On Datacenter-Network-Aware Load Balancing in MapReduce

Yanfang Le[†], Feng Wang[‡], Jiangchuan Liu[†], Funda Ergün[*]
[†]*Simon Fraser University, BC, Canada, Email: {yanfangl, jcliu}@sfu.ca*
[‡]*The University of Mississippi, Mississippi, USA, Email: fwang@cs.olemiss.edu*
[*]*Indiana University Bloomington, Indiana, USA, Email: fergun@indiana.edu*

***Abstract*—MapReduce has emerged as a powerful tool for distributed and scalable processing of voluminous data. For skewed data input, load balancing is necessary among the MapReduce worker nodes to minimize the overall finishing time, which however can incur massive data movement in a datacenter network. In this paper, we for the first time examine this problem of datacenter-network-aware load balancing in the shuffle subphase in MapReduce. Different from earlier studies that generally assume the network inside a datacenter has negligible delay and infinite capacity, we consider the traffic and bottlenecks in real datacenter networks by introducing the constraints on available network bandwidth, and demonstrate that the corresponding problem can be decomposed into two subproblems for network flow and load balancing, respectively. We show effective solutions to both of them, which together yield a complete solution towards near optimal datacenter-network-aware load balancing. A much simpler yet performance-wise comparable greedy algorithm is also developed for fast implementation in practice. The effectiveness of our solution has been demonstrated on synthetic and real public datasets.**

I. Introduction

With the recent burst of information from big data applications, an urgent need is posed to efficiently process the sheer amount of data and handle data-intensive tasks. MapReduce, due to its remarkable features in high flexibility, scalability, and fault tolerance, has emerged as the most popular paradigm for large-scale data processing [1]. The standard MapReduce consists of two phases: *Map* and *Reduce*. A centralized master in MapReduce breaks a computation job into smaller tasks that run on multiple workers in parallel. The mapper nodes process smaller tasks and generate intermediate values on the local nodes in the map phase and the reducer nodes aggregate the intermediate values to form the final output in the reduce phase. Many realworld tasks involving sheer amount of data items are expressible in this way and, as such, can be efficiently processed in this divide-and-conquer manner [1].

State-of-the-art MapReduce implementations rely on a default hash function to assign the intermediate values generated during the map phase to the reducer nodes in the reduce phase, which works well if the data are uniformly distributed. The real world data, however, are not necessarily uniform and often exhibit remarkable skewness, e.g., in PageRank or Inverted Index [2]. The experiments with the popular WordCount application also show similar phenomena [3]. Given that the overall finishing time is bounded by the slowest task, it can be dramatically prolonged with such skewed data. There have been pioneering works addressing this through balancing the load among different machines [4], [2], [5], [3]. To this end, in the *shuffle* subphase of Reduce, the reducers need to pull the intermediate data, i.e., (key, value) pairs from different map workers through the datacenter network, with the constraint that the tasks of the same key must be allocated to the same machine [3].

Despite the efforts toward minimizing data shuffle [6], [7], massive data movement can occur for load balancing when the application scale is large. Earlier studies for load balancing [2], [5], [4], [3] have generally assumed that the network inside the datacenter has enormous bandwidth and thus incurs negligible delay for such movement. Recent studies however suggest that the network inside the datacenter can be a potential bottleneck, too [8]. It has been shown that, in a real datacenter network, there is a great prevalence of highly utilized links (utility of 70% or higher), which are located throughout the network [9], [10]. As such, the point-to-point movement of intermediate data between the map tasks and reduce tasks in the shuffle subphase affects the overall performance of MapReduce [11]. Our measurement reveals that excessive network traffic for the sheer amount data movement can indeed contradict the benefit of load balancing if the network bottlenecks are not carefully avoided. Given the prevalence and the dynamics of the background traffic, the workloads of the machines, the data moving costs and the bottlenecks within datacenter network must be jointly considered when minimizing the overall finishing time for MapReduce.

In this paper, we for the first time examine this problem of datacenter-network-aware load balancing in the shuffle subphase in MapReduce. Different from earlier studies on load balancing that assume the network inside a datacenter is of negligible delay and infinite bandwidth, we consider the traffic and bottlenecks in real datacenter networks by introducing the constraints on available network bandwidth, and demonstrate that the corresponding problem can be decomposed into two subproblems for network flow and load balancing, respectively. We show effective solutions to both of them, which together yield a complete solution toward near optimal datacenter-network-aware load balancing. A much simpler yet performance-wise comparable greedy algorithm is also developed for fast implementation in practice.

We evaluate our algorithms with both on synthetic and real public datasets by comparing with state-of-the-art load bal-

Funda Ergün[‡] is also with Simon Fraser University, BC, Canada.

2159-6190/15 $31.00 © 2015 IEEE
DOI 10.1109/CLOUD.2015.71

ancing algorithm LPT [4], locality-aware partition algorithm LEEN [6] and the MapReduce default solution. The results show that, in practice, our greedy algorithm can significantly reduce the shuffle finishing time for MapReduce. It also enjoys comparable maximum load as the LPT, which is much smaller than LEEN and the MapReduce default solution.

