k-Balanced Sorting and Skew Join in MPI and MapReduce

Silu Huang, Ada Wai-Chee Fu

Department of Computer Science and Engineering, Chinese University of Hong Kong
slhuang,adafu@cse.cuhk.edu.hk

***Abstract*—**
We consider algorithms for sorting and skew equi-join operations for computer clusters. The proposed algorithms achieve the best known theoretical workload balancing guarantee, and exhibit close to optimal balancing in our experiments. Our empirical studies also show that the proposed sorting algorithm is up to 30% faster than the state-of-the-art algorithm.

I. Introduction

A Computer cluster consists of a set of computers or nodes connected to each other by a high speed local area network (LAN). Cluster computing has emerged as a commonly used infra-structure for efficient big data computation because of the elasticity of the cluster size and low cost CPUs.

We consider the design of parallel algorithms for the basic data management problems of sorting and join operation on two tables with skew key distributions, with the models of MPI and MapReduce for computer clusters. One important property of the algorithms is the balancing of the workload among the machines in the cluster. We deal with algorithms where the runtime for each machine depends in the same way on the sizes of its input or output, which are assumed to be big. Thus, we define the workload of a machine as the amount of input to or the output from the machine, whichever is bigger. Given t machines, we bound the workload W_i on each machine to within k times that of the bigger of the average input and output sizes.

$$W_i \leq k\,(\max(N_{in}, N_{out})/t) \tag{1}$$

where N_{in} is the input size, and N_{out} is the output size. For our algorithms, the running time of each machine depends mainly on the workload. We say that the algorithm is k-balanced. We propose sorting and skew join algorithms that are 2-balanced. Our experiments show that they achieve close to perfect workload distribution in all our test cases.

II. Sorting

We are given a set $\mathcal{S}$ of n objects, where each object is a real number. Our goal is to sort the n objects with a computer cluster. For simplicity the objects themselves are the sort keys. Let there be t machines in the cluster, namely, $\mathcal{M}_1, \mathcal{M}_2, ..., \mathcal{M}_t$. For simplicity we assume that n is a multiple of t, and let $m = n/t$. This assumption can be easily removed by padding some dummy objects to S. We assume that initially the n objects are evenly distributed to the t machines, so that each machine is assigned m objects.

A. Terasort - a randomized algorithm

Terasort is a parallel algorithm proposed to sort data in the size range of terabytes [9]. There are 3 rounds in Terasort: (1) a random sample set is collected from the input. (2) From the sample set, range boundaries are determined for t contiguous but disjoint ranges that partition the data according to the sort key values. (3) The objects that fall into a particular range are sent to a corresponding machine. Each machine, $\mathcal{M}_i$, will then sort the objects received, $\mathcal{S}_i$, so that combining the results of all machines gives a sorted result of the given dataset.

An interesting and useful result is derived in [11] showing that if the sampling probability ρ is set to $1/m \ln(nt)$, then with high probability, the number of objects distributed to each machine is $O(m)$. From their proof of this result, we can derive that Terasort is 32-balanced with high probability. To show that the algorithm is k-balanced for a small k with high probability, we would bound the load distribution, $|\mathcal{S}_i|$, by km for a small k, with high probability. To this end, we first make some changes to the above in the randomization step (Round 1). We replace the step of sampling each object by Algorithm $\mathbb{S}$ below. Algorithm $\mathbb{S}$ always returns exactly $\lceil \ln(nt) \rceil$ objects.

Algorithm $\mathbb{S}$ [6], [7]: Given objects $o_1, ..., o_m$, initially no object is selected. Next consider objects one by one from o_1 to o_m, when considering object o_k, let j be number of objects already selected, select object o_k with probability $(\lceil \ln(nt) \rceil - j)/(m - k + 1)$.

From Theorem 1 below, Terasort with Algorithm $\mathbb{S}$ is close to 5-balanced with high probability.

Theorem 1. *Given $n \geq 4t$ as input size for Terasort with Algorithm $\mathbb{S}$, $|S_i| \leq 5m + 1$ with probability at least $1 - 1/n$.*

Proofs for the theorems of this paper can be found in [5].

