Bi-objective traffic optimization in geo-distributed data flows

Anna-Valentini Michailidou[a], Anastasios Gounaris[a]

[a]*Department of Informatics, Aristotle University of Thessaloniki, Greece*
{annavalen,gounaria}@csd.auth.gr

Abstract

Recently, there have been several proposals in the area of geo-distributed big data processing. In this work, we aim to address a limitation of the existing solutions, namely to optimize task allocation across geographically distributed data centers, in a way that both the total traffic and the running time of the whole processing in complex multi-stage flows are targeted. Apart from proposing concrete efficient solutions for this combinatorial problem, we advocate to take a critical stand on the broadly spread claim that transferring distributed data to a single or fewer places is too costly. In our proposal, we judiciously reduce the participation of some data centers in the flow execution, and we show that, in a wide range of settings, this yields significant benefits. We show that a stochastic solution is superior to a fast greedy one, at the expense of optimization time of up to a few minutes. Compared to a state-of-the-art solution, we manage to decrease total traffic by 44% and running time by 37% on average. In several cases, the improvements can reach 1-2 orders of magnitude. Moreover, we provide evidence that simple heuristics are inferior. Our experimental evaluation comprises both extensive simulations and real runs in Spark.

Keywords: distributed flow optimization, latency minimization, communication minimization, Iridium

1. Introduction

Data-intensive processing platforms is still a technology in evolution. The emergence of the MapReduce framework and its descendants, such as Spark and Flink, has shaped the way in which big data is nowadays processed and analyzed. However, these technologies are tailored to a single data center, since they implicitly assume either a powerful multi-core server or a

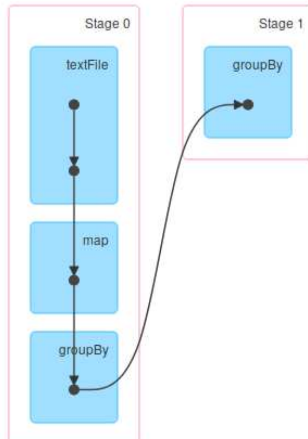


Figure 1: Example DAG of stages of a groupBy transformation

shared-nothing cluster, where the communication between the nodes is very fast. At the same time, the need for analyzing data from geo-distributed locations is rapidly increasing in several domains, which gives rise to the need to adjust and extend the big data processing technologies in order to become applicable across geo-distributed data centers in an efficient manner.

In the recent years, there have been several proposals that aim to address the problem of transferring MapReduce-like solutions to a geo-distributed setting, but they suffer from several limitations. According to a recent survey in [1], a significant such limitation is the lack of solutions that aim to optimize both the total traffic and the flow completion time in a combined manner. For example, there exist proposals that minimize the completion time, e.g., [2], or the total traffic, e.g., [3], but not both. Our work aims to fill this gap.

More specifically, in this paper we focus on the bi-objective traffic optimization problem in data flows, where we aim to reduce the total traffic of the network while keeping the running time under a given threshold set by the user (explained in Section 4.1). We use directed acyclic graphs (DAGs) to represent data flows. Each node represents a job (for example, in Spark each node refers to a Spark Stage) that needs to be parallel executed on different data centers (DCs) while the edges of the graph represent the data movement between the jobs. For example, Figure 1 shows a DAG in Spark where each node is a stage and the edge between the stages represents the data movement that occurs due to the groupBy transformation. Such data movements incur traffic cost and may contribute to the total running time.

The goal is, given an arbitrary initial distribution of the data, to distribute the workload among the DCs for each job, so that both objectives are met. A naive solution is clearly exponential in the number of the jobs.

The main characteristics of our approach are twofold. On the one hand, we exploit existing solutions to the largest possible extent. We use the state-of-the-art Iridium solution in [2] for minimizing running time and we extend it in two ways: (i) to make it more efficient for multi-job DAGs instead of simple two-stage MapReduce ones and (ii) to account for total traffic as well. On the other hand, we challenge the validity of a main motivation behind geo-distributed data flows, namely that it is too costly to gather data in a single place, e.g., [1, 4, 5]. We develop a fast Greedy solution and a more efficient but slower stochastic process that both aim to investigate task allocations that either decrease or discard the participation of some DCs to some jobs.

In summary, we make the following four contributions, whereas we also implemented our solution in Spark:

1. We show that using less DCs is more beneficial than using all the DCs available in a wide range of cases; the benefits are both in total traffic and flow execution time.

2. We propose two algorithms, a greedy and a stochastic one, that decrease the total traffic by re-arranging the task placement between the DCs, staying always under a given threshold in running time.

3. We conduct thorough experiments using realistic parameters. We show that our stochastic solution achieves an average of 37% reduction in the total running time while reducing traffic by 44% on average compared to the proposal in [2]. The improvements are higher if a hybrid initialization scheme is followed. In some cases, the improvement margins are 1-2 orders of magnitude. This is at the expense of optimization times of up to a few minutes.

4. We show that, similarly to using all available DCs, naive solutions, such as using a single DC for all job executions, are inferior to our techniques that more judiciously restrict the participation of all DCs in the flow execution.

The remainder of the paper is outlined as follows. In Section 2, we provide two motivation examples to show the challenges in our bi-objective optimization problem and the benefits from using less DCs. The related

Table 1: Parameters for the second motivation example

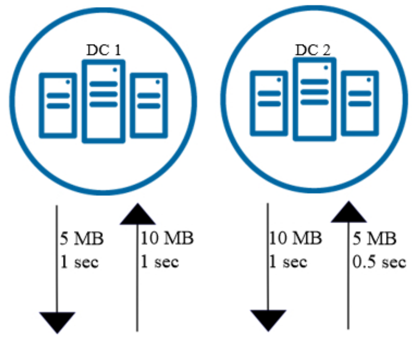
	DC1	**DC2**	**DC3**	**DC4**
$InputData(MB)$	50	60	40	75
$Uplink(MB/s)$	10	1	10	10
$Downlink(MB/s)$	1	10	1	10

work is discussed in Section 3. Our proposal and our algorithms are described in Section 4. In Section 5, we present our experiments and their results. Finally, in Section 6, we make a final discussion.

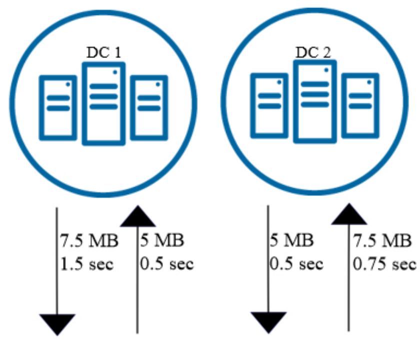
2. Motivation Example

Assume that we have a simple two stage graph running on two DCs, DC1 and DC2. That means that each stage of the graph is executed on both DCs in parallel. The overall size of data is 30MB. Initially, the data of the first stage are divided to the DCs as follows: 10 MB on DC1, and 20 MB on DC2. The uplink and downlink of DC1 (resp. DC2) are 5 MB/sec and 10 MB/sec (resp. 10 MB/sec and 10 MB/sec). The first allocation equally splits the workload for the second stage. This means that DC1 has to upload 5 MB and DC2 has to upload 10 MB. Also, DC1 downloads 10MBs and DC2 downloads 5 MB. Since data transmissions run in parallel, the running time is determined by the slowest link. As such, the total running time is 1 sec and the total traffic is 15MB, as can be seen in Figure 2a. However, there is a better configuration, in which the first DC has to upload 7.5 MB and the second 5 MB only; this is achieved by allocating 25% of tasks of the second stage on DC1 and 75% on DC2. This results in total running time of 1.5 sec and total movement of 12.5MB. This means that, in general, traffic time and running time can be contradicting objectives, thus it requires particular attention when trying to optimize both.

Now let's consider a more complex case, where 4 DCs participate (the key parameters are shown in Table 1). If we assign an equal fraction of tasks in each DC, then each DC has to upload and download some data. The time for these moves depends on the DC's uplink and downlink speed. The total running time of the job is the maximum of all the link finish times, as previously. In our example, the total running time is 46.25 secs and the total amount of data moved is 168.75 MB. This time is due to the bottleneck DC, DC3, the downlink speed of which is 1 MB/sec and the amount of data it has to download is 0.25*185=46.25 MB. If we assign the task allocation to



(a) Equal task allocation (50%-50%)



(b) Unequal task allocation (25%-75%)

Figure 2: Data movement costs and times for the first motivation example

Iridium [2], the new allocations are 0%, 78.5%, 0%, and 21.5%, respectively. This means that DC1 and DC3 do not execute any tasks at the second flow stage and therefore, they do not download any data; however, they need to distribute their data to the other DCs that have non-zero allocation. This scheme reduces time to only 12.95 secs and data movement to 161.78 MB. This example shows that it is possible to improve regarding both the running time and the total data movement by not using all DCs throughout the execution.