The rest of the paper is structured as follows. Section II introduces the background and motivation. Section III considers the constraints of available network bandwidth and proposes our solutions. Section IV offers the results of performance evaluation. We discuss further enhancements in Section V. The related work is reviewed in Section VI and finally we conclude the paper in SectionVII.

II. Background and Motivation

In this section, we present an overview of MapReduce and discuss the network and data skewness issues therein that motivate our study.

A. Background of MapReduce and Datacenter Network

MapReduce is a distributed computing tool that employs a master-worker architecture, where a single master node manages multiple worker nodes. Initially, the master node partitions the input datasets into multiple even-sized smaller chunks and distributes them to the worker nodes. Each chunk of the input is first processed by a *map* task, which will generate an enormous amount of intermediate (key, value) pairs on the local disks and report the keys and their locations to the master node. The master node then partitions the (key, value) pairs into different reducer nodes based on the keys. The *reduce* tasks will be activated to first pull the data from the map workers (referred as the *shuffle* subphase), and then apply a reduce function to the list of (key, value) pairs on each key.

Existing datacenters generally adopt a multi-root tree topology to interconnect server nodes [9], [10]. In Figure 1, we present a generic datacenter network topology. The 3 layers of the datacenter are the *edge* layer, which contains the Top-of-Rack switches that connect the servers to the datacenter's network; the *aggregation* layer, which is used to interconnect the ToR switches in the edge layer; and the *core* layer, which consists of devices that connect the datacenter to the WAN. In general, for a datacenter, only part of its servers will serve as MapReduce workers.

B. Why Datacenter-Network-Aware?

State-of-the-art studies on optimizing the performance of MapReduce have generally assumed that the keys of the intermediate (key, value) pairs are uniformly distributed [11], [7]. As such, hashing has been used to partition the intermediate data into reduce workers, and a naive hash function, i.e., *Hash (HashCode (intermediate key ID) mod numberOfReducer)* , has been widely used in practice, with which the destinations of the data are known before moving.

It is known that, for highly skewed input data, the simple hashing does not work well, and there have been efforts towards load balancing for such input, particularly in the reduce phase [12], [5]. Assuming that the interconnected network for servers within a datacenter is of ultra-high speed, that is, ideally, of infinite bandwidth, these works literally ignore the original location of the (key, value) pairs as well as the cost to move the data to their destinations, simply sort the keys by decreasing order of the frequency of keys, and then assign to the least loaded machine [4]. Consider an example shown in Figure 2, where key z may be assigned to node A because initially the load of all the nodes is 0, thus making $\{z\}, \{w\}, \{x, y\}$ be assigned to node A, B, C with load 8, 7, 4, respectively. The resultant loads are well balanced; yet, almost all the data have incurred movement, causing significant network traffic.

To understand the impact of such traffics in real world, we have run a series of experiments on a cloud testbed that resembles the configuration of Amazon EC2. Figure 3 shows the detailed results of the processing time in the reduce phase for a *K-Medoids* application, where we adopt 3 instances in 3 physical machines to have a clear view on the data movement. To accurately count the times, we have also enabled the *slow start running* mode, which means the reduce phase will start after all the mappers finish [6], [4]. In the figure, the left part of the processing time for each task gives the shuffling time and the processing time afterwards is roughly the computation time of the reduce function [1]. We can see that, even in this simplified setup, for each task, the latency for shuffling is comparable to that of the reduce function, although only the latter is responsible for computation.

There have been efforts toward minimizing the shuffled data for load balancing, so as to reduce the moving cost and consequently the associated latency [6], [7]. The latency however is not only affected by the amount of the data shuffled, but also by the traffic distribution within the datacenter network. It is known that, roughly 20% of the core links in a real datacenter network are 70% or higher utilized at least 50% of the time, and the link utilizations vary significantly over time [10]. For a datacenter network carrying over the traffic of 1500 monitored machines, 86% of the links observe congestion lasting at least 10 seconds, and 15% observe congestion lasting over 100 seconds [9]. As such, without avoiding potential network bottlenecks, simply minimizing the data to be shuffled does not necessarily achieve desirable performance. Consider Figure 2 again. To minimize the data shuffled, keys $\{x\}, \{z\}, \{y, w\}$ may be assigned to nodes A, B, C, respectively, if the strategy is blind to the locations of the network bottlenecks. This unfortunately hits the bottleneck link from switch $S1$ to switch $S2$. Instead, if we are aware of the bottleneck link by assigning key w to machine A, the overall finishing time can be reduced. In summary, the traffic and the bottlenecks of the underlying network play critical roles toward designing a practical load balancing strategy for MapReduce in real world datacenters.

[1]It in fact includes both the sorting time and the computation time of the reduce function, while the former is small enough to be neglected.

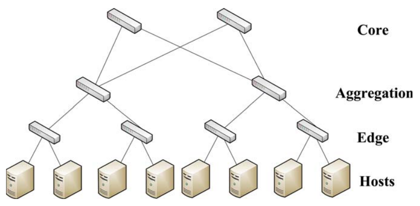


Figure 1. Typical 3-layer datacenter network topology.

