Optimal Reducer Placement to Minimize Data Transfer in MapReduce-Style Processing

Xiao Meng and Lukasz Golab
University of Waterloo, Canada
{*x36meng, lgolab*}*@uwaterloo.ca*

***Abstract*—MapReduce-style processing has become the standard for large-scale distributed platforms, with Hadoop being the most popular implementation. While Hadoop achieves parallelism by scheduling Map and Reduce tasks across the available nodes, it incurs communication overhead in the Shuffle stage which sends intermediate results from mappers to reducers. The problem we solve is as follows: given a collection of mapper outputs (intermediate key-value pairs) and a partitioning of this collection among the reducers, which node should each reducer run on to minimize data transfer? We reduce two natural formulations of this problem to optimization problems for which polynomial solutions exist. We show that our techniques can cut communication costs by 50 percent or more compared to Hadoop's default reducer placement, which leads to lower network utilization and faster MapReduce job runtimes.**

I. Introduction

With the explosion of data volume, velocity and variety, MapReduce has become the standard for big data processing in distributed environments, with Apache Hadoop [1] being the most popular implementation. Hadoop achieves parallelism by scheduling Map tasks (mappers) and Reduce tasks (reducers) over distributed data storage across a cluster of servers.

In MapReduce-style processing, the Map stage locally produces intermediate key-value pairs on each processing node. Next, a *partitioner* module assigns intermediate keys to reducers for processing, a task we call *key partitioning*. Evenly distributing intermediate results, and thus the processing load, among the reducers corresponds to the NP-hard problem of *Minimum Makespan Scheduling* [9]. Much of the previous work on Hadoop task scheduling focuses on load balancing during the Reduce stage [22], [10], [23], [6], [19], [18].

However, key-value pairs assigned to a particular reducer may be scattered among the processing nodes and need to be transferred to the node on which this reducer will run. This *data shuffling* can lead to network congestion and poor performance; e.g., a week-long trace from Facebook's Hadoop cluster containing 188,000 MapReduce jobs shows that data transfer accounts for one third of the running time on average, and over half the running time for 26% of the jobs [4].

To address this problem, we assume that a key partitioning has been computed, e.g., by the Hadoop Partitioner, and we answer the following question: which node should each reducer run on to minimize data transfer? We call this problem *reducer placement*. While key partitioning to balance load is NP-hard, our problem, in which we compute an optimal reducer placement for a given key partitioning, is solvable in polynomial time. We make the following contributions:

1) We introduce the problem of minimizing data transfer in MapReduce-style computation: the *reducer placement* problem. We formulate two versions of this problem: 1) minimizing total data transfer to reduce network congestion and 2) minimizing the maximum data transfer to any one reducer to mitigate shuffle skew.
2) We reduce both versions to our problem to optimization problems with existing polynomial-time solutions. We show that version 1) corresponds to Linear Sum Assignment and version 2) corresponds to Linear Bottleneck Assignment.
3) We empirically validate our approach, showing over 50 percent improvements over Hadoop's default reducer placement [21] in terms of total data transfer (network utilization) and 10-50 percent improvement in maximum data transfer.

The remainder of this paper is organized as follows. In Section II, we review related work. We formulate our reducer placement problems in Section III and solve them in Section IV. We present experimental results in Section V and we conclude in Section VI.

II. Related Work

A MapReduce job is automatically divided into a group of map tasks and a group of reduce tasks. When computation starts, it enters the Map stage. Many map tasks (mappers) run in parallel, each one consuming data residing locally on the node on which it is running. Mappers generate intermediate key-value pairs and write them to local storage. Partitioners then perform key partitioning, i.e., assigning a group of intermediate keys to each reducer for processing. Next, in the Shuffle stage, intermediate results, i.e.,

key-value pairs generated by the mappers, are sent to the corresponding reducers. Finally, reduce tasks run in parallel, applying reducer code to the intermediate results assigned to them to produce the final output.

We now discuss related work on optimizing the MapReduce framework. While there are various possible optimization objectives (e.g., fairness, response time, availability, energy efficiency) [21], we focus on work involving data locality, which directly impacts data transfer.

Once a MapReduce job is submitted, user-level and job-level scheduling algorithms (e.g., Hadoop default, Fair and Capacity schedulers) perform resource allocation and management for this user and job. There are two schedulers that take data locality into account. Delay scheduling delays some users' jobs if the nodes which are currently idle do not have the data required by these jobs [24]. Quincy [15] is another attempt to balance data locality and fairness for concurrent jobs by representing jobs as flow networks and finding the minimum flow cost. In this paper, we consider different objectives and different granularity (reduce tasks of the same job vs. concurrent jobs from different users).

