-gather dependency model.*

Proof. We observe that any solution to the problem also defines a feasible solution to the LP, and thus the value achieved by the LP is a lower bound on the optimal value. Conversely, a solution to the LP can be translated into a solution for the problem; the only concern with this solution is the rounding errors; the impact of rounding errors can be minimized through a simple heuristic (we omit the details for brevity) and it is typically small since the number of tasks is very large. Hence, this solution is nearly optimal.

3.4 All-to-all dependency

We now show the multiplicative constraints needed for all-to-all stage dependency. We later present a heuristic approach for solving the program with these non-convex constraints. Consider a logical edge $e = (u, v)$. Say half of stage u is computed on DC i and the second half on j. Additionally, assume half of stage v is computed on DC k and half on ℓ. In this case, we need exactly 25% of the data to flow from each DC in $\{i, j\}$ to a DC in $\{k, l\}$. In general, the flow from i, u to k, v is proportional to the product $OUT_{i,u,T} \cdot IN_{j,v,T}$. Naively, the constraint should be:

$$\forall i, j \in [n], e = (u, v) \in E \,:\, \sum_{t \le T} r_{i,j,e,t} = D_e \frac{\mathsf{OUT}_{i,u,T}}{D_{\mathsf{OUT},u}} \cdot \frac{\mathsf{IN}_{j,v,T}}{D_{\mathsf{IN},v}}.$$

Note that $\frac{D_e}{D_{\mathsf{OUT},u} D_{\mathsf{IN},v}}$ is a constant, but the product $\mathsf{OUT}_{i,u,T} \cdot \mathsf{IN}_{j,v,T}$ contains two variables, and is thus non-convex. We use a first-order Taylor expansion to approximate this product. While this approach falls short of guaranteeing performance bounds, it obtains good results in practice. See Section 5 for details and a simulation study.

All-to-all dependency leads to one more complication: the progress of a stage may be limited by its slowest parent. Consider a logical edge $e = (u, v)$. Assume that stage u was scheduled half each on DC i, and DC j; it has completed 50% of its work at i but only 30% of its work at j. Then, stage v can complete no more than 30% of its work at *any* DC. This leads us to the following constraint:

$$\forall i, j \in [n], e = (u, v) \in E, t \in [T] \,:\, \frac{\sum_{t' \le t} r_{i,j,e,t'}}{\sum_{t' \le T} r_{i,j,e,t'}} \ge \frac{\mathsf{COMP}_{j,v,t}}{\mathsf{COMP}_{j,v,T}}.$$

That is, the fraction of the data sent is an upper bound for the fraction of the data computed of v, so that if only 30% was sent by time t, no more than 30% of v will be computed by time t. Multiplying both sides of the inequality by $\mathsf{COMP}_{j,v,T} \times \sum_{t' \leq T} r_{i,j,e,t'}$, we obtain yet another multiplication of variables. This product can also be approximated similarly with first-order Taylor expansion, see Sec. 5 for details.

4 Strict-Precedence

We now proceed to study the case of strict precedence constraints, which requires a different LP formulation. The solution to the new LP does not induce a schedule as in previous section, rather we need to construct a feasible schedule (satisfying strict precedences) from this solution. We present an approximation algorithm for all-to-all stage dependency, under a bus network topology. We conclude this section by highlighting the additional constraints needed for general network topology. These constraints contain multiplicative constraints that require a heuristic approach for solving them, which we describe in Sec. 5. We now present the LP for strict-precedence constraints. In this LP, the total makespan T is a variable, and the objective function is to minimize T.

4.1 Physical flow constraints for bus topology

For simplicity of exposition, we present the physical flow constraints for a bus topology. The constraints required for a general network topology are highlighted in 4.4. We define variables $d_{i,j,e}$ for $e = (u,v)$ to denote the total amount of data which we transfer from stage u on data center i to stage v on DC j. We define variables $t_{i,j,e}$ to denote the total amount of time it takes us to transfer that data. We define variables $r_{i,j,e,c}$ to denote the total amount of time in which we dedicate exactly c Mbps for the data transfer $d_{i,j,e}$. The total amount of data transferred is: $\forall i \neq j \in [n], e \in E \,:\, d_{i,j,e} = \sum_{c \in [min\{u_i,d_j\}]} c * r_{i,j,e,c}$.