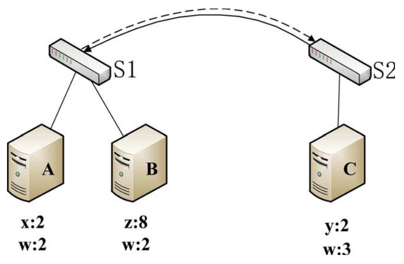


Figure 2. **x:2** under machine A indicates the key x with size 2 in the machine A. Note that the link from switch $S1$ to switch $S2$ is a bottleneck link.

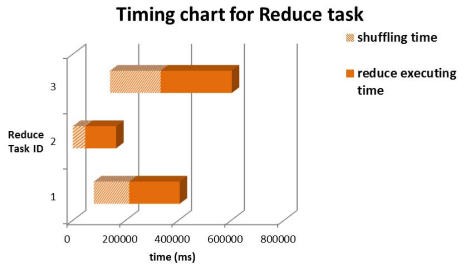


Figure 3. The timing flow of each *Reduce* task.

III. Datacenter-Network-Aware Load Balancing

A. Problem Statement

Define a *cluster* as a set of all the (key, value) pairs that share the same key. The (key, value) pairs of a cluster may be generated by different mappers, and they are often scattered on different nodes. In the *Reduce* phase, each reducer should deal with several tasks, each of which will compute multiple clusters. Before *Reduce* starts, we should first partition the keys into several subsets with the consideration of load balancing among different nodes and move the partitions to the corresponding nodes through the network. Therefore, the overall finishing time of a whole job not only depends on the computation, but also on the network performance for moving partitions to their corresponding reducers.

While sharing certain similarities with the Internet, datacenter networks exhibit many unique characteristics that affect the instantiation of the cost. In particular, a network inside the datacenter network is often of a regular topology and the hop count between a pair of nodes is generally short. The network-related cost therefore mainly comes from the bandwidth constraint of the bottleneck along a path. Although such bottleneck bandwidth can be much higher than that in the wide area Internet, as we have shown earlier, it can create non-negligible delays when moving big data for MapReduce operations [13]. Given there are multiple paths between node pairs, our focus is then to identify the best path for moving data partitions while still keeping the load on each node as balanced as possible.

Here, we assume that at the beginning of the *shuffle* subphase, all the intermediate data have been ready and that the bandwidth from a node to itself is unlimited. We assume that there are totally K different keys (thus K different clusters) and use $C = \{1, 2, \cdots, i, \cdots, K\}$ to denote the set of these keys. The network topology is defined as $G = (V, E)$. For $e \in E$, we have $r_e \in R$ to denote the link rate of edge e, i.e., the available bandwidth. Some nodes in V are the hosts that can run map/reduce tasks, denoted by $M = \{1, 2, \cdots, m\}$, with $m = |M|$ as the number of different machines. We use $X_{k,s}$ to denote the size of cluster k generated by the mapper on node s and $Y_{k,j} = 1$ if cluster k is assigned to node j for $k \in C$, $s, j \in M$.

Suppose that paths $p_1, p_2, \cdots, p_w$ share a link e and have the amount of data $b_1, b_2, \cdots, b_w$ to be transmitted over them, respectively. The time for the data to go through link e is thus $\sum_{i=1}^{w} b_i/r_e$. The total completion time for the *shuffle* subphase is then the maximum value for all link e in the topology. Our goal is to find an assignment to place the clusters over all possible path selections and possible cluster placements so as to minimize the overall finishing time.

Obviously, the size of the data assigned to node j is $\sum_{k=1}^{K} Y_{k,j} \sum_{s=1}^{m} X_{k,s}$. Let $f_{ksj}(u, v)$ indicate the size of cluster k from machine s to machine j that will put on the link (u, v). Thus, the amount of data to go through link (u, v) is

$$D(u, v) = \sum_{s=1}^{m} \sum_{j=1}^{m} \sum_{k=1}^{K} f_{ksj}(u, v). \tag{1}$$

In this context, our objective is find $f_{ksj}(u, v) \in \mathbb{R}$ and $Y_{k,j} \in \{0, 1\}$ so as to minimize

$$\max_{j\in M} \left\{ \max_{(u,v)} \{ \frac{D(u, v)}{r_{(u,v)}} \} + f_c(\sum_{k=1}^{K} Y_{k,j} \sum_{s=1}^{m} X_{k,s}) \right\} \tag{2}$$

where the (u, v) for the inner max should be chosen as

$$\forall (u, v) \in \left\{ (u, v) | f_{ksj}(u, v) \neq 0, s, j \in M, u, v \in V, k \in C \right\},$$

and $r_{(u,v)}$ is the rate, i.e., the available bandwidth, of edge (u, v). $f_c(s)$ denotes the reduce function computation time and takes s (the size of the data to be processed) as the input. We know that the overall reduce finishing time is bounded by the maximum finishing time on each machine, which consists of the shuffle finishing time and the reduce function computation time. For a specific machine j, the shuffle finishing time is determined by the time that the last packet goes through a certain link that is on the paths to the machine j. This link must have the maximum transmission time for the node j, which is the first part of Formula 2. The second part is the computation time of the reduce function for the workload on node j. Since the problem is very complicated, we divide this problem into the network flow part and the load balancing part, and propose our solutions accordingly.