Next, map tasks are generated and scheduled. Similar to Delay scheduling, MatchMaking [13] may delay map tasks until the nodes that have the required data are idle. BAlance-Reduce (BAR) [17] considers data locality while optimizing for completion time. Locality for mapper placement has also been studied in [11] and modelled as a linear assignment problem. We do not address mapper scheduling in this paper.

The next stage concerns reduce task scheduling, which consists of two problems we mentioned earlier: key partitioning to assign groups of intermediate keys to reducers, and reducer placement to decide which reducer will run on which node. The default Hadoop partitioner uses hashing, but users may also write customer partitioner code. Furthermore, there is prior work on approximate algorithms for the NP-hard problem of optimal key partitioning to balance reducer load; see, e.g., [22], [19], [18], [6].

Given a key partitioning, the next step is to place the reducers on the nodes in the cluster, which is the problem we want to solve. In Hadoop, reducers are placed randomly on lightly-utilized nodes without considering data locality. The closest work to ours is the Center-of-gravity (CoGRS) algorithm [12]. When a node becomes free, this algorithm gives preference to reduce tasks for which most of the key-value pairs are already on this node. While this strategy reduces data transfer, we formalize and optimally solve the problem of minimizing data transfer.

III. Problem Definition

Suppose we have a cluster of n servers performing MapReduce-style processing. Suppose the Map stage has terminated and let K_k^j be the number of intermediate key-value pairs for key k generated by the mapper(s) running on server j. Suppose some key partitioning algorithm produced a partition of the intermediate key space consisting of n key groups: $G_1, G_2, \ldots, G_n$. Let G_i be the key group assigned to reducer i.

A single server may run multiple reducers, perhaps as many as the number of cores. In other words, any key group G_i can be assigned to multiple reducers running in parallel on the same server. Since we are interested in minimizing data transfer, all we need to know is which keys will be processed at which server. Thus, in this paper, key groups are assigned to servers and the number of "reducers" implicitly equals n.

Let C be an $n \times n$ data communication *cost matrix*, whose (i,j)th entry, denoted c_{ij}, is the communication cost assuming we place the ith reducer (which is responsible for key group G_i) on the jth server. In the simplest case, we can count the number of key-value pairs required by the ith reducer, namely those corresponding to the keys in G_i, which are not already on the jth server[1]. Formally:

$$c_{ij} = \sum_{k \in G_i, m \neq j} K_k^m$$

In this paper, we want to find optimal reducer placements. We represent a reducer placement using a binary $n \times n$ matrix X, whose (i,j)th entry, denoted x_{ij}, is defined as follows:

$$x_{ij} = \left\{ \begin{array}{ll} 1 & \text{if reducer i is assigned to server j} \\ 0 & \text{otherwise} \end{array} \right\}$$

The first version of the reducer placement problem is to minimize the total data communication cost to reduce network congestion. Formally, we want to compute a reducer placement matrix X to minimize

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

such that each reducer is placed on exactly one server and each server hosts exactly one reducer:

$$\sum_{j=1}^{n} x_{ij} = 1 \ (i = 1, 2, ..., n)$$

$$\sum_{i=1}^{n} x_{ij} = 1 \ (j = 1, 2, ..., n)$$

[1]For datacenters with heterogeneous network architectures, the number of key-value pairs that need to be transferred can be weighted by the distance or link speed between the two machines.

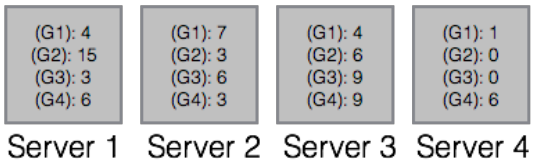


Figure 1: An example of an intermediate key distribution over four servers.

$$x_{ij} \in 0, 1 \ (i, j = 1, 2, ..., n).$$

The second version is to minimize the maximum data communication cost for any one server to mitigate shuffle skew. Formally, we want to compute a reducer placement matrix X to minimize

$$\max_i \sum_{j=1}^{n} c_{ij} x_{ij}$$

subject to the same constraints as above.