Our work can be seen as an extension to Iridium. In our extensions, both objectives are targeted more systematically and efficiently, whereas we also target more complex flows, i.e., flows with multiple stages. As shown in Section 5, our techniques are more efficient than Iridium in both metrics. We continue the discussion of the related work in the next section.

Table 2: Summary of the most relevant related work (*) Not both metrics simultaneously; (**) tailored to SQL queries; (***) tailored to graph processing.

Proposal	**Aggregate traffic reduction**	**Running time reduction**
WANalytics[3]	✓	
Shuffle Optimazation[6]	✓	
Pixida[4]	✓	
G-MR[7]	✓(*)	✓(*)
Meta-MapReduce[8]	✓	
Jetstream[9]	✓	
Geode[5]	✓(**)	✓(**)
Iridium[2]		✓
Nebula[10]		✓
G-Cut[11]	✓(***)	✓(***)
Rout[7]		✓
THIS WORK	✓	✓

3. Related Work

An extensive survey of geo-distributed extensions to MapReduce has appeared in [1]. The most relevant research to our work consists of proposals that target either the total traffic or the flow running time, summarized in Table 2.

Regarding the works that target the total traffic, a prominent example is the WANalytics proposal in [3]. Similar to our work, it targets generic DAGs but does not consider running time. [6] focuses also on data traffic and suggests a Spark-based framework that targets the best placement of data, but does not set an upper threshold on running time. Pixida[4] converts the data traffic reduction problem to a graph repartitioning one. G-MR[7] is a Hadoop-based framework that detects the best task placement through solving a shortest path problem considering either running time or data movement, but not both; moreover its complexity is exponential in the number of nodes.

There are also some more specific solutions. For example, Meta-MapReduce[8] focuses on decreasing the data movement by avoiding to move data that will not participate in the final output. Jetstream[9] decreases the data traffic using degradation, aggregation and filtering operations on the data at the expense of lower accuracy. Geode[5] targets both data traffic and running time reduction in SQL queries using caching and copies of the data, but the proposed technique cannot be extended to generic DAGs.

Regarding the proposals that target running time, we have already introduced Iridium [2], against which we directly compare our proposal. Iridium has been compared against [3], and is found that it achieves much lower times at the expense of small increases in total traffic; on the contrary, we show that we can improve both metrics. However, Iridium also deals with the problem of initial data placement, whereas we assume that this placement is fixed. Nebula[10] tries to find the optimal task allocation to minimize the running time but does not consider the overall data traffic. Rout[7] also tries to minimize the running time by selecting the most beneficial data placement but does not focus on data traffic either. G-Cut[11] aims to minimize the running time re-arranging the tasks while keeping the traffic below a threshold, but it is specific to geo-distributed graph processing. The proposal in [12] targets both metrics but is tailored to a single MapReduce flow with the reducer being executed on a single DC.

Note that there are several proposals in geo-distributed processing that focus on other issues than performance; for example, [13] focuses on resilience, and [14] deals with DC configuration. Other topics that are tangential to our research involve understanding of wide-area data transfer performance, e.g., [15], flow modeling and performance prediction, e.g., [16], accesses to geo-distributed Web Services, e.g., [17], and advances in edge computing, e.g., [18].

Finally, our stochastic solution has been inspired by a randomized solution for determining the appropriate degree of parallelism in Spark flows, where it was proven that such stochastic solutions are a powerful tool for finding local optimal solutions in bi-objective problems [19]. Here, we capitalize on this experience and we successfully apply a stochastic solution to a new setting.

4. Our proposal

Our proposal aims to tackle the two main limitations of existing works: (i) it considers generic DAG flows in a more comprehensive and efficient manner through taking into consideration the impact of the decisions taken regarding a specific stage of the flow on the other ones; and (ii) it aims to decrease both the overall running time and the total communication across inter-DC connections. The rationale behind the design of the techniques is that, in several scenarios, it is more beneficial to allocate (the largest part of) the workload to less DCs than those initially holding the data.

We make three salient assumptions: (i) as in [2], the inter-DC communication cost is the dominant one; (ii) a limited number of DCs is adequate to perform the main processing corresponding to a DAG vertex; and (iii) the initial distribution of input data across DCs is fixed.

4.1. Notation and Problem Definition

A geo-distributed data flow is represented as a DAG $G(V, E)$. Each node $v_j \in V$, where $j = 1 \dots N$ and $N = |V|$, represents a *job* and each edge represents a shuffle data movement between the jobs. For example, in Spark data flows, a job corresponds to a *Spark stage*; in between such stages, data shuffling takes place. Each job runs in parallel in M DCs: i.e., each DC becomes responsible for a fraction of the job execution with the magnitude of the fraction devised by our algorithms. Conceptually, the workload of a job is split into small units of work, each allocated to a specific processing element, e.g., a multi-core server of a specific DC, as an atomic unit. We refer to these splits as *tasks*. Due to shuffling, in the generic case, it is necessary to move data between DCs before the execution of each task. This data movement is the dominant factor regarding the running time of the jobs, while the actual execution time of the job is considered to be negligible.

In this work we deal with the allocation of sets of tasks to each DC for each job. Let I^j be the input dataset size of v_j. If the selectivity of the job is a^j, then the output dataset is of size $S^j = a^j * I^j$. If v_j has outgoing edges in G, S^j is divided into M parts to be sent to the jobs downstream, denoted by $r_i^j S^j$, $i = 1 \dots M$, s.t. $\sum r_i^j = 1$. Essentially, r_i^j corresponds to the fraction of tasks of the *children nodes* of v_j assigned to the i^{th} DC (tasks are assumed to be infinitesimally divisible). In other words, r_i^j values affect the workload allocation of jobs v_k, where $(j, k) \in E$. Overall, each DC has to transfer a fraction of $(1 - r_i^j)$ of its local output data S_i^j, and to receive a total of $r_i^j * (S^j - S_i^j)$ data from all the other DCs.[1] Following the rationale in [2], we specify the uplink (resp. downlink) bandwidth of the i^{th} DC as U_i (resp. D_i). Table 3 summarizes the main notation.

Based on the above, the time for a site to send data regarding the output of a job is $TU_i^j = (1 - r_i^j) * S_i^j / U_i$, and the time to receive data is $TD_i^j = r_i^j * (S^j - S_i^j)/D_i$. The running time RT_j of v_j is $max\{TU_i^j, TD_i^j\}$.

The total data movement from a node v_j is equal to $DM_j = \sum_{i=1}^{M}(1 - r_i^j) * S_i^j$. The total data movement is $DM(G) = \sum_{j=1}^{N} DM_j$, where v_j has

[1] Note that in general, $S_i^j \neq r_i^j S^j$, i.e., the distribution of the intermediate results in a job is not necessarily the same as the way these results are shuffled in the next jobs. However, assuming a uniform distribution of results, it holds that $S_i^j = mean(r_i^k) * S^j$, where $(k, j) \in E$

Table 3: Notations used in the paper.

Symbol	**Meaning**
$G(V,E)$	the data flow DAG
N, M	number of jobs and DCs
I^j	amount of input data of a job $v_j \in V$
α^j	selectivity of a job v_j
S^j	amount of intermediate output data of a job ($S^j = a^j * I^j$)
U_i	uplink bandwidth on DC i
D_i	downlink bandwidth on DC i
S_i^j	amount of intermediate data of v_j on DC i
r_i^j	fraction of tasks executed on DC i for jobs succeeding v_j
TU_i^j, TD_i^j	running time of intermediate data transfer on up and down link of DC i
$RT(G)$	total running time of G
$DM(G)$	total data movement between DCs in G
RT_j	running time of job v_j
DM_j	total data movement between DCs of job v_j
allocations	A $N \times M$ array holding in each row $allocations[j]$ the $r_i^j,\ j = 1 \dots N,\ i = 1 \dots M$ values

at least one outgoing edge.

The running time of a G, $RT(G)$ is the maximum sum of RT_j values across any path from a source job (v_j without incoming edges) to a sink one (v_j without outgoing edges); sink nodes have zero running time by default.

In this paper we aim to the bi-objective traffic optimization of the data flow. That means that we try to minimize the total data movement considering the execution time of G. At a nutshell, we follow a two-step approach:

1. We use Iridium[2] as our guideline for the initial assignment of tasks, i.e., computation of the r_i^j values, to the DCs. Iridium decides the allocation for each job separately, after performing a topological sorting on G and considers the nodes from the upstream to the downstream ones.

2. We re-arrange the allocations with a view to decreasing the total movement cost while not allowing running time degradation more than ε.

More formally, the problem we target is defined as follows:

Problem Statement: Given a dataflow G, a fixed distribution of the initial data across M DCs, and a running time value $RTbase$, compute the r_i^j values s.t. $DM(G)$ is minimized and $RT(G)$ is always less than $(1+\varepsilon)RTbase$, where ε is a small constant $\varepsilon > -1$. If $0 > \varepsilon > -1$, then we enforce the solutions to seek improvements regarding both $DM(G)$ and $RT(G)$; when ε is positive, we tolerate increases in $RT(G)$.[2]

Note that the higher we set ε, the more the problem tends to be a single-objective optimization (that of minimizing $DM(G)$) in practice.