B. Optimizing Network Flow

For the networking part, given the destination of each cluster, our input of the networking part, T, is a list of tuples $T_i = (s_i, b_i, d_i)$, which means that b_i size of data at machine s_i will be delivered to machine d_i. Note that the size of $|T|$ is at most $m^2 - m$, since it is not necessary to transmit data when s_i is equal to d_i.

Let $f_i(u, v)$ denote the size of tuple T_i going through edge (u, v). The finishing time of data transmission is determined by the slowest edge. Thus we want to minimize

$$t = \max_{(u,v)\in E} \sum_{i=1}^{|T|} f_i(u, v)/r_{(u,v)} \ .$$

In other words, if we are given a time t to ship all the data to their destinations, we can reconstruct the graph G with capacity $r_{(u,v)} * t$ for each edge (u, v). We then need try to find a feasible solution, such that all the data can reach their destinations with the following constraints

$$\sum_{i=1}^{|T|} f_i(u, v) \leq r_{(u,v)} * t, \forall (u, v) \in E \ ;$$

$$\sum_{w\in V} f_i(u, w) = 0, \text{when } u \neq s_i, d_i \ ;$$

$$f_i(u, v) = -f_i(v, u), \forall u, v \ ;$$

$$\sum_{u\in V} f_i(s_i, u) = \sum_{u\in V} f_i(u, d_i) = b_i \ , \qquad (3)$$

and make t as small as possible. This problem produces an integer flow to satisfy all the demands. However, even its decision version is a strongly NP-complete problem. Fortunately, we can find the maximum ratio λ such that at least λ ratio of each tuple can be shipped without exceeding the capacity constraints, which is the maximum concurrent flow problem [14]. Here we need to maximize

$$\lambda = \min_{1\leq i\leq |T|} \sum_{w\in V} f_i(s_i, w)/b_i \ ,$$

and accordingly, we need to revise the equation (3) to less than or equal to b_i. This is equivalent to the problem of determining the minimum ratio by which the capacity must be increased to make all the data be shipped to their destinations. One solution to this problem is to use linear programming, which can get the λ for a given time t.

Here, λ is always less than or equal to 1. To achieve the minimum value of t, if $\lambda = 1$, we can decrease the current value of t; otherwise, we can increase the current value of t. Note that the solution is a feasible solution when $\lambda = 1$, although t may be too large. By this means, we can develop an algorithm to optimize network flow as summarized in Algorithm 1.

In the algorithm, given a time t and by reconstructing the graph G with the capacity $r_e * t$ for each edge e, the linear programming is applied to find the maximum minimum ratio λ. A binary search is then applied on the interval $[0, maxT]$

Algorithm 1 Network-flow(T)

```
Input: a list of $T_i = (s_i, b_i, d_i)$ tuples
 1: $maxS = \sum_{i=1}^{|T|} b_i$
 2: $maxT = maxS/min\ r_e$
 3: $previousT = maxT$
 4: $t = maxT/2$
 5: $low = 0$, $high = maxT$
 6: while true do
 7:     reconstruct a graph $G$ with the capacity of each edge $e$
        equal to $r_e * t$
 8:     use linear programming to get the maximum minimum
        ratio $\lambda$, such that each tuple can be shipped without ex-
        ceeding the capacity constraints, and the corresponding
        flow $F$.
 9:     if $\lambda == 1$ then
10:         if $previousT == t$ then
11:             $FinalT = t$
12:             $goodFlow = F$
13:             break
14:         else
15:             high= (low+high)/2
16:         end if
17:     else if $\lambda < 1$ then
18:         low= (low+high)/2+1
19:         if high < low then
20:             high = low
21:         end if
22:     end if
23:     $previousT = t$
24:     t= (low+high)/2
25: end while
Output: $FinalT$, $goodFlow$
```

until the time t is converged to the optimal value with λ still equal to 1. $maxT$ is the upper bound of the maximum transmission time and can be calculated by the size of all the data over the minimum bandwidth link among all the links.

If the $\lambda < 1$, there exists at least one tuple that only part of its data is transmitted. Thus, we search the time on the higher part of the interval; otherwise, we search the time on the lower part. This process will continue until λ is equal to 1 and the time in the previous iteration and the current iteration are the same. Our algorithm also guarantees that when the time converges, all the data can be transmitted over the network. This is because the $maxT$ is an upper bound of the time and the least time t that makes λ equal to 1 is always in the searching space $[0, maxT]$. We thus have the following theorem.

Theorem 1. *Algorithm 1 can achieve optimality for the network flow part with the given destinations for the clusters.*

C. Integrating with Load Balance

Analyzing the structure of our problem can help us to design a good heuristic algorithm based on the theoretical guaranteed classical solution, while considering the interactions of these two parts. For the load balancing part, our problem is similar to the well-known NP-complete problem to schedule jobs on the identical machines [15], where we can deem each cluster as a job, and the processing time of each job is thus the size of each cluster. The state-of-the-art solution for this problem is to sort the clusters in non-increasing order by the size, and put them into the least load machine one by one, which can achieve $4/3-$approximation to the optimal solution [15].