We conclude this section with a simple example. Figure 1 shows a distribution of intermediate keys generated by the Map stage on four servers. We use the notation "G1: N" to say that N key-value pairs from key group G1 were generated on a given server. The corresponding cost matrix is shown below.

$$\begin{Bmatrix} 12 & 9 & 12 & 15 \\ 9 & 21 & 18 & 24 \\ 15 & 12 & 9 & 18 \\ 18 & 21 & 15 & 18 \end{Bmatrix}$$

For example, $c_{11} = 12$ is the data communication cost of processing key group G_1 on server 1. Four key-value pairs for G_1 are already on server 1, but the remaining 12 must be transferred to server 1 from the other three servers.

Consider the reducer placement $[1, 3, 4, 2]$, i.e., reducer 1 is placed on server 1, reducer 2 is placed on server 3, reducer 4 is placed on server 3 and reducer 2 is placed on server 4. The total data communication cost is $c_{11} + c_{23} + c_{34} + c_{42} = 12 + 18 + 18 + 21 = 69$. The maximum communication cost per-reducer is 21. On the other hand, the reducer placement $[3, 1, 2, 4]$ has a lower total communication cost of 51 and a maximum communication cost per-reducer of 18.

Note: computing optimal reducer placements is important only if there is skew in the distribution of key groups after the Map tasks are finished, i.e., if the numbers in a particular row of the cost matrix are different. In the above example, if it were true that $c_{11} = c_{12} = c_{13} = c_{14}$, then the communication cost of reducer 1 would be there same regardless of where it was placed.

IV. Our Solution

This section presents our solutions to the two reducer placement problems defined in Section III. We require an $n \times n$ communication cost matrix C as input, meaning that we need to know the intermediate key distribution on each of the n servers. Fortunately, a variety of keyspace summarization techniques for MapReduce-like systems exist, using sampling [19], [20], histograms [10], [16], sketches [22], [23], etc. The output is a reducer placement matrix X.

A. DataSum: Minimizing Total Data Transfer

Our first problem corresponds to Linear Sum Assignment which can be solved optimally in polynomial time using the so-called Hungarian algorithm [2], with a time complexity of $\mathcal{O}(n^3)$. Our solution, named *DataSum*, is shown in Algorithm 1; however, there exist more time-efficient algorithms for the linear sum assignment problem that may be used instead [7], [8].

The Hungarian algorithm exploits the following property: if a number is added to or subtracted from all entries of any one row or column in C, then an optimal solution for the modified matrix is the same as that for the original cost matrix C. To obtain a solution, the algorithm keeps adding or subtracting entries in C until all the zeros can be covered by n straight horizontal or vertical lines. When that happens, an optimal solution consists of n zero-cells in the modified matrix, such that no two zeros lie in the same row or column, and is returned in line 11 of the algorithm.

```
1  Input: an n × n cost matrix C_ij;
2  foreach i in [1,n] (i.e. for each row of C) do
3  |   Subtract the smallest entry in the ith row of C
   |     from all entries in this row;
4  end
5  foreach j in [1,n] (i.e. for each column of C) do
6  |   Subtract the smallest entry in the jth column
   |     of C from all entries in this column;
7  end
8  while true do
9  |   Cover the zero cost entries in C with L straight
   |     (horizontal or vertical) lines;
10 |   if L == n then
11 |   |   return an optimal assignment in C;
12 |   p = smallest c_ij not covered by any line;
13 |   Subtract p from each row that is not yet
   |     covered by any straight line;
14 |   Add p to each column covered by a straight
   |     line;
15 end
```

Algorithm 1: *DataSum.*

We give a worked example of *DataSum* using the following cost matrix:

$$\left\{\begin{array}{cccc} 4 & 3 & 4 & 5 \\ 3 & 7 & 6 & 8 \\ 5 & 4 & 3 & 6 \\ 6 & 7 & 5 & 6 \end{array}\right\}$$

First, *DataSum* subtracts 3 from all entries in row 1, 3 from all entries in row 2, 3 from all entries in row 3, and 5 from all entries in row 4. This gives the following matrix:

$$\left\{\begin{array}{cccc} 1 & 0 & 1 & 2 \\ 0 & 4 & 3 & 5 \\ 2 & 1 & 0 & 3 \\ 1 & 2 & 0 & 1 \end{array}\right\}$$

Next, the algorithm subtracts 0 from all entries in column 1, 2 and 3, and 1 from all entries in column 4, giving the following matrix:

$$\left\{\begin{array}{cccc} 1 & 0 & 1 & 1 \\ 0 & 4 & 3 & 4 \\ 2 & 1 & 0 & 2 \\ 1 & 2 & 0 & 0 \end{array}\right\}$$

We can now cover all zeros in this matrix with four straight lines: one horizontal line through each row, or equivalently, one vertical line through each column. The only possible set of four zero-cells in this matrix, no two of which lie in the same row or column, is: c_{12}, c_{21}, c_{33} and c_{44}. This corresponds to an optimal reducer placement of $[2, 1, 3, 4]$, with a total data transfer of $3 + 3 + 3 + 6 = 15$.