As for the total transfer time $t_{i,j,e}$ - it will be the sum of times at different transfer speeds: $\forall i \neq j \in [n], e \in E \,:\, t_{i,j,e} = \sum_{c \in [min\{u_i,d_j\}]} r_{i,j,e,c}$.

We denote the time it takes to complete every logical edge e by t_e. We know that the time to complete every logical edge is lower bounded by the time it takes to transfer its data for any pair of DCs (i,j): $\forall i,j \in [n], e \in E \,:\, t_e \geq t_{i,j,e}$. If edge e takes time t_e then the total amount of data sent within that time frame from DC i cannot exceed $u_i t_e$. This leads us to the constraint: $\forall i \in [n], e \in E \,:\, \sum_{j \neq i, c \in [min\{u_i,d_j\}]} c * r_{i,j,e,c} \leq u_i t_e$.

Similarly, the total amount of data sent to DC j within t_e time cannot exceed $d_j t_e$. This leads us to the constraint: $\forall j \in [n], e \in E \,:\, \sum_{i \neq j, c \in [min\{u_i,d_j\}]} c r_{i,j,e,c} \leq d_j t_e$.

4.2 Computation and logical flow constraints

We define variables $x_{i,u,c}$ to denote the amount of time for which we dedicate exactly c cores of DC i to the computation of stage u. We define variables $d_{i,u}$ to denote the amount of output data of stage u to be found on DC i. With c_u being the output size per core per time unit, for stage u. The total amount of output data for stage u on DC i, $d_{i,u}$, is the sum of matching computations: $\forall i \in [n], u \in V \,:\, d_{i,u} = \sum_{c \in [C_i]} c c_u x_{i,u,c}$.

This amount is equal to the total amount of outgoing data, plus the amount of data there to be used as output. The corresponding constraint is: $\forall i \in [n], u \in V \,:\, d_{i,u} = \sum_{j \in [n], e \in Out(u)} d_{i,j,e} + O_{i,u}$. We used $Out(u)$ to denote the outgoing edges of u.

Similarly, the amount of data there, over the output to input ratio s_u, is equal to the amount of incoming data, plus the amount of data that was part of the job input there. The corresponding constraint is: $\forall j \in [n], v \in V \,:\, d_{j,v}/s_v = \sum_{i \in [n], e \in In(v)} d_{i,j,e} + I_{j,v}$. We used $In(v)$ to denote the incoming edges of v.

We define variables $t_{i,u}$ to denote the time it takes to compute stage u on DC i, and variables t_u to denote the total time it takes to compute the stage u over all DCs. It follows that: $\forall i \in [n], u \in V \,:\, t_{i,u} = \sum_{c \in [C_i]} x_{i,u,c}$, and $\forall i \in [n], u \in V \,:\, t_{i,u} \leq t_u$. With these constraints, t_u is the maximum time between all values $t_{i,u}$.

We also require that all computations is fully completed. Assume stage u requires a total of $D_{OUT,u}/c_u$ cores$\times$time, then: $\forall u \in V \,:\, \sum_{i \in [n]} d_{i,u} = D_u/c_u$. Similarly, we require that all data is indeed transferred. Assume edge e requires D_e data to be transferred, then $\forall e \in E \,:\, \sum_{i,j \in [n]} d_{i,j,e} = D_e$.

For every logical chain, i.e. sequence of edges $C \in G$ - we demand that the total computation and data transfer time of the chain, does not exceed T. This leads us to: $s.t. \forall C \in G \,:\, \sum_{u \in C} t_u + \sum_{e \in C} t_e \leq T$.