In our solution, we adopt the slow start in MapReduce, which waits until all the mappers finish, i.e., all the intermediate data are generated and ready to be transmitted. Notice that the same key could be on different machines before the placement. We propose a network flow-based algorithm outlined in Algorithm 2. The input of the algorithm is a list of tuples $(i, u, x_{i,u})$, indicating key i of size $x_{i,u}$ at machine u. The algorithm first sorts the key i by non-increasing order of the total size of all the tuples with the same key. For each key j, it forms the input set of $T_i = (m_i, s_i, n_i)$ for Algorithm 1. It then tries all the machines and assigns the key to the machine with minimal cost, which is the sum of the shuffling time and the computation time.

Algorithm 2 Network Flow-based Partition Placement

```
Input: a list of $(i, u, x_{i,u})$ tuples
1:  sort the key $i$ by non-increasing order of size $\sum_{u=1}^{m} x_{i,u}$
2:  for each key $j$ do
3:      $maxT = \sum_{i=1}^{j} x_{i,u}/min\ r_e$
4:      for each machine $n$ with load less than or equal to the average load do
5:          $Y_{j,n} = 1$ and $Y_{j,r} = 0, \forall r \neq n$
6:          Form a set $T = \{(s_i, b_i, d_i) | b_i = \sum_{i=1}^{j} Y_{i,d_i} x_{i,s_i}, s_i, d_i \in M\}$
7:          $t_n$, $F_n$ = Network-flow($T$)
8:      end for
9:      reassign $Y_{j,n} = 1$ and $Y_{j,r} = 0, \forall r \neq n$ based on $t_n$ and $F_n$, such that the objective value is minimized when key $j$ is assigned to machine $n$
10: end for
```

Theorem 2. *The load balancing part of the algorithm 2 can achieve 2-approximation.*

Proof: Since the average load, ave, is less than or equal to the maximum load of the optimal solution, OPT, and the maximum size of any key is also less than or equal to that of OPT. Assume the maximum load is on machine j, and the tuple with size b is the last new one assigned to the machine j with load L at that time. Because the algorithm always chooses the node with load less than or equal to the average, thus $L \leq ave \leq OPT$ and $b \leq OPT$, as a result, the maximum load is $L + b \leq 2 * OPT$. In other words, the maximum load in worst case still can achieve 2-approximation to the optimal solution. ■

Although Algorithm 2 can achieve a well-bounded performance, in practice, we find that linear programming may take relatively longer time with larger number of machines and data size. To this end, we further design a greedy heuristic called *Greedy Network-aware Partition Placement*. This algorithm works by trying to place the data flow on the links with the least finishing time while choosing the destination machine with the load less than or equal to the average. The algorithm first sorts the keys by non-increasing order of the size and assigns each key to the least load partition, similar to the state-of-the-art load balancing algorithm. Then, for each partition, it tries all the machines and will pick a machine, to which the least finishing time can be achieved over all partitions that have been assigned so far. Besides, for each tuple that belongs to a partition, we also choose the path that yields the least shuffle finishing time. This continues until all the partitions are assigned to machines and all the tuples find their best paths to their destinations.

IV. Performance Evaluation

A. Methodology

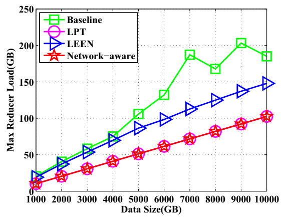
We evaluate our solution by both synthesized data sets and a real data trace. The real trace is adopted from [16], which describes the Wikipedia page-to-page link for each term and can generate $130,160,392$ (key, value) pairs [6]. For the synthesized data sets, we adopt a typical setting as used in [3], [4], which generally follows the Zipf distribution. By default, we set the skewness parameter $s = 0.7$ and the size of a data set is $5,000,000$ (key, value) pairs with each pair of the size $1MB$. We use a typical datacenter network topology, namely, the fat-tree topology [17] [2] and set the pod number to 12, which leads to 432 hosts and 180 switches in total. The machines used for the MapReduce workers are randomly chosen from the 432 hosts and the default number of the reducers is 100. All the links in the network are set to the bandwidth of $1GB/s$. To emulate the real world datacenter network traffic, we also generate the background traffic according to the characteristics of the datacenter network reported in [10].

For comparison, we implement another three algorithms: the Baseline algorithm that uses the default routing algorithm and hash function in MapReduce to determine the cluster placement; a state-of-art load balancing algorithm named LPT [4], which sorts the keys by the decreasing order of their size and each time assigns one key to the machine currently with the least load; and LEEN [6], which is the state-of-art locality- and fairness- aware key partitioning algorithm proposed for MapReduce. Due to the linear programming procedure in the algorithm 2 might take long time to solve, especially when the number of the reducers and the number of the (key, value) pairs grow large, we use our greedy network-aware partition placement algorithm (denoted as Network-aware) for the evaluation, where the observed performance difference when compared to our linear programming approach on small scale

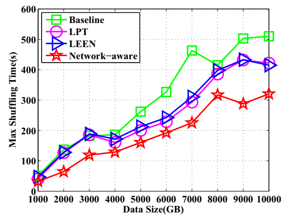
[2]It is worth noting that our algorithms can apply to the general datacenter network topology, because we did not assume any network topology in our algorithm design.

datasets, is generally within 20%. Also, to better understand how our solution can improve the overall performance, instead of using an arbitrary complexity function on the load of each machine, we separate the final results into the maximum load on a reducer (Max Reducer Load) and the maximum shuffling time (Max Shuffling Time) during the shuffle subphase. We then investigate how our solution performs with different data size, reducer number and data skewness, respectively.