B. DataMax: Minimizing Maximum Data Transfer

Our second problem corresponds to Linear Bottleneck Assignment which can also be solved in polynomial time by a variety of existing algorithms. Our solution, referred to as *DataMax* is shown in Algorithm 2.

We use a simple threshold algorithm [3] which consists of two stages (see [5] for a more efficient augmenting-path algorithm). In the first stage, a threshold cost value c^* is chosen. In the second stage, a threshold cost matrix $\bar{C}[c^*]$ is defined for that threshold based on the cost matrix C provided as input. The (i, j)th entry of $\bar{C}[c^*]$, denoted $\bar{c}_{ij}$, is defined as follows.

$$\bar{c}_{ij} = \left\{\begin{array}{ll} 1, & \text{if } c_{ij} > c^* \\ 0, & \text{otherwise} \end{array}\right\}$$

The algorithm then checks whether a reducer placement with zero total cost exists for $\bar{C}[c^*]$. To do so, the insight is that linear assignment problems are instances of *bipartite graph perfect matching*, where for graph $G(V, E)$, a *perfect matching* $PM \subset E$ exists if every vertex is incident to exactly one edge in PM. Thus, the algorithm defines a bipartite graph $G[c^*](V, E)$ which has an edge $[i, j] \in E$ iff $c_{ij} \leq c^*$, for a given threshold cost value c^*. A perfect matching in this graph implies that a zero-cost reducer placement exists in $\bar{C}[c^*]$, which in turn gives an optimal solution for the original cost matrix C.

```
1  Input: an n × n cost matrix C;
2  c*_0 ← min c_ij, c*_1 ← max c_ij;
3  if c*_0 ≠ c*_1 then
4  |  while C* = {c_ij | c*_0 < c_ij < c*_1} ≠ ∅ do
5  |  |  c* ← median of C*;
6  |  |  if a perfect matching exists in G[c*] then
7  |  |  |  c*_1 ← c*
8  |  |  else
9  |  |  |  c*_0 ← c*
10 |  |  end
11 |  end
12 |  if G[c*_0] not checked for perfect matching then
13 |  |  if a perfect matching exists in G[c*_0] then
14 |  |  |  c*_1 ← c*
15 |  return a perfect matching in G[c*_1];
16 else
17 |  Any reducer placement is optimal;
18 end
```

Algorithm 2: *DataMax*.

The time complexity of *DataMax* is $\mathcal{O}(T(n) \log n)$ where $T(n)$ is the time complexity of checking for a perfect matching. In our implementation, we use the Hopcroft-Karp algorithm [14] with time complexity of $\mathcal{O}(n^{2.5})$ for dense graphs (common in MapReduce jobs whose intermediate results are scattered across the cluster, leaving the cost matrix with few zeros).

We now give a worked example of *DataMax* on the same cost matrix as the one used in the worked example of *DataSum*. We show the matrix again below for convenience.

$$\left\{\begin{array}{cccc} 4 & 3 & 4 & 5 \\ 3 & 7 & 6 & 8 \\ 5 & 4 & 3 & 6 \\ 6 & 7 & 5 & 6 \end{array}\right\}$$

At the beginning, we have $c^*_0 = 3$ and $c^*_1 = 8$. The median of all c_{ij} within the range $[c^*_0, c^*_1]$ is $c^* = 6$. This gives the following threshold matrix, $\bar{C}[6]$, which translates to the bipartite graph in Figure 2.

$$\left\{\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}\right\}$$

There is a perfect matching in this graph (highlighted in blue) so we update $c^*_1 \leftarrow c^* = 6$. The new c^* is 5, from which we generate a new threshold matrix,

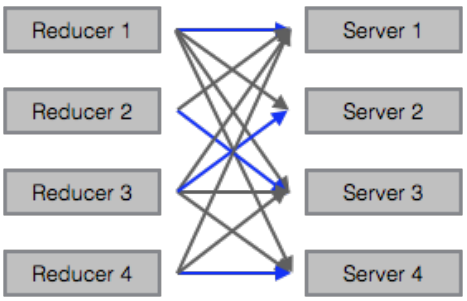


Figure 2: Bipartite graph for $\bar{C}[6]$.