In our solution, the first step derives the $RTbase$ value from the result of the Iridium solution. Then, our main contribution refers to the second step, for which we propose two techniques, a stochastic and a greedy one.

4.2. A greedy solution

The first solution is a greedy algorithm that is described in Algorithm 1. The input of the algorithm is (i) the initial allocation of tasks on the DCs according to [2], (ii) the threshold $RTthreshold = (1+\varepsilon)RTbase$, where $RTbase$ is the initial $RT(G)$, and (iii) the initial $DM(G)$. The output is the new allocation of tasks optimized for lower $DM(G)$ with the new $RT(G)$ to be less than the threshold.

Algorithm 1 consists of two loops. The internal one iterates over all jobs. Its rationale is that the DC with the smallest non-zero r_i^j acts as a bottleneck, and its fractions of tasks should be further decreased by an β factor (by default set to 1/3). So the algorithm detects the bottleneck DC for each job, decreases its r_i^j by β, and distributes the removed workload to all the remaining DCs proportionally to their own current portions (inactive DCs remain with zero allocation). After we exit the internal loop, we choose the most beneficial modification of the task allocation referring to a single job in terms of $DM(G)$ whose $RT(G)$ is under the threshold. This reallocation triggers reallocations to all the other nodes downstream, since the data of the downstream nodes are re-divided (S^j is re arranged to the DCs); the reallocation is computed through a LP solution as in [2]. This process is repeated $N * M$ times.[3]

The complexity of the technique loop is dominated by solving the linear program with M variables of [2] for all the affected jobs, which are $O(N)$.

[2] We can also regard positive values of ε as the percentage of the performance degradation that is tolerated.

[3] We can easily modify the number of iterations, but in practice, even if we increase it, the performance does not improve compared to the next stochastic solution.

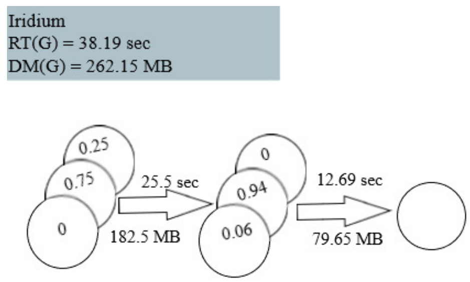
Algorithm 1 Greedy algorithm

Require: $allocations, RTthreshold, DM(G)$
  for $i \leftarrow 1$ to $N * M$ **do**
    $best \leftarrow allocations$ //holds the best reallocation for all the jobs
    $bestRT \leftarrow$ Calculate $RT(G)$ using $allocations$
    $bestDM \leftarrow DM(G)$
    for $j \leftarrow 1$ to N **do**
      Reallocate tasks by reducing bottleneck's fraction by β
      Calculate $DM(G)$ reduction due to modifications in Node j
    end for
    Choose the local change with the highest local $DM(G)$ reduction meeting the $RTthreshold$
    $tempAllocations \leftarrow$ apply changes to G
    Calculate $RT(G)'$ using $tempAllocations$
    Calculate $DM(G)'$ using $tempAllocations$
    if $RT(G)' \leq RTthreshold$ **then**
      $benefit \leftarrow bestDM - DM(G)'$
      if $benefit > 0$ **then**
        $best \leftarrow tempAllocations$
        $bestRT \leftarrow RT(G)'$
        $bestDM \leftarrow DM(G)'$
      end if
    end if
  end for
  return $best$, $bestRT$, $bestDM$

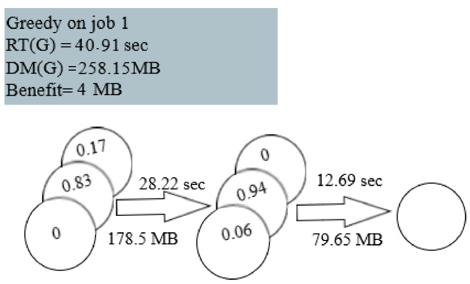
Therefore, the total complexity is solving an LP program $O(N^2M)$ times. The number of reallocations considered is also in $O(N^2M)$.

4.2.1. Example

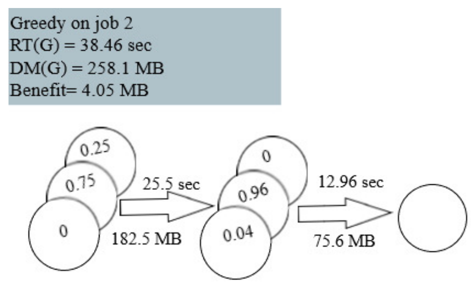
Assume that we have a linear DAG G with three nodes running on three DCs as can be seen in Figure 3; each node is being executed in all the DCs in parallel. In the figure, each circle corresponds to a job-DC pair annotated by the corresponding r_i^j value. The uplink and downlink of the DCs are U=(10, 1, 10), D=(10, 5, 5). The S_i^1 values are S_i^1=(120, 100, 50) and α =1 for both jobs. Figure 3a shows the result of Iridium, which decides the allocation of r_i^1=(0, 0.75, 0.25) regarding the results of the first job. Consequently, S_i^2 becomes $(0, 202.5, 67.5)$. Iridium then assigns the fractions r_i^2=(0.06, 0.94, 0) thus, $RT(G) = 38.19$ sec and $DM(G) = 262.15$ MB (Figure 3a).



(a) Iridium's task allocation



(b) Greedy re-allocation regarding the first job



(c) Greedy re-allocation regarding the second job

Figure 3: Example using the Greedy algorithm

Now, let us suppose that we set $RTthreshold = (1 + 0.1)38.19 = 42$ secs and we execute the external loop of Algorithm 1 a single time. First, we check the first job, for which the 3rd DC forms a bottleneck. We remove 1/3 of its workload and we transfer it to the 2nd DC, which is the only other DC with non-zero allocation; this results in r_i^1=(0, 0.83, 0.17). The new $RT(G)$ is 40.91 sec. The benefit in the data movement is 4 MB (Figure 3b). Then, Greedy proceeds to the second job and reallocates its tasks as follows: r_i^2=(0.04, 0.96, 0) (the bottleneck DC is the first one) and the metrics become $RT(G) = 38.46$ sec and $DM(G) = 258.1$ MB. The new running time is under the threshold and the benefit for this second re-allocation is 4.05 MB (Figure 3c). As a final result of this iteration, the most beneficial reallocation is the second one and it is accepted; if it had any downstream nodes, we would rerun LP for these nodes.

Overall, in a single iteration, even in a very simple graph, we managed to improve upon Iridium by 1.5% in terms of $DM(G)$ at the expense of 0.7% increase in $RT(G)$.

Algorithm 2 Iterated Local Search algorithm

Require: $allocations, RTthreshold, DM(G)$
  Calculate $DM(G)$ using $allocations$
  Calculate $RT(G)$ using $allocations$
  $tempAllocations \leftarrow SHC(tempAllocations, RTthreshold, DM(G))$
  for $i \leftarrow 1$ to $iter1$ **do**
    $tempAllocations \leftarrow perturbation(allocations)$
    $tempAllocations \leftarrow SHC(tempAllocations, RTthreshold, DM(G))$
    Calculate $RT(G)'$ using $tempAllocations$
    Calculate $DM(G)'$ using $tempAllocations$
    if $RT(G)' \leq RTthreshold$ **then**
      if $DM(G)' \leq DM(G)$ **then**
        $allocations \leftarrow tempAllocations$
        $DM(G) \leftarrow DM(G)'$
        $RT(G) \leftarrow RT(G)'$
      end if
    end if
  end for
  return $allocations$, $RT(G), DM(G)$

4.3. A stochastic solution

Our stochastic solution is an Iterated Local Search (ILS) algorithm that uses Stochastic Hill Climbing (SHC) as an internal heuristic mechanism. The algorithm is described in Algorithm 2. The input of the algorithm is the initial allocation of tasks on the DCs and the threshold on the $RT(G)$. The output is the new allocation of tasks optimized for lower $DM(G)$ as well as the new $RT(G)$ and $DM(G)$. ILS first applies the heuristic mechanism on the initial solution and then iterates $iter1$ times. In each iteration, it creates a perturbation of the current solution, applies the heuristic mechanism to it and checks if the $DM(G)$ is optimized while $RT(G)$ is under the threshold. The rationale behind using hill climbing is not to move very far away from a neighborhood that is considered to be a good starting point, while we perturb the intermediate solutions to adequately cover the search space.