Note that for a general DAG, the number of chains can be exponential. However, these constraints can be rewritten in a way that their number becomes polynomial. In a nutshell, we can do so by introducing new variables $START_u, END_u$ for the start and end time of every stage u; we omit the details for brevity. We also demand that the total cores$\times$hours on every DC would not exceed its computation limits, that is: $\forall i \in [n] \,:\, \sum_{u \in V} \sum_{c \in [C_i]} c * x_{i,u,c} \leq TC_i$.

Additionally, we demand that if $I_{i,u}$ input data arrives at DC i for stage u, it will all be transferred out of it: $\forall i \in [n], u \in V \,:\, \sum_{j \in [n], e \in Out(u)} d_{i,j,e} = I_{i,u} s_u$, where we used $Out(u)$ to denote the outgoing edges of u. Similarly, if $O_{j,v}$ output data is demanded for stage v on DC j, then: $\forall j \in [n], v \in V \,:\, \sum_{i \in [n], e \in In(v)} d_{i,j,e} = O_{j,v}$, where we used $In(v)$ to denote the incoming edges of v.

4.3 The Algorithm

We now show how to turn the solution of the LP into an actual feasible schedule. We emphasize that the algorithm holds for general topology, although we can guarantee an approximation ratio only for bus topology, as we show later.

We say a stage u is available if all incoming data transfers for stage u have already completed. We say a logical edge $e=(u,v)$ is available if stage u has already completed. We use the LP variables $d_{i,u}$ to dictate the amount of data of stage u to be calculated on DC i. At every point in time, we denote by $k_{out,i}$ the number of logical edges currently available to be transferred from DC i, and by $k_{in,j}$ the number of logical edges currently available to be transferred into DC j. The algorithm is as follows:

1. While some stage has not completed:

2. For every DC i, denote by k the number of different stages currently available, having $d_{i,u} > 0$. Start running each of these stages on $\frac{C_i}{k}$ of the cores of DC i.

3. For every pair of DCs i,j, logical edge $e=(u,v)$, denote $d_{i,j,e} = d_{i,u} \cdot d_{j,v}$.

4. For every pair of DCs i,j, logical edge e, denote $c_{out,i,j,e} = \frac{u_i}{k_{out,i}} \frac{d_{i,j,e}}{\sum_{j'} d_{i,j',e}}$.

5. For every pair of DCs i,j, logical edge e, denote $c_{in,i,j,e} = \frac{d_j}{k_{in,j}} \frac{d_{i,j,e}}{\sum_{i'} d_{i',j,e}}$.

6. Start using $min(c_{out,i,j,e}, c_{in,i,j,e})$ Mbps for sending data from i to j associated with the logical edge e.

7. Continue running until some stage or data transfer completes, and then go back to step 1.

It follows directly from the definition of the algorithm that the solution produced is feasible; we omit a formal proof due to lack of space.

Analysis for bus topology. We now analyze the approximation factor of the algorithm with all-to-all stage dependency, under a bus network topology. For a general DAG, we define the width ω as the maximum between the maximum number of stages that can run in parallel (i.e., independent of each other), and the maximum number of edges that can run in parallel. We will show that our algorithm is within ω times of optimal. For the special case of a single chain DAG, $\omega = 1$; this includes the case of all map-reduce job DAGs; hence our algorithm is optimal for map-reduce jobs on a bus network– the predominant case considered by prior work (e.g., [22]).