B. Results on Synthetic Datasets

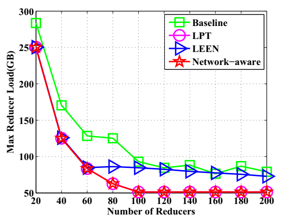


(a) Max reducer load

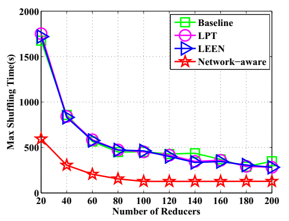


(b) Max shuffling time

Figure 4. Scalability with Different Data Size on Synthetic Datasets.

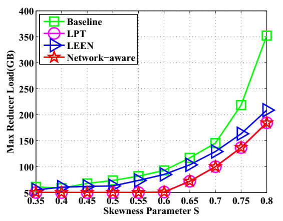


(a) Max reducer load

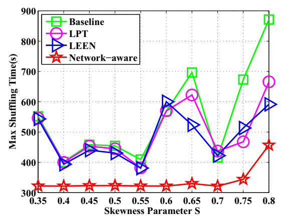


(b) Max shuffling time

Figure 5. Adaptability to Various Reducer Number on Synthetic Datasets.



(a) Max reducer load



(b) Max shuffling time

Figure 6. Resistance to Diverse Data Skewness on Synthetic Datasets.

1) Scale with Different Data Size: Figure 4 shows how the four algorithms perform with different size of the synthesized datasets from $1,000$ GB to $10,000$ GB. It is easy to see that in Figure 4(a), the max reducer loads of both the Baseline and LEEN algorithms become much worse when the size of the datasets grows. Especially when the data size is $10,000$ GB, the performance degradation can be up to 60% as compared to the LPT algorithm, which is a state-of-the-art design dedicated to optimize the load balance. On the other hand, our Network-aware solution performs almost identical to the LPT algorithm, further demonstrating that our solution can effectively balance the load among the reducers.

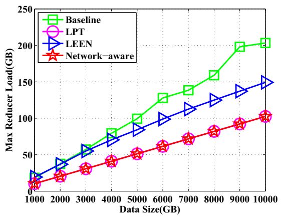
Figure 4(b) shows how the maximum shuffling time of the four algorithms changes with different data size. We can see that our solution always outperforms the other three algorithms with the gain up to 40.69%. The Baseline algorithm generally performs the worst as it uses the default hash function in MapReduce which is oblivious to the costs of moving data across the network. The LEEN algorithm performs slightly better than the LPT algorithm as it considers the data locality and tries to move the minimum amount of data over the network, which can help reduce the max shuffling time if the data happen to go through the routing paths with similar amount of background traffics. However, as neither of the two algorithms is aware of the traffic bottlenecks in the network, their max shuffling time can significantly increase if any link along the routing path is heavily loaded by the background traffic, while our Network-aware solution can successfully avoid such situations and constantly achieve much lower max shuffling time than the other three algorithms, especially when the data size becomes large.

Comparing the results in both Figures 4(a) and 4(b), it is not surprising to see that as the data size grows, both the max reducer load and the max shuffling time will increase. However, our solution increases in a much slower pace than the other three algorithms, indicating a good scalability for big data processing. Moveover, our solution can successfully reduce the max shuffling time while still performing almost the same to the LPT algorithm on optimizing the load balance, which further confirms the superiority of our algorithm design.

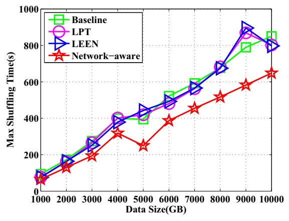
2) Adapt to Various Reducer Number: We next examine how the performance changes for all the four algorithms by varying the number of reducers from 20 to 200. The results are illustrated in Figure 5. In term of the max reducer load as shown in Figure 5(a), our Network-aware solution stays almost the same as the LPT algorithm and always outperforms the LEEN and Baseline algorithms with the reduction up to 44.3% on the max reducer load. One interesting observation is that our solution becomes stable when the number of reducers is greater than 100. A close look at the data distribution reveals that there is one key that is extremely frequent and our solution successfully assigns this key solely on a reducer while still keeping the loads on the other reducers less than this, which further confirms that our solution can reach a near optimal load balance.

For the maximum shuffling time shown in Figure 5(b), our Network-aware algorithm always outperforms the other three algorithms, with the performance gain up to 55.22%. In both Figures 5(a) and 5(b), we can see that our solution can well adapt to the various number of available reducer resources, which can be an important feature in practice, especially given the dynamic resource availability due to the interferences from other applications in the datacenter, this feature can allow the available resources to be fully exploited.