The perturbation is an algorithm that given the initial allocation and a parameter d, chooses d random jobs and some random DCs and rearranges their task placement fractions by β regardless of the resulting $RT(G)$ and $DM(G)$ values. The Stochastic Hill Climbing heuristic is presented in Algorithm 3. The input of the algorithm is an initial allocation of tasks on the

Algorithm 3 The SHC algorithm

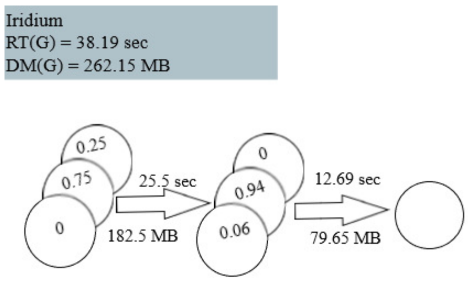
Require: $allocations, RTthreshold, RT(G), DM(G)$

```
for i ← 1 to iter2 do
    Pick a random job
    for eachDC do
        With probability 0.5, reallocate tasks regarding the random job
        through reducing DC's fraction by β
    end for
    tempAllocations ← apply changes to G
    Calculate RT(G)' using tempAllocations
    Calculate DM(G)' using tempAllocations
    if RT(G)' ≤ RTthreshold then
        benefit ← DM(G) − DM(G)'
        if benefit > 0 then
            allocations ← tempAllocations
            DM(G) ← DM(G)'
        end if
    end if
end for
return allocations
```

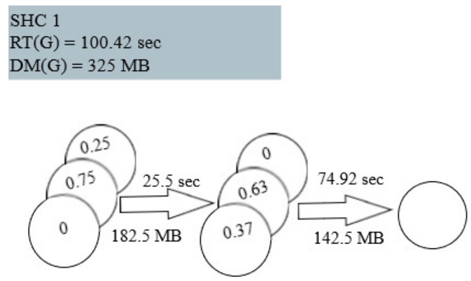
DCs, the threshold of the $RT(G)$, the running time of the initial allocation $RT(G)$ and the initial allocation's $DM(G)$ It consists of a loop that iterates $iter2$ times. In each iteration, it picks a random job and some random DCs. For each DC, it removes a β fraction of its tasks, while dividing this workload to all the other active DCs. If this move is beneficial and the $RT(G)$ remains under the threshold then the initial allocation array is replaced with the current allocation.

4.3.1. Example

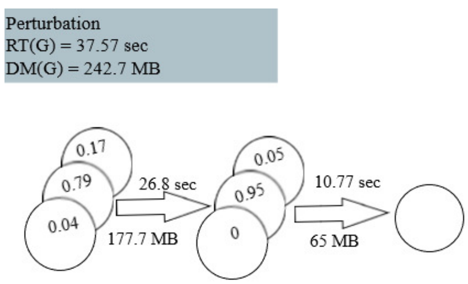
Continuing on our previous example of Greedy, we now show a possible way in which ILS can behave (Figure 4a). We assume $iter1 = iter2 = d = 1$. At first ILS applies SHC on the initial solution. SHC chooses randomly a job, for example the second and then chooses a random DC, for example the second one. The new fractions become r_i^2=(0.37, 0.63, 0). The new metrics are $RT(G)$=100.42 sec and $DM(G)$=325 (Figure 4b). The running time is not under the threshold so the new allocation is rejected. Then ILS perturbates the initial solution. It chooses a random job, for example the first and its third DC. The new results can be seen in Figure 4c. The next



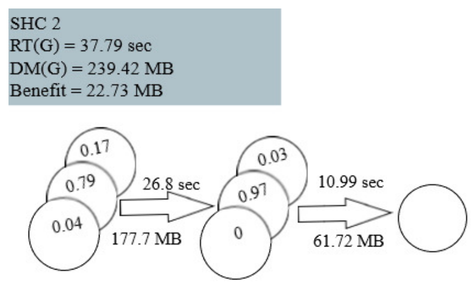
(a) Iridium's task allocation



(b) 1st SHC run



(c) Perturbation



(d) 2nd SHC run

Figure 4: Example using ILS algorithm

move is to call SHC on that new allocation. SHC randomly chooses the second job and the third DC that is r_i^2=(0, 0.97, 0.03). The new $RT(G)$ decreases to 37.79 sec and the $DM(G)$ to 239.42 MB (Figure 4d). Thus, the solution is accepted. In this example, we can see that ILS can also decrease both $DM(G)$ and $RT(G)$.

5. Evaluation

The purpose of the experiments is threefold: firstly to evaluate the relative efficiency of ILS and Greedy in a wide range of scenarios, secondly, to provide concrete insights into the benefits expected, and finally, to compare our solutions against simple techniques, according to which we gather all data on a single DC. We provide both simulations and real runs on a small cluster. $DM(G)$ values are the same in both settings. Using simulations, we can cover a broader range of test scenarios, where we can enforce $RT(G)$ to depend on the communication cost only. On the other hand, the real setting shows actual $RT(G)$ values, where CPU processing is lightweight but not

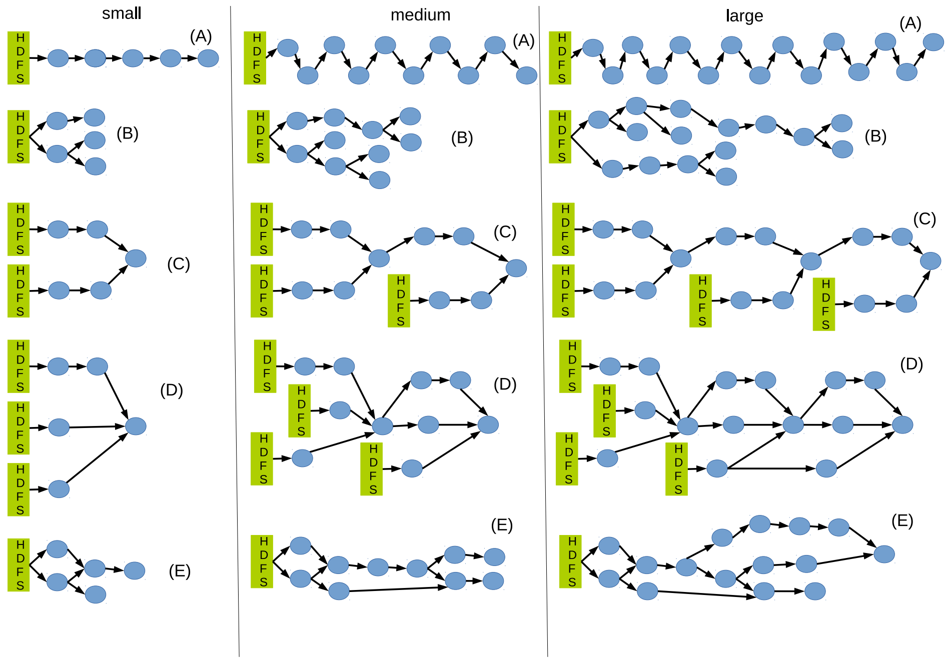


Figure 5: DAGs considered in the experiments (taken from [19])

negligible.

5.1. Experimental Simulation Setting

The experiments we performed were on a range of DAGs as shown in Figure 5 taken from [19]. The DAGs are in three sizes and five types. Type(A) represents a flow in a form of chain and refers to applications that use an initial dataset and produce a final one. In Type(B), more than one final datasets are produced. Type(C) is an extension to Type(A), where some nodes have multiple input datasets. Type(B) and Type(C) are binary tree graphs. Type(D) contains jobs with three input datasets. The graphs have the form of a mesh. Type(E) extends (B) and represents more generic DAGs and not only trees. Each type comes in three sizes, small, medium and large with 5, 10, 15 number of non-source nodes, respectively. Overall there are 15 DAGs. The types are generic enough to capture arbitrary computations, such as those from the TPC-DS and TPC-H benchmarks. For example, as Figure 6 shows, running TPC-H in Spark is equivalent to running multiple Type(B) DAGs.

We also experimented with 3 values of $M = 5, 10, 15$ and 3 values of $\varepsilon = 0.1, 0.2, 0.5$. The experiments were performed for every combination

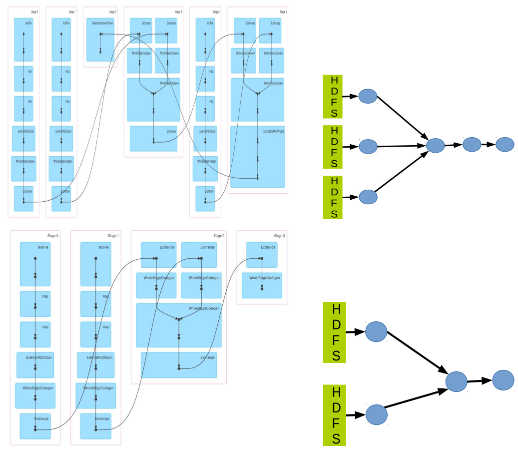


Figure 6: Two representative TPC-H DAGs running in Spark and their corresponding higher-level representation

of DAG, number of DCs and ε value. Unless otherwise stated, $d = 3$, $iter1 = iter2 = 75$ and $\beta = 1/3$. For the remainder of the variables, we resort to a setting similar to the one in [2]. The initial dataset I^j of the source nodes is randomly generated in the range [100MB, 1GB]. The U_i and D_i of each DC fall into the range of [100MB, 2GB]. The selectivities α of the jobs are between 0.01 and 2 with 50% of the job selectivities ranging from 0.01 to 0.5, 25% of them ranging from 0.5 to 1 and the rest 25% ranging from 1 to 2 (similar to the selectivities in Facebook production analytics according to [2]). For each combination of DAG type, M and ε, we created 60 random instances according to the parameters above, and we report the average values.