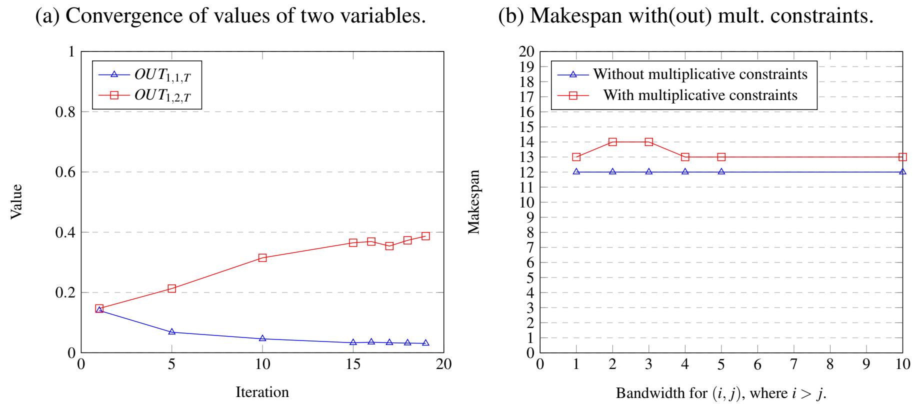
Theorem 2. *The algorithm is an ω-approximation for the strict-precedence model, with all-to-all stage dependencies, under a bus topology.*

Proof. Since every feasible solution to the problem is also a feasible solution for the LP, we know that T is a lower bound on the optimal execution time. We show that ωT is an upper bound for the execution time of our solution, and from this obtain the desired approximation factor.

In a general DAG of width ω, we might have up to ω stages executing in parallel, in the worst case. This means that ω stages are competing over the same DC i. Thus, each will get at least $\frac{C_i}{\omega}$ cores, and will complete within at most $\omega t_{i,u}$ time. Therefore, every stage u will complete within at most ωt_u.

We know from the LP that $\forall i \in [n], e \in E : \sum_{j \neq i, c \in [min\{u_i, d_j\}]} c * r_{i,j,e,c} \leq u_i t_e$, and that $\forall j \in [n], e \in E : \sum_{i \neq j, c \in [min\{u_i, d_j\}]} c * r_{i,j,e,c} \leq d_j t_e$. For every logical edge e we assume at least one of these constraints is tight, otherwise we can lower the value of t_e. Assume w.l.o.g. that this constraint is for an upload link, and some DC i. The total data that needs to be transferred from DC i, regarding e is: $u_i t_e$. We know there are at

Fig. 2: Performance of the iterative approximation heuristic.



most ω logical edges that run in parallel with e, and thus $k_{out,i} \leq \omega$. Since we give a total of $\frac{u_i}{k_{out,i}} \geq \frac{u_i}{\omega}$ to the edge e, we know it will complete within at most ωt_e time.

We know from the LP that $\forall C \in G : \sum_{u \in C} t_u + \sum_{e \in C} t_e \leq T$. The total execution time for every chain $C \in G$ will not exceed $\sum_{u \in C} \omega t_u + \sum_{e \in C} \omega t_e \leq \omega T$. Since every chain completes within ωT time, the entire DAG finishes in ωT time. Hence, for a general DAG, we guarantee an approximation factor of ω.

4.4 *Physical flow constraints for a general network topology*

For a general network topology, we need to apply different physical constraints. In an all-to-all scenario, these constraints also include multiplicative constraints. In a bus topology we were able to avoid this issue, but in a general topology, the flow has different paths, and we need the LP to know exactly how much flow goes between every pair of DCs i, j. In the full version of the paper we define the full set of constraints. These constraints, together with the previous ones lead to an LP formulation. This LP is then solved iteratively as described in Section 5. The LP solution allows us to use the algorithm presented in Section 4.3 as a heuristic solution to the general network model.

5 Heuristics

In this section, we first describe how we deal with the non-convex constraints which we obtain for the All-to-All dependency model. We then perform basic simulations to demonstrate that "linearizing" these constraints works well in practice.
Linearizing multiplicative constraints. Recall that our multiplicative constraints are of the form: $\forall i, j \in [n], e = (u,v) \in E : \sum_{t \leq T} r_{i,j,e,t} = D_e \frac{\mathsf{OUT}_{i,u,T}}{D_{\mathsf{OUT},u}} \cdot \frac{\mathsf{IN}_{j,v,T}}{D_{\mathsf{IN},v}}$.