3) Resist to Diverse Data Skewness: Figure 6 shows the results of the max reducer load and max shuffling time as a function of the skewness on different synthesized datasets. We vary the skewness parameter s in the Zipf distribution from 0.35 to 0.8, where the larger s is, the more skew the dataset is. As shown in Figure 6(a), we can see that when the data is less skew (i.e., $s = 0.35$), although the gaps of the max reducer loads among all the algorithms are relatively small, our solution performs almost the same to the LPT algorithm, and always outperforms the LEEN and Baseline algorithms, which becomes more remarkable as the skewness grows large, with a gain as much as 47.73%. The result of the maximum shuffling time shown in Figure 6(b) also shows that our algorithm always outperform the other three algorithms with a reduction up to 49.01%, which further verifies the effectiveness of our algorithm. It is interesting to see that the other three algorithms are experiencing notable fluctuations in Figure 6(b). This is because these three algorithms are blind to the network status when assigning the (key, value) pairs to the machines, which can lead to unexpected heavy traffics on certain links and cause network bottlenecks.

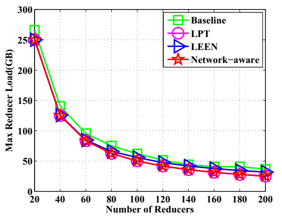


(a) Max reducer load

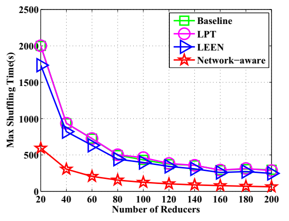


(b) Max shuffling time

Figure 7. Performance as a Function of Data Size on Real Data Trace.



(a) Max reducer load



(b) Max shuffling time

Figure 8. Performance as a Function of Reducer Number on Real Data Trace.

C. Results on Real Data Trace

Besides synthesized datasets, we also conduct extensive evaluations with a real trace from [16], where we emulate the map phase with the original tuples from the trace randomly allocated on different machines. We vary the number of (key, value) pairs from $1,000,000$ to $10,000,000$. Given that the size of one (key, value) pair is 1 MB, the size of the data is thus from $1,000$ GB to $10,000$ GB. Figure 7 shows the results of the max reducer load and the max shuffling time, where the trend matches well to that in Figure 4, further confirming the validity of the evaluation results under the synthesized datasets. We also examine how the four algorithms perform with different reducer number. The results are shown in Figure 8. Again, our solution has the best performance in terms of both max reducer load and max shuffling time. Figure 8(b) shows a similar trend and gain to Figure 5(b), while the gain on the max reducer load in Figure 8(a) is relatively small compared to that shown in Figure 5(a). We examine the key id and distribution in the dataset and find that the key ids happen to be well aligned with the hash function mapping mechanism, which results in a relatively good load balancing even for the Baseline algorithm. Nevertheless, the gain in Figure 8(a) is still up to 32.5% when the reducers number is 180.

In summary, our Network-aware solution always outperforms the other three algorithms by achieving the lowest max shuffling time. It also achieves almost the same max reducer load to the LPT algorithm that focuses on optimizing the load balancing. Moreover, our solution can adapt to various amount of available reducer resources and resist well to diverse data skewness with excellent scalability on the data size.

V. Further Discussion

It is worth noting that the shuffle subphase of MapReduce can overlap with the map phase, i.e., shuffle can start before all the map tasks finish, which may reduce the finishing time of the reduce phase. However, the early start of the shuffle subphase will occupy the task slots and do nothing but pull data, which may lead to performance degradation of map tasks. In addition, once the shuffle subphase starts earlier than the end of the map phase, for each new key in the generated (key, value) pairs, we have to dynamically decide which machine it should be assigned to, which can become more challenging for optimization as we do not have the actual size of each cluster yet. It is thus interesting to further investigate such issues as how early the shuffle subphase can start so as to have an overall performance improvement.

Also, the algorithms proposed in this paper are centralized solutions. Given that the current MapReduce often has a centralized entity for management, our solution can be implemented there, thus being practically useful for real-world big data processing. It will also work well in a software defined networking environment [18] that allows centralized management on network services and performance. On the other hand, in certain situations, e.g., for better scalability or avoiding the single point of failure, a solution that can route the data through the network in a distributed manner (as the current fat-tree routing mechanism does) may be preferred. Yet, this may cause the performance degradation as we now need to make distributed decisions with partial or imprecise information about the network. Therefore, it is worth further exploring on how to deal with these issues and integrate our idea into a distributed mechanism.

We are also investigating other practical issues on implementing our algorithm in real world MapReduce packages, particularly Hadoop [19], one of the most popular open source implementations. We have set up a server cluster for this purpose, which is a fully controlled and configurable environment with homogeneous machines. In the long run, we expect to move the implementation to the public cloud environment.

The real world network interference and unfairness under the different users of the public cloud computing will have to be accommodated, together with the heterogeneous and instable machine resources. These new factors would all affect the overall job finishing time, and the problem formulation as well as solution will have to be refined to reflect these changes.