5.2. Experimental Results and Key Remarks

5.2.1. Main comparison

In the first set of experiments, we compared Greedy to ILS regarding their improvements upon the technique in [2], when we set $\varepsilon = 0.2$. The results are presented in Figure 7 and Figure 8 for $DM(G)$ and $RT(G)$, respectively. As we can observe from Figure 7, ILS is more beneficial than Greedy regarding the total $DM(G)$. Greedy reduces data traffic by a mean of 1.24% while ILS by 44%. More importantly, ILS reduces, most of the time, the $RT(G)$ as well, with a mean reduction of 37% while Greedy increases the $RT(G)$ by 4%. Especially, for the (E) type of DAG, the reductions are over 90%, which corresponds to improvements by an order of magnitude.

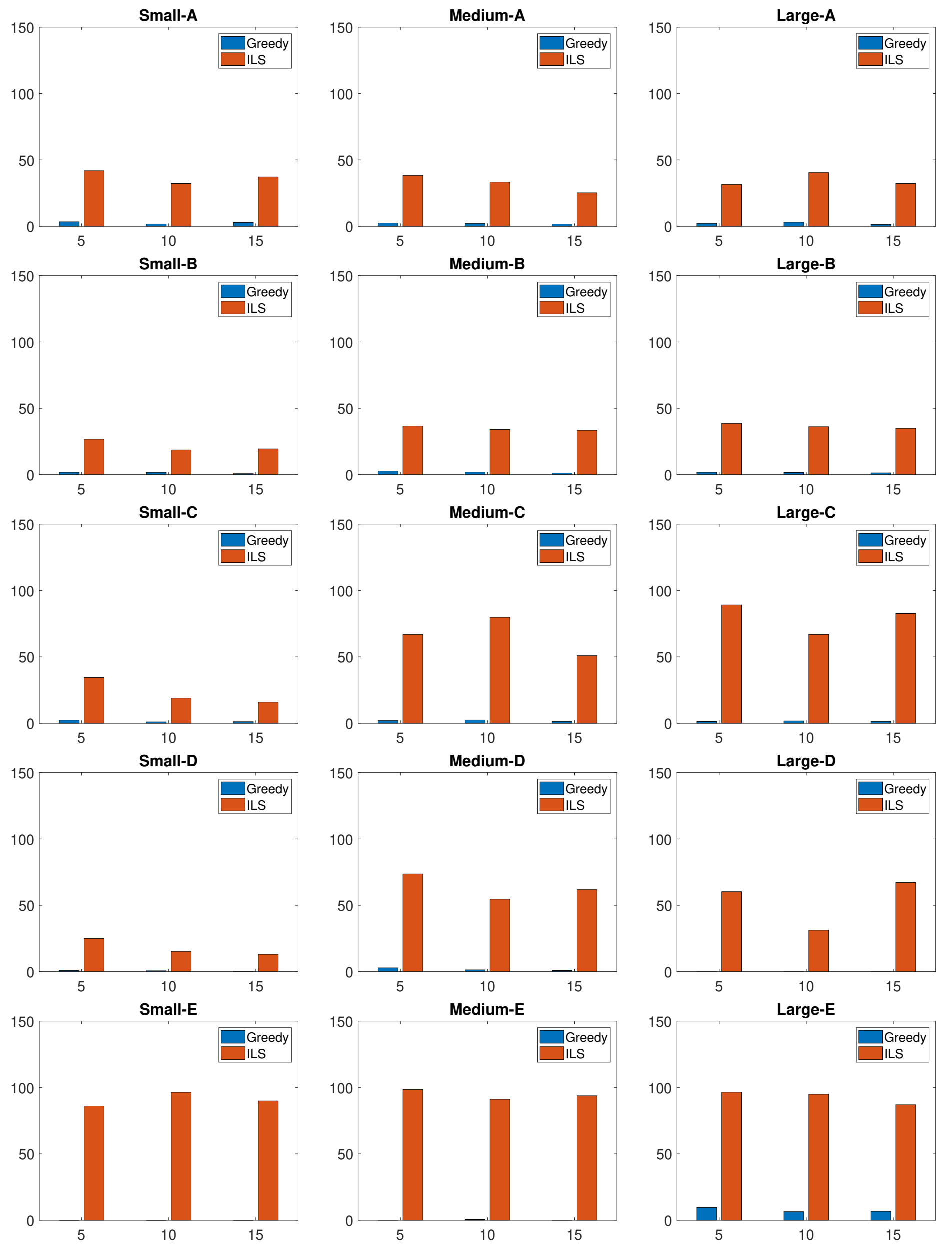


Figure 7: Percentage of $DM(G)$ reduction for M =5, 10 and 15 when ε=20%

In general, the more complex types (C), (D) and (E) benefit more than the simpler ones; this is because, in complex DAGs, there are more inter-

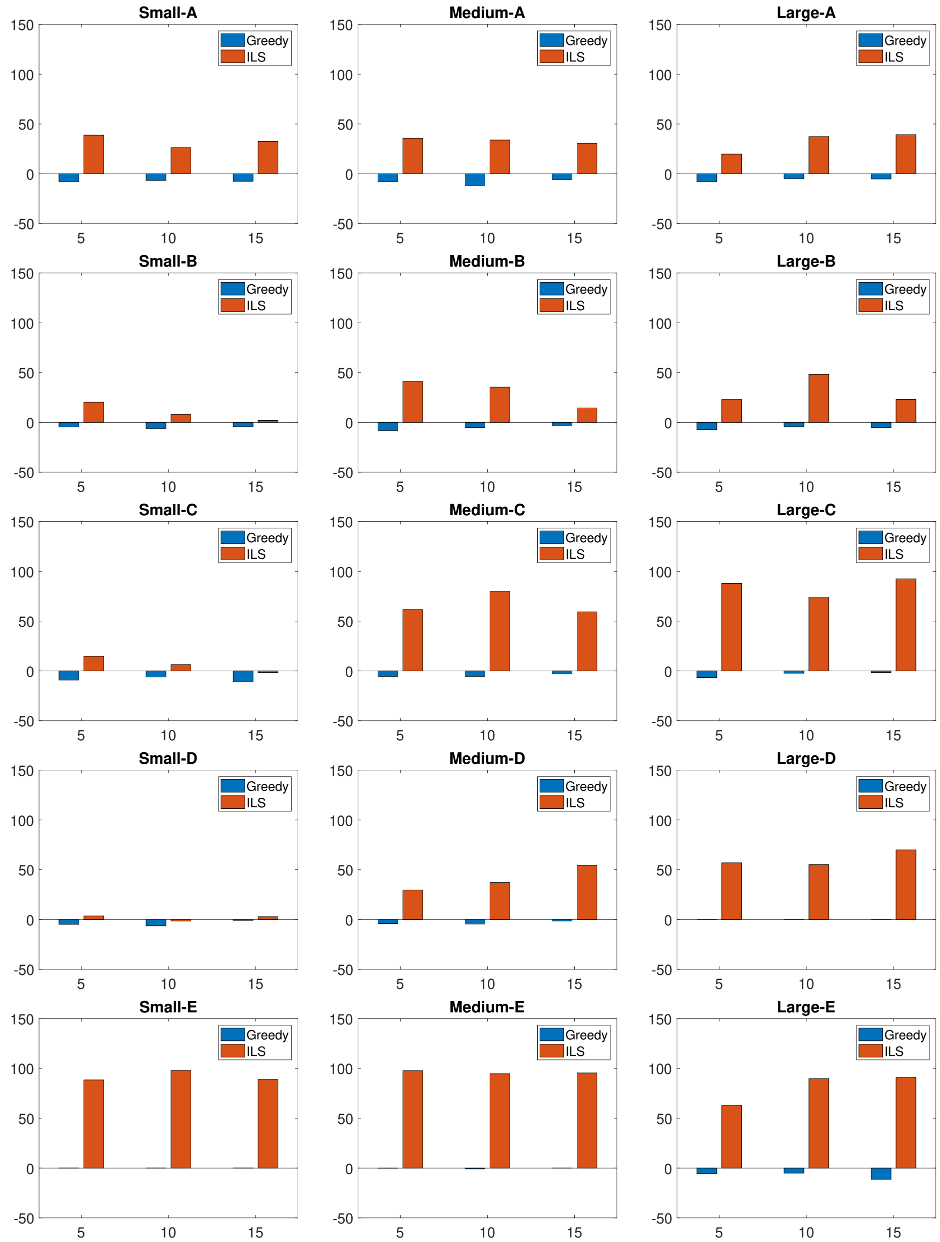


Figure 8: Percentage of $RT(G)$ reduction for M =5, 10 and 15 when ε=20%

dependencies between the DAG nodes that prevent simple greedy heuristics to perform efficiently.