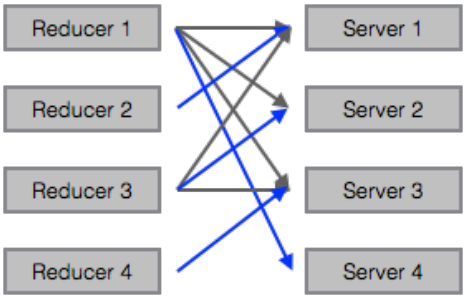


Figure 3: Bipartite graph for $\bar{C}[5]$.

$\bar{C}[5]$, shown below and the corresponding bipartite graph shown in Figure 3.

$$\begin{Bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{Bmatrix}$$

Again, there exists a perfect matching (highlighted in blue), and we update $c_1^* \leftarrow c^* = 5$. The new c^* is 4 which leads to $\bar{C}[4]$ and the corresponding bipartite graph in Figure 4.

$$\begin{Bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{Bmatrix}$$

There is no perfect matching in Figure 4, so we update $c_0^* \leftarrow c^* = 4$. For $c_0^* = 4$ and $c_1^* = 5$, there is no value that falls in this range (that is, $C^* = \{c_{ij} | c_0^* < c_{ij} < c_1^*\} = \varnothing$), so the while-loop terminates. The final assignment uses a threshold of 5, corresponding to the perfect matching illustrated in Figure 3. Thus, we get the following reducer placement: [4,1,2,3]. The maximum data transfer per reducer is 5.

V. Experiments

This section presents our experimental results using a private cluster of 16 nodes (as well as a subset of 8 nodes from this cluster) running CentOS 6.4. Each node is equipped with 4 Intel Xeon E5-2620 2.10 GHz

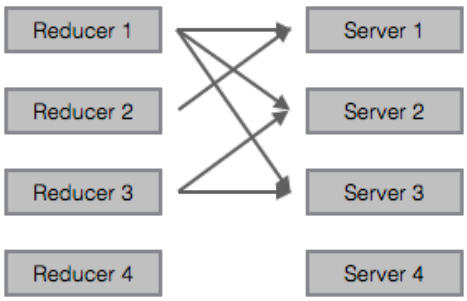


Figure 4: Bipartite graph for $\bar{C}[4]$.

Table I: Dataset details.

Dataset	Source	Size	Key
New York Times	UCI ML Repo.	958MB	docID
PubMed	UCI ML Repo.	7.3GB	docID

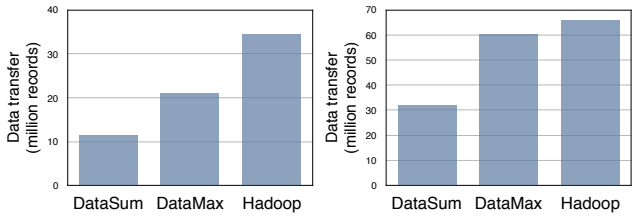
6-core CPUs, 64 GB of DDR3 RAM and 2.7 TB of local storage. The cluster runs Apache Hadoop 1.2.1.

To test *DataSum* and *DataMax* against native Hadoop strategies, we use the TeraSort benchmark from the native Hadoop distribution, available at *sortbenchmark.org*. TeraSort is an external sort and is known to be shuffle-intensive. We use two real datasets described in Table I, New York Times (NYT) and PubMed, and available at *archive.ics.uci.edu/ml/datasets/Bag+of+Words*. Both of these datasets consist of documents, and, for sorting, we use document IDs (docIDs) as intermiediate keys. Document IDs are not primary keys: there may be many records with the same docID. Both datasets have skewed distribution of the docIDs; i.e., some docIDs are much more frequent than others.

Since our cluster resides on a single rack and all the network links have the same speed, we calculate cost matrices using the simple method mentioned in Section III, i.e., by counting the number of key-value pairs that need to be transferred.