The Taylor series is expanded around *estimated* values for $\mathsf{OUT}_{i,u,T}$ and $\mathsf{IN}_{j,v,T}$, denoted $\widehat{\mathsf{OUT}}_{i,u,T}$ and $\widehat{\mathsf{IN}}_{j,v,T}$ respectively. We solve the LP iteratively, and use the values that the LP found for the variables in the previous iteration as the estimated values for the next iteration. The stopping condition for this procedure is when the difference in consecutive values is smaller than a configurable parameter (or when exceeding a maximal number of iterations). This iterative procedure is described in detail in Appendix 6. We next show empirically that the estimations converge quickly, and further that the corresponding values lead to adequate performance.
Simulations. Our goal is twofold: (i) demonstrate that the iterative approach for approximating the multiplication constraints indeed converges quickly to a feasible solution. (ii) show that the obtained solution is "good". In particular, albeit using approximations, we would like to demonstrate convergence to near-optimal values.
Setting. In our simulations, we use a chain DAG, consisting of ten stages with different initial data distribution and computation requirements. We assume soft-precedence between stages, and all-to-all dependency. The network topology is a clique, i.e., there is a link between every two datacenters. In each run, we vary the bandwidths of the links, as we elaborate below.
Convergence. Figure 2a shows the convergence of the variables $\mathsf{OUT}_{i,u,T}$ for $i = 1$ and $u = 1,2$. As can be seen, we obtain rather stable values after 15 iterations. We note that we obtain similar convergence behavior for other variables, and also for other DAGs that we have tested. The significance of convergence is that our scheme stabilizes, and we can rely on the obtained values for the multiplication of two variables. This does not imply that we converge to "good" values of the variables. We next address the quality of the approximation.
Quality of the approximation. We check the obtained makespan with and without the multiplicative constraints. Any feasible solution to the actual problem must satisfy all the constraints, including the multiplicative constraints. Accordingly, the optimal solution that satisfies the constraints excluding the multiplicative ones, is obviously a lower bound for the optimum. We next show empirically that the value of the execution time with the multiplicative constraints is indeed very close to the value without them, thus close to the optimal value. We again use a chain DAG with ten stages. The physical network is a clique. For each link (i, j), we assign a bandwidth of one if $i < j$, and for each run, choose a different value out of the set $\{1,2,3,4,5,10\}$ for the remaining links (i.e., (i, j) s.t. $i > j$). Figure 2b shows that the execution time without the multiplicative constraints is always 12 time units. Adding the multiplicative constraints increased the execution time to $13 - 14$ units in all of the runs (i.e., an $8 - 17\%$ increase in the makespan). This indicates that the

potential loss due to the approximation of multiplications is not substantial. Note that the makespan of our algorithm is non necessarily monotone in the bandwidth assigned to the $i > j$ links – indicating that our approximation is sub-optimal. However, what matters most here is that the obtained makespan is close to optimal.

References

1. Hadoop YARN Project. `http://bit.ly/1iS8xvP`.
2. Seattle department of transportation live traffic videos. `http://web6.seattle.gov/travelers/`.
3. TPC-H Benchmark. `http://bit.ly/1KRK5gl`.
4. TPC-DS Benchmark. `http://bit.ly/1J6uDap`, 2012.
5. A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.
6. Sameer Agarwal, Srikanth Kandula, Nico Burno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Re-optimizing data parallel computing. In *NSDI*, 2012.
7. Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.
8. Michael Armbrust et al. Spark sql: Relational data processing in spark. In *SIGMOD*, 2015.
9. Peter Bodík, Ishai Menache, Joseph Seffi Naor, and Jonathan Yaniv. Brief announcement: deadline-aware scheduling of big-data processing jobs. In *SPAA*, pages 211–213, 2014.
10. Ronnie Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.
11. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.
12. Pierre-François Dutot, Grégory Mounié, and Denis Trystram. Scheduling parallel tasks approximation algorithms. In *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*. 2004.
13. Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 1969.
14. Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.
15. Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *SIGCOMM*, 2013.
16. Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-area analytics with multiple resources. In *EuroSys*, 2018.
17. Chien-Chun Hung, Leana Golubchik, and Minlan Yu. Scheduling jobs across geo-distributed datacenters. In *SOCC*, 2015.
18. IDC. Network video surveillance: Addressing storage challenges. `http://bit.ly/1OGOtzA`, 2012.
19. Michael Isard. Autopilot: Automatic Data Center Management. *OSR*, 41(2), 2007.
20. Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.
21. Klaus Jansen and Hu Zhang. Scheduling malleable tasks with precedence constraints. *J. Comput. Syst. Sci.*, 78(1):245–259, 2012.
22. Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed analytics. In *SIGCOMM*, 2015.
23. Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