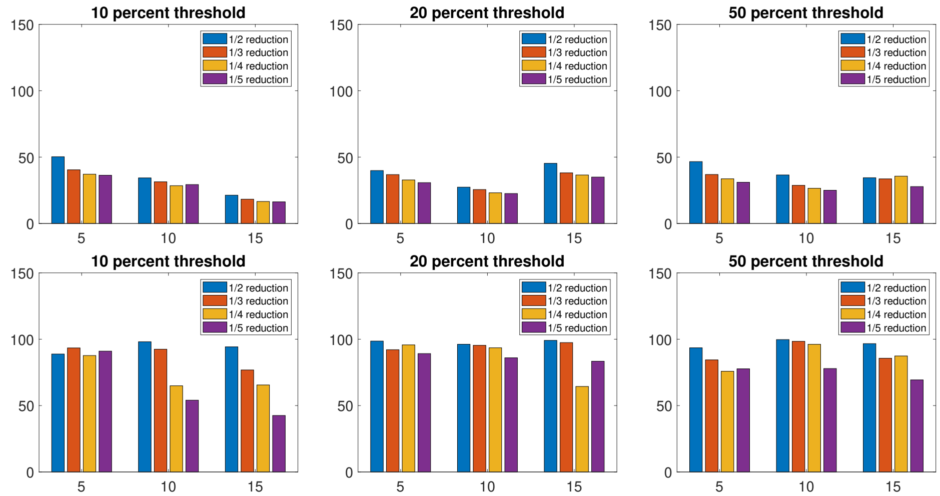


Figure 9: Percentage of $DM(G)$ reduction for the Small-A (top) and Large-E (bottom) DAGs when running ILS for different M (horizontal axis), ε and β values

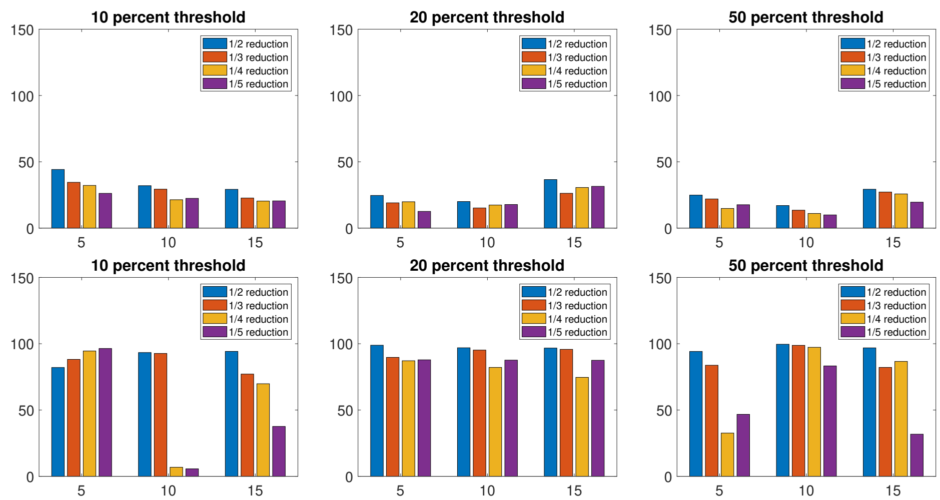


Figure 10: Percentage of $RT(G)$ reduction for the Small-A (top) and Large-E (bottom) DAGs when running ILS for different M (horizontal axis), ε and β values

5.2.2. Sensitivity to the β and d parameters

In the next experiment, we test the impact of β, and more specifically, we experiment with decrease factors of 1/2, 1/3, 1/4 and 1/5. We show results for the Large-E and Small-A DAG types in the Figures 9 and 10. From these figures, we can see that the higher the β the more the reduction of the data

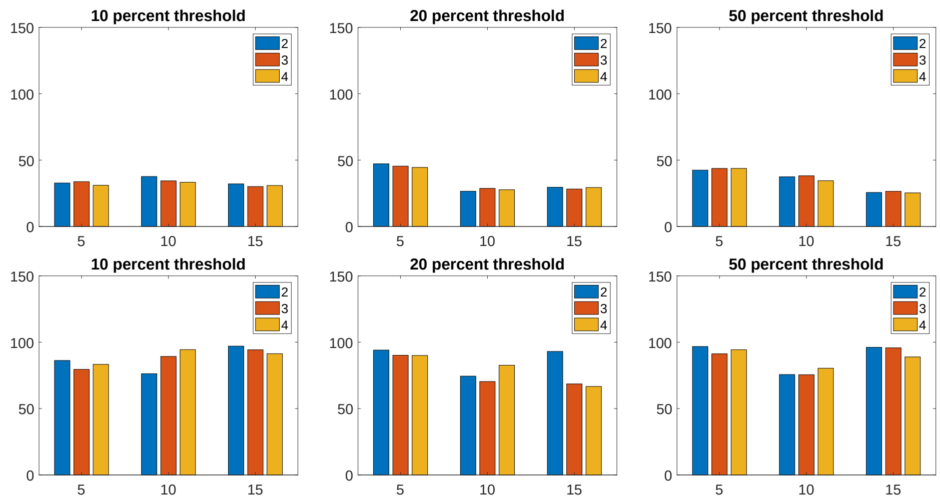


Figure 11: Percentage of $DM(G)$ reduction for the Small-A (top) and Large-E (bottom) DAGs when running ILS for different M (horizontal axis), ε and d values

traffic and running time. More specifically, for Large-E, the average data traffic reductions are 96%, 91%, 81%, 75% and the time reductions 95%, 89%, 70%, 63% when β is 1/2, 1/3, 1/4, 1/5, respectively. Additionally, the reduction is significant higher in Large-E than in Small-A. When running the experiments for Small-A, the data traffic reductions are 37%, 32%, 30%, 28% and the time reductions 29%, 24%, 21%, 18%, for the same β values.

In the next set of experiments, we examine the impact of the variable d on ILS. More specifically, we run ILS for d =2,3,4. The results in Figure 11 indicate that none value of d outperforms the others, i.e., ILS does not rely on the value of d. However, a pattern for the complex DAGs of type (E) is that for limited or many machine choices (i.e., M=5 and 15, respectively), not allowing too much deviation from the initial solutions through a smaller value of d yields better results; for the M values in between the opposite holds.

5.2.3. Impact of initial allocation and convergence

In this experiment, we tested if setting the Greedy algorithm's solution as the initial solution in ILS would have better results than initializing according to the Iridium's solution. We also examined the case in which ILS runs over a random initial allocation. The experiment was performed on the Small-A and Large-E DAGs. The results in Figure 12 show that, in most cases, the random initialization results to a worse outcome than ILS and

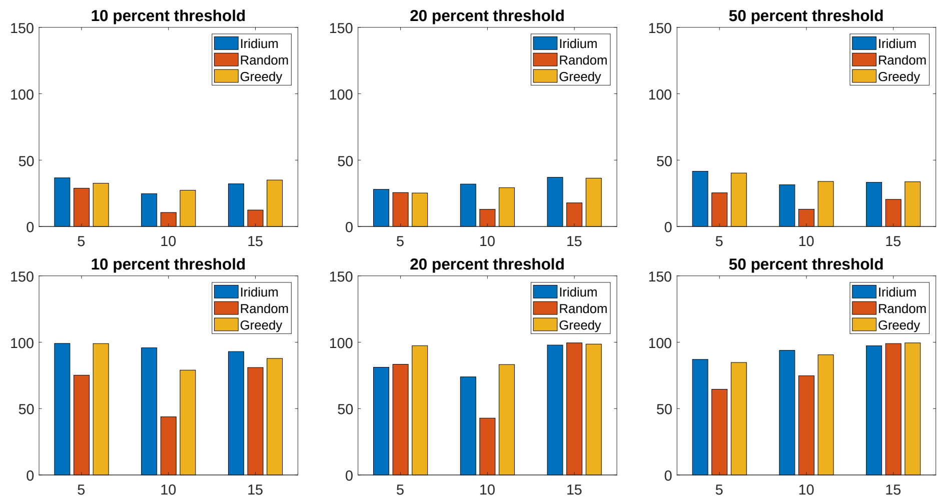


Figure 12: Percentage of $DM(G)$ reduction for the Small-A (top) and Large-E (bottom) DAGs when running ILS on top of Iridium, a random Solution and Greedy for different M (horizontal axis) and ε values

Greedy. Interestingly, ILS and Greedy not only do not dominate each other, but, in certain cases, are superior by a large margin.

Following up on investigating the issue of initial allocation, we examined the convergence rate of ILS with the three different initializations. The results shown in Figure 13 indicate that ILS closely approaches its final output well before the 75th iteration, which is the last one. This allows us to propose a meta-solution, in which ILS runs half of its external iterations starting from the Iridium solution and the other half starting according to the Greedy solution. As shown, in Figure 13, this will not lead to significant performance degradation compared to the results in Figures 7 and 8; and as shown in Figure 12, it may yield further improvements.