A. Experiment I: Minimizing total data transfer

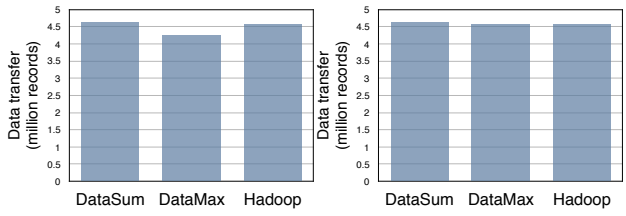
For this set of experiments, our methodology is as follows. First, we load the datasets into the Hadoop Distributed File System (HDFS) and compute the distribution of keys locally stored on each server. We then run the Map stage of the TeraSort benchmark followed by key partitioning, which is done by the custom sampling-based tree partitioner included in TeraSort. At this point, we have the key distribution for each server and the key groups, which allows us to compute the cost matrix. Next, we proceed to the Reduce stage and extract the reducer placement used by standard Hadoop from the Hadoop log file. Finally, we compute our reducer placements using



(a) NYT, 8 nodes (b) NYT, 16 nodes

(c) PubMed, 8 nodes (d) PubMed, 16 nodes

Figure 5: Total data transfer for *DataSum*, *DataMax* and Native Hadoop.



(a) NYT, 8 nodes (b) NYT, 16 nodes

(c) PubMed, 8 nodes (d) PubMed, 16 nodes

Figure 6: Maximum data transfer per server for *DataSum*, *DataMax* and Native Hadoop.

DataSum and *DataMax*, and we calculate data transfer incurred by each strategy from the cost matrix. Thus, we are not calculating the actual data transfer that took place during the MapReduce job, but rather we are estimating it based on the actual key distribution and the actual key partitioning.

We use the New York Times and PubMed datasets, and we use the full 16-node cluster and an 8-node subset. Figures 5a and 5b show the results for New York Times using 8 and 16 servers, respectively. Figures 5c and 5d show the results for PubMed. The Y axis shows the total data transfer across all the reducers. We include all three tested algorithms, keeping in mind that only *DataSum* optimizes for total data transfer.

We conclude that:

- *DataSum reduces the total data transfer by at least 50 percent in all tested scenarios compared to Hadoop. Notably, proper key partitioning alone (performed by the custom TeraSort partitioner) can still lead to high data transfer costs, which can be minimized by applying our reducer placement techniques.*
- *Not surprisingly, DataMax gives reducer placements that have higher total data transfer than those of DataSum, but is still better than Hadoop.*

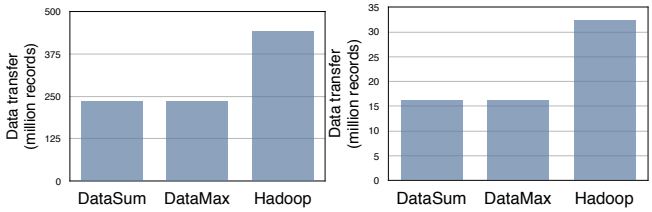
B. Experiment II: Minimizing max. data transfer per-server

Here, we again use the New York Times and PubMed datasets and the same methodology as in the previous experiment. Figures 6a and 6b show the results for New York Times, and Figures 6c and 6d show the results for PubMed (8 and 16 servers, respectively). In this experiment, the Y axis shows the maximum data transfer per server.

Interestingly, *DataMax* reduced the maximum data transfer per server compared to Hadoop only by 10-12 percent. Upon further inspection, we found that the combination of key distribution across the servers and key partitioning was to blame. Many key groups created by the TeraSort partitioner contained a small number of frequent keys which were localized to a small number of servers (between one and three). However, three key groups included key-value pairs that were scattered nearly-uniformly across nearly every server. Thus, no matter which nodes were assigned these three key groups for processing, the data transfer costs were approximately equal since there was no single node that locally stored a majority of the required key-value pairs. As a result, one of these three key groups always had nearly the same maximum per-reducer data transfer (within 10 or so percent), regardless of reducer placement. This explains the results in Figures 6a through 6d.

We hypothesized that *DataMax* would perform better than Hadoop on the TeraSort benchmark if the key distribution or the partitioner were different. For example, if each key group had a "preferred" server and no key group had key-value pairs uniformly scattered across all servers, then *DataMax* would be much more likely than standard Hadoop to find those preferred servers. To test this hypothesis, we modified the PubMed dataset by deleting the keys which were previously scattered uniformly across all servers.