VI. Related Work

MapReduce [1] has emerged as a powerful tool to handle the the rapid growth of data-intensive applications in datacenter computing. Nevertheless, MapReduce has inherent limitations on its performance and efficiency. This is partially because it has not been well studied and fine-tuned as compared to the conventional tools. As a consequence, there have been many studies towards improving the performance of MapReduce, especially on the load balancing issue during the shuffle subphase.

Recent studies on the shuffle subphase have shown that skewed loads may be introduced towards the reduce tasks. The straggler problem was first described and studied in [1]. It is shown that the straggler problem in MapReduce is caused by the Zipf distribution of the input or intermediate data [20]. There are researches on reducer's slow-start-synchronization barriers [12], locality-aware and fairness-aware key partition [6], Skewtune [2], and online load balancing [3]. There is a flourish of works on minimizing the high data moving cost during the shuffle subphase. These works mostly focus on how to put the computation nodes and the storage nodes as close as possible (e.g. Purlieus [21]), or move data as small as possible, e.g. improving data locality in reduce task [11] and LARTS [7].

These studies have made assumptions on the network traffic and performance such as the data movement is mostly affected by the hop count, there is no background network traffic, or there is abundant bandwidth in a datacenter network that inter-connects servers. However, the data transmission time in the real world is bounded by the bottleneck link, which has the maximum finishing time to transmit the data, and could be significantly affected by the task placement, i.e, an inappropriate reduce task placement may lead to significant traffics on certain links and cause network bottlenecks.

VII. Conclusion

In this paper, we for the first time examined the datacenter-network-aware load balancing problem in the shuffle subphase of MapReduce. Different from earlier studies on load balancing that assumed the network inside a datacenter is of negligible delay and infinite bandwidth, we considered the traffic and bottlenecks in real datacenter networks by introducing the constraints on available network bandwidth, and demonstrated that the corresponding problem can be decomposed into two subproblems for network flow and load balancing, respectively. We proposed effective solutions to both of them, which together yield a complete solution toward near optimal datacenter-network-aware load balancing. A much simpler yet performance-wise comparable greedy algorithm was also developed for fast implementation in practice. The performance evaluation results confirmed the superiority of our algorithms.

VIII. Acknowledgement

Jiangchuan Liu's research is supported by a Canada NSERC Discovery Grant and an NSERC Strategic Project Grant. Feng Wang's research is supported by a Start-up Grant from the University of Mississippi, MS. Funda Ergün's research is supported by a Canada NSERC Discovery Grant and a PIMS Collaborative Research Grant.

REFERENCE

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[2] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD*, Scottsdale, Arizona, USA, 2012.
[3] Y. Le, J. Liu, F. Ergun, and D. Wang, "Online load balancing for mapreduce with skewed data input," in *Proceedings of IEEE INFOCOM*, Toronto, Canada, 2014.
[4] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Handling data skew in mapreduce," *SciTePress*, 2011.
[5] S. R. Ramakrishnan, G. Swart, and A. Urmanov, "Balancing reducer skew in mapreduce workloads using progressive sampling," in *Proceedings of the 3rd ACM SOCC*, San Jose, California, 2012.
[6] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud," in *2nd IEEE CloudCom*, Indianapolis, Indiana, 2010.
[7] M. Hammoud and M. Sakr, "Locality-aware reduce task scheduling for mapreduce," in *3rd IEEE CloudCom*, Athens, Greece, 2011.
[8] A. Greenberg, J. R. Hamilton, 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 *Proceedings of the ACM SIGCOMM*, Barcelona, Spain, 2009.
[9] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proceedings of the 9th ACM IMC*, Chicago, Illinois, USA, 2009.
[10] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM IMC*, Melbourne, Australia, 2010.
[11] J. Tan, S. Meng, X. Meng, and L. Zhang, "Improving reducetask data locality for sequential mapreduce jobs," in *2013 Proceedings of IEEE INFOCOM*, Turin, Italy, 2013.
[12] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Load balancing in mapreduce based on scalable cardinality estimates," in *28th IEEE ICDE*, Washington, DC, USA, 2012.
[13] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan, "Choreo: Network-aware task placement for cloud applications," in *Proceedings of the 13th ACM IMC*, Barcelona, Spain, 2013.
[14] F. Shahrokhi and D. W. Matula, "The maximum concurrent flow problem," *J. ACM*, vol. 37, no. 2, pp. 318–334, Apr. 1990.
[15] J. Kleinberg and E. Tardos, *Algorithm design*. Pearson Education India, 2006.
[16] "Wikipedia page-to-page link," *http://haselgrove.id.au/wikipedia.htm.*
[17] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of the ACM SIGCOMM*, Seattle, WA, USA, 2008.
[18] O. N. Fundation, "Software-defined networking: The new norm for networks," *ONF White Paper*, 2012.
[19] "Apache hadoop," *http://hadoop.apache.org/.*
[20] J. Lin, "The curse of zipf and limits to parallelization:a look at the stragglers problem in mapreduce," in *The 7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2009.
[21] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: locality-aware resource allocation for mapreduce in a cloud," in *Proceedings of SC*, Seattle, Washington, 2011.