24. Ashish Thusoo et al. Hive- a warehousing solution over a map-reduce framework. In *VLDB*, 2009.
25. Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.
26. M. Zaharia et al. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, 2010.

6 Appendix: Linearizing Multiplicative Constraints

Recall that our multiplicative constraints are of the form

$$\forall i,j \in [n], e=(u,v) \in E \, : \, \sum_{t \leq T} r_{i,j,e,t} = D_e \frac{OUT_{i,u,T}}{D_{OUT,u}} \cdot \frac{IN_{j,v,T}}{D_{IN,v}}$$

The Taylor series is expanded around the estimated values $\widehat{OUT}_{i,u,T}, \widehat{IN}_{j,v,T}$. We solve the LP iteratively, and use the values the LP found for the variables in the previous iteration as the estimated values. After several iterations the values converge to their true value, and thus the multiplications become more accurate.

Recall that the first-order Taylor expansion for the multiplication $x \cdot y$, expanded around the point $(\hat{x},\hat{y})$ is: $x \cdot y \simeq \hat{x} \cdot \hat{y} + (x-\hat{x}) \cdot \hat{y} + (y-\hat{y}) \cdot \hat{x}$. Dividing both sides by the constant $\frac{D_e}{D_{OUT,u} D_{IN,v}}$, and approximating the multiplication using first-order Taylor, we obtain $\forall i,j \in [n], e=(u,v) \in E$:

$$\frac{D_{OUT,u} D_{IN,v}}{D_e} \sum_{t \leq T} r_{i,j,e,t} \geq \widehat{OUT}_{i,u,T} \cdot \widehat{IN}_{j,v,T} + (OUT_{i,u,T} - \widehat{OUT}_{i,u,T}) \cdot \widehat{IN}_{j,v,T} + (IN_{j,v,T} - \widehat{IN}_{j,v,T}) \cdot \widehat{OUT}_{i,u,T}.$$

Note that we have turned this equality constraint into a non-equality, since we have other constraints for the total flow from previous sections. Our multiplication evaluating the flow might be slightly more or less than the true multiplication value. In case it is more than the true value - we simply send less flow, and no constraints are violated. In case it is less than the true value - we need to send more data than planned - which might violate capacity constraints. In this case - we simply use a little more time for the entire flow to be sent. As long as the approximation is reasonable - this extra-time will be small.

To obtain relatively accurate values for $\widehat{OUT}_{i,u,T}, \widehat{IN}_{j,v,T}$, we solve the LP iteratively, and use the previous values as our approximation. For the first iteration only we use $\widehat{OUT}_{i,u,T} = 0, \widehat{IN}_{j,v,T} = 0$. We use the same approach for our second set of multiplicative constraints (see Section 3.4) $\forall i,j \in [n], e=(u,v) \in E, t \in [T]$: $\frac{\sum_{t' \leq t} r_{i,j,e,t'}}{\sum_{t' \leq T} r_{i,j,e,t'}} \geq \frac{COMP_{j,v,t}}{COMP_{j,v,T}}$; details omitted for brevity. The resulting LP is then solved iteratively, as described, to obtain a nearly feasible solution. The small infeasibility translates into some extra-time required for completing flows that are larger than anticipated by the LP.