Figure 7a shows the total data transfer and Figure 7b shows the maximum per-server data transfer of each of the three tested techniques using the modified datasets. *DataMax* computes similar reducer



(a) Total data transfer (b) Maximum data transfer

Figure 7: Total and maximum data transfer for the modified dataset, 16 nodes.

placements as *DataSum*, and both of our techniques reduce the total and maximum data transfer by about 50 percent compared to Hadoop. This is likely the best-case scenario for our algorithms given the PubMed dataset and the TeraSort benchmark (including the custom TeraSort partitioner).

Based on the results in this section, we conclude that:

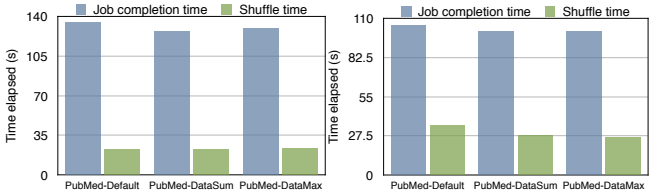
- *The relative improvement of DataMax versus Hadoop in terms of maximum data transfer per-server is sensitive to the distribution of intermediate keys and the key partitioning. With the default data placement on HDFS and the TeraSort partitioner, the improvement was only 10 percent. With a modified PubMed dataset, the best-case improvement was about 50 percent.*

C. Experiment III: Job runtimes in a Hadoop cluster

In this experiment, we implement our schedulers in Hadoop 1.2.1, and study how they impact MapReduce runtimes. We use the default TeraSort benchmark from Hadoop distribution, over both the original and a modified PubMed, all tested on 8 nodes of the cluster. We define *job completion time* as the total runtime and *shuffle time* as the time between the finish time of the last shuffle and the beginning of the first reduce task. We compare *DataSum* and *DataMax* with the default Hadoop scheduler (JobQueueTaskScheduler, essentially a FIFO random strategy). Cost matrices are computed offline in this experiment.

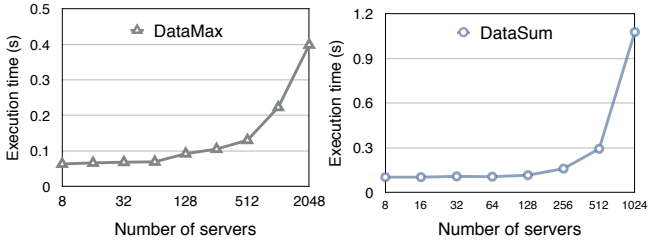
Figure 8a and Figure 8b plot the TeraSort completion times (blue bars) and shuffle times (green bars) on original PubMed and modified PubMed, respectively. We observe that:

- *For the original PubMed, our techniques improve shuffle runtimes by around 10 percent and job completion times by about 6 percent over standard Hadoop. This is consistent with the 10 percent improvement in maximum data transfer per server from Section V-B.*
- *For the modified more skewed PubMed from Section V-C, our techniques can further improve the shuffle time by around 24 percent. Thus, DataSum and DataMax can substantially reduce network utilization and also reduce job runtimes (depending on the skew in the key distribution).*



(a) original PubMed (b) skewed PubMed

Figure 8: Hadoop Job completion and shuffle times, 8 nodes.



(a) *DataMax* (b) *DataSum*

Figure 9: *DataMax* and *DataSum* execution times for different numbers of servers.

D. Experiment IV: Efficiency and scalability of computing optimal reducer placements

The cost of the improvements in network utilization and job runtimes, of course, is that we need to keep track of the key distribution and run our algorithms. In the final experiment, we compute the running time of *DataSum* and *DataMax* on random cost matrices of various sizes. We implemented stand-alone versions of these algorithms in Java, and ran them on the login node of the cluster. Figure 9a and Figure 9b show the average running time (over three runs) as a function of n, the number of servers and therefore also the number rows and columns in the cost matrices. *DataMax* is simpler and therefore faster than *DataSum*, but both can compute optimal reducer placements for thousands of servers within a second.

We reiterate that there are several ways to improve performance which we will investigate in future work: by optimizing our implementations, or by using more efficient and/or distributed algorithms for the corresponding linear assignment problems. Additionally, we can estimate cost matrices early, even before the Map stage terminates, to further reduce the impact of computing an optimal reducer placement on job completion time.