5.2.4. On the need for non-naive heuristics

Up to now, we have concentrated on experiments that can provide evidence on the higher efficiency of ILS over Greedy and Iridium; improvements over the latter can be 1-2 orders of magnitude. A question may arise as to whether following a simple approach, such as gathering all data in a single DC, can yield better results. To test this hypothesis, in our last set of our experiments, we tried totally removing a DC rather than decreasing its task allocation proportion while making sure that at least one DC is working, i.e., β was set to 1. We also tested running all the jobs in only one DC. The

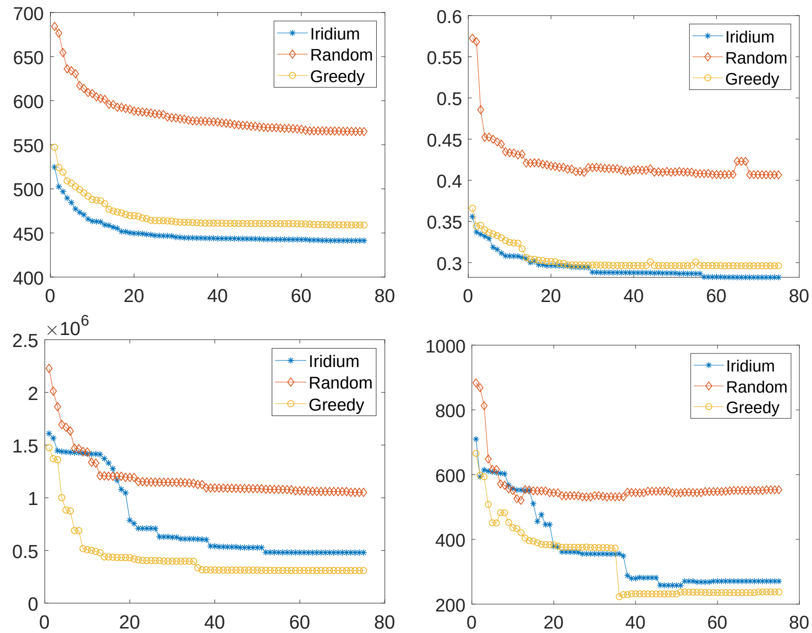


Figure 13: $DM(G)$ (left) and $RT(G)$ (right) convergence rate for the Small-A (top) and Large-E (bottom) DAGs when running ILS on top of Iridium, a random Solution and Greedy for 75 iterations (M=10, ε=10%)

experiments were performed on the Small-A, Large-A and Large-E DAGs. Running all the tasks in one DC means that the data (except for the source jobs) will end up being in that DC. The reduction for Large-E when using ILS or the one DC execution is up to 99% both for traffic and time reduction. For the linear DAGs like Small-A and Large-A the reductions are not that high. As we can see in Figure 14 and Figure 15, the naive solution of running all the jobs in one DC is the most beneficial regarding the traffic reduction by an average of 49% while the running time reduction is only 3.3%. Our technique, which gradually removes DCs using ILS, is more beneficial regarding the running time with 43.5% but less beneficial on data traffic with 32.5% of reduction. However, this means that our technique is applicable when we set $\varepsilon < -0.1$, and in general, yields better improvements on a combined metric that considers $DM(G)$ and $RT(G)$ as of equal importance. More specifically, the ratio of the $RT(G)$ reduction to the $DM(G)$ reduction is 43.5/32.5=1.33 in our case, whereas only 0.067 for the naive solution. This justifies the need to develop more advanced solutions to the problem of bi-objective task allocation in geo-distributed flows.

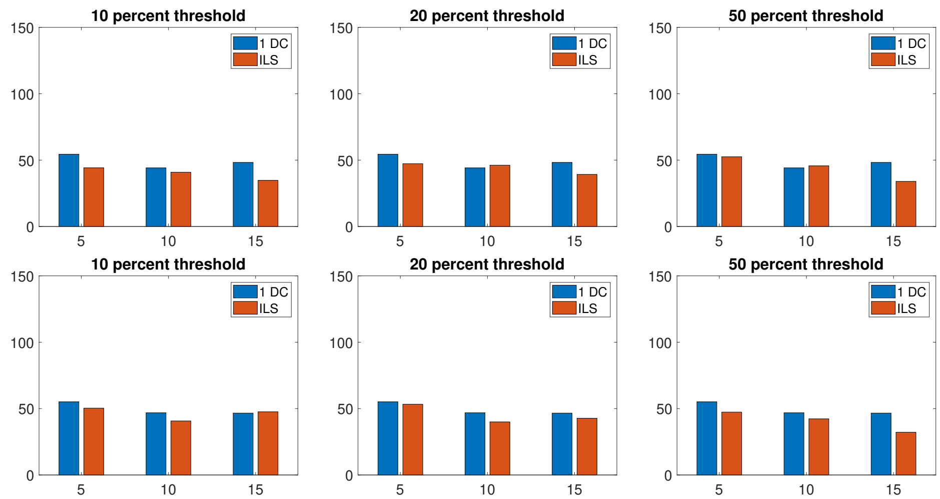


Figure 14: Percentage of $DM(G)$ reduction of ILS and a naive solution that allocates all tasks to a single DC for the Small-A (top) and Large-A (bottom) DAGs for different M (horizontal axis) and ε values

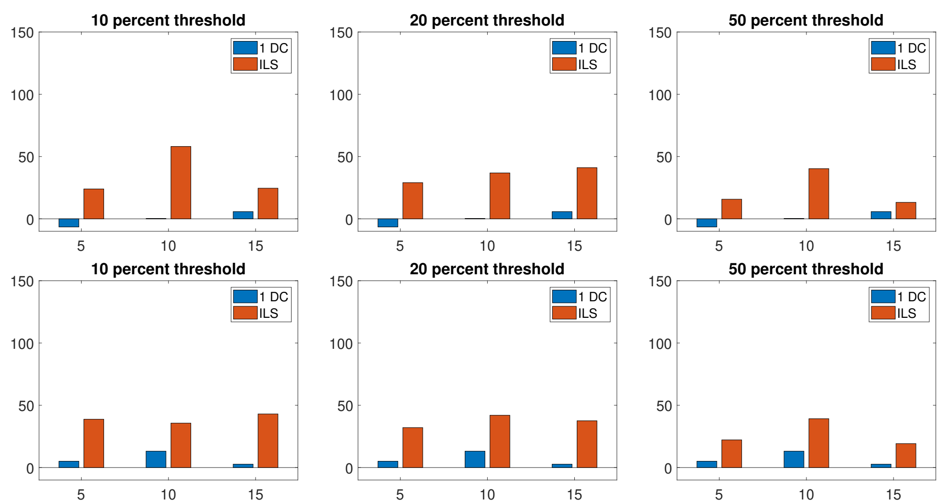


Figure 15: Percentage of $RT(G)$ reduction of ILS and a naive solution that allocates all tasks to a single DC for the Small-A (top) and Large-A (bottom) DAGs for different M (horizontal axis) and ε values

5.2.5. Summary

The main observations are summarized as follows:

1. ILS induces significant larger reduction in $DM(G)$ and $RT(G)$ than

Greedy. Specifically, ILS succeeds an average of 44% reduction in $DM(G)$ while Greedy can reduce it by only 1.24%.

2. ILS achieves most of the time both $RT(G)$ and $DM(G)$ reduction showing that Iridium's task placement is not optimal for complex graphs and that re-arranging the tasks between DCs will likely improve both metrics. In our experiments, ILS decreases time by an average 37%.

3. The β parameter is of high significance. In our experiments, we showed that removing 1/2 of its tasks is more beneficial than removing smaller proportions, i.e., 1/3, 1/4, 1/5. Removing larger proportions has a bigger effect on the total allocation plan thus providing a more beneficial solution. Specifically, $\beta = 1/2$ is on average, 30% more beneficial on data traffic reduction and 55% more beneficial on running time reduction than $\beta = 1/5$.

4. ILS and Greedy can be combined, in the sense that Greedy can serve as the initial allocation further refined by ILS. A complete solution is to run ILS on top of both Greedy and Iridium; running ILS on top of a random initial allocation is always inferior.

5. Running a job in fewer DCs is in general more effective than using all the DCs. In some cases running all the jobs in one DC is even more beneficial. In our experiments we found that running all the jobs in one DC from the start is more beneficial regarding the traffic reduction but gradually removing the DCs using ILS is more beneficial when taking into consideration both time and traffic reduction. Therefore, our solutions are the only option for $\varepsilon < -0.1$.

6. The DAGs which benefit more from our algorithms are the larger and more complex ones like those of the (C), (D) (E) types, with the latter having the highest improvements.