VI. Conclusions

In this paper, we solved a scheduling problem in the context of MapReduce-style processing. We showed how to assign reducers to processing servers in a way that 1) minimizes the total data transfer or 2) minimizes the maximum data transfer per-reducer. We provided optimal polynomial-time solutions for these problems by reducing them to existing optimization problems: Linear Sum Assignment and Linear Bottleneck Assignment, respectively. Our experimental results showed over 50-percent improvements in network utilization (total data transfer) over standard Hadoop. The improvement in maximum data transfer per-server was less pronounced and more sensitive to the key distribution and key partitioning, and ranged from 10 percent to a best-case of 50 percent; this led to a similar improvement in shuffle runtimes.

We suggest two directions for future work. First, we can combine key partitioning and reducer placement and investigate pareto-optimal solutions with respect to their load balancing and communication cost. Second, we plan to study reducer placement for workflows of multiple MapReduce jobs, where the key-value pairs required by a reducer of some job may have already been computed on some node during a previous job.

References

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] Burkard, R., Cela, E. Linear assignment problems and extensions. *Handbook of combinatorial optimization*, 75-149, Springer.

[3] Burkard, R., Dell'Amico, M., Martello, S. Assignment problems: revised reprint, 2012.

[4] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M.I., Stoica, I. Managing data transfers in computer clusters with orchestra. *SIGCOMM 2011*, 41(4), 98-109.

[5] Derigs, U., Zimmermann, U. An augmenting path method for solving linear bottleneck assignment problems. *Computing 1978*, 19(4), 285-295.

[6] Dhawalia, P., Kailasam, S., Janakiram, D. Chisel++: handling partitioning skew in MapReduce framework using efficient range partitioning technique. *International Workshop on Data Intensive Distributed Computing 2014*, 21-28.

[7] Gabow, H., Tarjan, R. Almost-optimum speed-ups of algorithms for bipartite matching and related problems. *STOC 1988*, 514-527.

[8] Gabow, H., Tarjan, R. Faster scaling algorithms for network problems. *SIAM Journal on Computing 1989*, 18(5), 1013-1036.

[9] Graham, R.L., Lawler, E.L., Lenstra, J.K., Kan, A.R. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics 1979*, 5, 287-326.

[10] Gufler, B., Augsten, N., Reiser, A., Kemper, A. Load balancing in MapReduce based on scalable cardinality estimates. *ICDE 2012*, 522-533.

[11] Guo, Z., Fox, G., Zhou, M. Investigation of data locality in mapreduce. *CCGrid 2012*, 419-426.

[12] Hammoud, M., Rehman, M. S., Sakr, M. F. Center-of-gravity reduce task scheduling to lower mapreduce network traffic. *CLOUD 2012*, 49-58.

[13] He, C., Lu, Y., Swanson, D. Matchmaking: A new mapreduce scheduling technique. *CloudCom 2011*, 40-47.

[14] Hopcroft, J.E., Karp, R.M. A $n^{5/2}$ algorithm for maximum matchings in bipartite. *Annual Symposium on Switching and Automata Theory 1971*, 122-125.

[15] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., Goldberg, A. Quincy: fair scheduling for distributed computing clusters. *SOSP 2009*, 261-276.

[16] Jestes, J., Yi, K., Li, F. Building wavelet histograms on large data in MapReduce. *VLDB 2011*, 5(2), 109-120.

[17] Jin, J., Luo, J., Song, A., Dong, F., Xiong, R. BAR: an efficient data locality driven task scheduling algorithm for cloud computing. *CCGrid 2011*, 295-304.

[18] Le, Y., Liu, J., Ergun, F., Wang, D. Online load balancing for mapreduce with skewed data input. *INFOCOM 2014*, 2004-2012.

[19] Ramakrishnan, S.R., Swart, G., Urmanov, A. Balancing reducer skew in MapReduce workloads using progressive sampling. *SoCC 2012*, 16.

[20] Tang, Z., Ma, W., Li, K., Li, K. A Data Skew Oriented Reduce Placement Algorithm Based on Sampling. *IEEE Transactions on Cloud Computing 2016*.

[21] Tiwari, N., Sarkar, S., Bellur, U., Indrawan, M. Classification framework of MapReduce scheduling algorithms. *ACM Computing Surveys (CSUR) 2015*, 47(3), 49.

[22] Yan, W., Xue, Y., Malin, B. Scalable and robust key group size estimation for reducer load balancing in MapReduce. *IEEE International Conference on Big Data 2013*, 156-162.

[23] Yan, W., Xue, Y., Malin, B. Scalable load balancing for mapreduce-based record linkage. *IPCCC 2013*, 1-10.

[24] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. *EuroSys 2010*, 265-278.