5.3. Spark implementation details and experiments on a real cluster

Our approach has been incorporated into Apache Spark 2.3.2. To enforce our own task placement, we override the `TaskSchedulerImpl` class, where we disable the shuffling of the offers the executors make for a task. We also edited the `TaskSetManager` class to set the task locality to `‘‘Any‘‘` and thus prevent Spark from deciding a placement for the tasks based on the data location.

Table 4: Network capacity of the machines in the cluster (in MB/sec)

	Case A		Case B		Case C		Case D	
Machine	uplink	downlink	uplink	downlink	uplink	downlink	uplink	downlink
1	2	2	5	5	10	10	500	500
2	1	4	2.5	10	5	20	250	1000
3	3	2	7.5	5	15	10	750	500

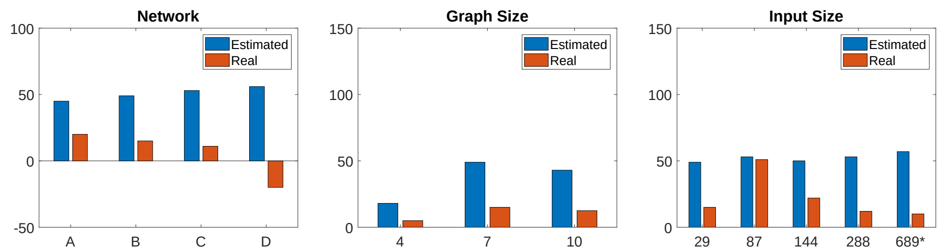


Figure 16: Comparing the percentage of estimated and real $RT(G)$ reductions of ILS for different network configurations (left), different DAG size (middle) and different input size (right, the star means that we switch to Case-C).

The real experiments complement the simulated ones only with regards to the $RT(G)$ value; the $DM(G)$ improvements in a real setting are exactly the same as in the simulations. We use three machines, and we examine chain workflows of Type (A), where, in each stage, data is simply repartitioned with selectivity set to 1. The number of nodes in the DAG is initially set to 7 (repartitioning cannot be enforced to the first one), and the input size is 143.6 MB. Table 4 shows the network capacity in four settings.[4] The data allocations for each Spark stage are estimated offline before the execution begins.

In Figure 16, we compare the estimated RT reduction that ILS achieves over Iridium against the real reduction observed in Spark. On the left handside, we observe that the slowest the network, the highest the actual reduction in $RT(G)$. Then, we keep the Case-B in the table, and we modify the DAG size (see Figure 16(middle)); the actual reductions are more significant for not very small DAGs. Finally, we modify the input size; as shown in Figure 16(right), real executions are sensitive to this metric. Overall, we see that the simulations tend to overestimate the $RT(G)$ reductions mainly due to the non-negligible processing cost in practice, but there are cases, where

[4]In order to set the bandwidth limits of the executors, we used the `Wonder Shaper` script from `https://github.com/magnific0/wondershaper`

Table 5: Real times referring to Figure 16(right)

size (in MBs)	Iridium Time (in mins)	ILS Time (in mins)
29	0.3	0.23
87	1.2	0.58
144	2.6	2.2
288	4.8	4.2
689	7	6.3

Table 6: Running times of the algorithms for different M values (in sec).

	Small A			Medium C			Large E		
Algorithm \M	5	10	15	5	10	15	5	10	15
Iridium	2	2	2	3	3	3	3	3	3
Greedy	1	1	1	3	7	12	4	13	17
ILS (75 iterations)	68	72	78	87	97	107	107	133	167

the estimates are accurate. In the real experiments, we observed $RT(G)$ reductions up to 51%, but if we exclude the minimum and the maximum values, the mean reduction is 13.75%.

Regarding real times, Table 5 shows the actual running times in our 3-machine cluster. In these cases, the overhead to find the optimized allocation for ILS was 35 secs, which means that the overhead is outweighed by the benefits for the input sizes larger than 200MB. In the next section, we provide further overhead information for larger DAGs.

5.4. Optimization Overhead

To assess the runtime overhead of our solutions, we conducted experiments on a machine with i7-4510U CPU at 2.00GHz with 8 GB of RAM. The running time of each algorithm is presented in Table 6, where we can see that ILS is significantly slower than the other techniques, but still, it runs in less than 3 minutes, which renders it a practical option for data-intensive flows, which typically have larger running times (in general, if the running time is lower, then the number of iterations can be decreased).

6. Discussion

This work deals with the problem of minimizing both the total traffic and the communication-bounded flow execution time in multi-stage geo-distributed data flows. We propose a fast greedy solution and a slower, but more efficient stochastic solution. The latter can achieve significant average reductions of 44% and 37% regarding the two metrics, respectively, compared to the Iridium proposal [2]. In specific cases, the improvements can reach 1-2 orders of magnitude. Moreover, we show that a hybrid scheme for the initial allocation refined by the stochastic solution is preferable. Further, we argue that both using all data centers and using only a single one are inferior to our techniques, which follow a middle approach. We believe that this insight is a valuable contribution of our work, since it essentially advocates to take a critical stand on the broadly spread claim that transferring distributed data to a single or fewer places is too costly. Our solution is implemented and tested in Spark as well.

Our work can be extended in several ways. Two promising directions for future work is to extend our solutions for the cases where the processing capacity of a data center is limited and to account also for the processing cost on the data centers, which is now assumed to be negligible compared to the data transmission time. Another line of research could deal with optimizing for multiple queries, which relates also to the issue of revising the initial data allocation. Finally, since our solutions rely on accurate statistics, developing robust techniques that can tolerate inaccuracies in statistical metadata is important.

References

[1] S. Dolev, P. Florissi, E. Gudes, S. Sharma, I. Singer, A survey on geographically distributed big-data processing using mapreduce, IEEE Transactions on Big Data (2017) 1–1.

[2] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, I. Stoica, Low latency geo-distributed data analytics, in: ACM SIGCOMM Computer Communication Review, volume 45, ACM, 2015, pp. 421–434.

[3] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, K. Karanasos, J. Padhye, G. Varghese, Wanalytics: Geo-distributed analytics for a data intensive world, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, 2015, pp. 1087–1092.

[4] K. Kloudas, M. Mamede, N. Preguiça, R. Rodrigues, Pixida: Optimizing data parallel jobs in wide-area data analytics, Proc. VLDB Endow. 9 (2015) 72–83.

[5] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, G. Varghese, Global analytics in the face of bandwidth and regulatory constraints, in: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), USENIX Association, Oakland, CA, 2015, pp. 323–336.

[6] S. Liu, H. Wang, B. Li, Optimizing shuffle in wide-area data analytics, in: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), 2017, pp. 560–571.

[7] P. Eugster, C. Jayalath, K. Kogan, J. Stephen, Big data analytics beyond the single datacenter, Computer 50 (2017) 60–68.

[8] F. N. Afrati, S. Dolev, S. Sharma, J. D. Ullman, Meta-mapreduce: A technique for reducing communication in mapreduce computations, CoRR abs/1508.01171 (2015).

[9] A. Rabkin, M. Arye, S. Sen, V. S. Pai, M. J. Freedman, Aggregation and degradation in jetstream: Streaming analytics in the wide area, in: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), 2014, pp. 275–288.

[10] M. Ryden, K. Oh, A. Chandra, J. Weissman, Nebula: Distributed edge cloud for data-intensive computing, in: 2014 International Conference on Collaboration Technologies and Systems (CTS), 2014, pp. 491–492.

[11] A. C. Zhou, S. Ibrahim, B. He, On achieving efficient data transfer for graph processing in geo-distributed datacenters, in: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), 2017, pp. 1397–1407.

[12] W. Xiao, W. Bao, X. Zhu, L. Liu, Cost-aware big data processing across geo-distributed datacenters, IEEE Trans. Parallel Distrib. Syst. 28 (2017) 3114–3127.

[13] P. A. R. S. Costa, F. M. V. Ramos, M. Correia, On the design of resilient multicloud mapreduce, IEEE Cloud Computing 4 (2017) 74–82.

[14] I. Narayanan, A. Kansal, A. Sivasubramaniam, Right-sizing geo-distributed data centers for availability and latency, in: 37th IEEE International Conference on Distributed Computing Systems,ICDCS, 2017, pp. 230–240.

[15] Z. Liu, P. Balaprakash, R. Kettimuthu, I. T. Foster, Explaining wide area data transfer performance, in: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, HPDC, 2017, pp. 167–178.

[16] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, I. Stoica, Ernest: Efficient performance prediction for large-scale advanced analytics, in: 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI, 2016, pp. 363–378.

[17] Z. Wu, H. V. Madhyastha, Cost-effective geo-distributed storage for low-latency web services, IEEE Data Eng. Bull. 40 (2017) 26–40.

[18] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, S. Sinha, Real-time video analytics: The killer app for edge computing, IEEE Computer 50 (2017) 58–67.

[19] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, J. Torres, Dynamic configuration of partitioning in spark applications, IEEE Trans. Parallel Distrib. Syst. 28 (2017) 1891–1904.