B. SMMS sorting - a deterministic algorithm

Our proposed parallel algorithm is called SMMS (Sort-Map-Merge Sorting). The idea is that if we evenly divide the data into subsets S_i, and sort each S_i first, we may derive better range boundaries than random sampling. We have implemented the algorithm on MPI. Note that if implemented on Hadoop, the sorting by Hadoop can be turned off in the last merging step. In the first round, each machine samples $s + 1$ objects as follows. The $m = n/t$ objects $\mathcal{S}_i$ in each machine $\mathcal{M}_i$ are sorted and divided into s equi-depth (equi-frequency) intervals. Let the objects received by $\mathcal{M}_i$ in sorted order be $o_1, o_2, ..., o_m$. $\mathcal{M}_i$ picks $s+1$ sample objects $\lambda_{i,0}, \lambda_{i,1}, ..., \lambda_{i,s}$, where $\lambda_{i,0} = o_1$, and $\lambda_{i,j}$ is the $\lceil j * m/s \rceil$-th smallest object

SMMS Sorting - a deterministic algorithm

Round 1: $\mathcal{S}$ is evenly distributed among t machines. Each machine $\mathcal{M}_i$ handle a subset $S_i \subset \mathcal{S}$, where $|S_i| = n/t = m$. On each $\mathcal{M}_i$, sort subset S_i locally and pick $\lambda_{i,0}$, $\lambda_{i,1}$, ...,$\lambda_{i,s}$ and send to machine $\mathcal{M}_1$, where $\lambda_{i,0}$ is the smallest object in S_i and for $j > 0$, $\lambda_{i,j}$ is the $\lceil j * m/s \rceil$-th smallest object in S_i.

Round 2: $\mathcal{M}_1$ receives $\{\lambda_{i,j}, 1 \le i \le t, 0 \le j \le s\}$.
$\mathcal{M}_1$ selects global boundary numbers b_0,b_1,...b_t. Each interval $[b_i, b_{i+1})$ is called a **bucket**. The selection is obtained by Algorithm 1. $b_0, b_1, ..., b_t$ are sent to all machines.

Round 3: Every $\mathcal{M}_i$ sends the objects in $[b_{k-1}, b_k)$ from its local storage to $\mathcal{M}_k$, for each $1 \le k \le t$. Every $\mathcal{M}_i$ merges objects received in sorted order.

Fig. 1. SMMS sorting Algorithm

in S_i. Thus, $\lambda_{i,1} = o_{\lceil m/s \rceil}$, $\lambda_{i,2} = o_{\lceil 2m/s \rceil}$, ..., $\lambda_{i,s} = o_m$. $s+1$ is the sampling size, and s is a multiple of t. Let $s = rt$, where $r \ge 1$ is a small integer. The sampled objects are sent to machine $\mathcal{M}_1$.

In Round 2, $\mathcal{M}_1$ collects all the sample objects from every machine and then computes $t + 1$ global key boundaries $b_0, b_1, ..., b_t$, so that each interval $[b_i, b_{i+1})$ forms a bucket β_{i+1} and the intervals partition the data set. Each data objects belongs to one bucket. The algorithm to compute the boundaries will be described in the next subsection. The boundaries are sent to all machines. In Round 3, each machine distributes the sorted data S_i according to the bucket boundaries, so that data belonging to bucket β_i go to machine $\mathcal{M}_i$. $\mathcal{M}_i$ merges the data coming from other machines to form the sorted list for bucket β_i. The sorted lists from all machines form the sorted result set. The pseudocode for SMMS is given in Figure 1.

1) Algorithm 1: computing bucket boundaries: Algorithm 1 is used to compute the global boundary values of $b_0, ..., b_t$ in Round 2 of the SMMS Algorithm. The input to this algorithm consists of the local boundary values $\lambda_{i,j}$ from each machine $\mathcal{M}_i$. In this computation, we apply linear interpolation for each interval $[\lambda_{i,j}, \lambda_{i,j+1})$ on each $\mathcal{M}_i$. Since there are m/s objects in the interval by construction, we compute the mean value $\mu_{i,j} = (m/s)/(\lambda_{i,j+1} - \lambda_{i,j})$ for Algorithm 1. We also set $\mu_{i,s} = 0$, $1 \le i \le t$.

Each interval $[b_i, b_{i+1})$, $0 \le i < t$ is called a **bucket**. We use the term *bucket density* for the number of objects in a bucket, denoted by $\mathcal{D}[b_k, b_{k+1})$, $0 \le k < t$. Note that b_k is not necessarily an input object, where $0 \le k \le t$. The selection ensures that the estimated bucket density based on $\mu_{i,j}$, $1 \le i \le t, 1 \le j \le s$, for $\mathcal{D}[b_k, b_{k+1})$ is equal to m, where $1 \le k < t$. A priority queue Q is maintained for storing triplets of the form $\langle \lambda, i, \mu \rangle$, which are sorted by the first value λ as the key. In the triplet $\langle \lambda, i, \mu \rangle$, λ and μ correspond to a certain pair of ($\lambda_{i,j}$, $\mu_{i,j}$) values from $\mathcal{M}_i$. Variable cur keeps count for the estimated density of the current bucket until it reaches m, in which case, a new boundary $b[k]$ is determined. The value of cfd is for the combined frequency distribution over the value domain. An example is shown in Figure 2. Here we have 2 machines, given 40 objects, each machine samples 3 objects, namely, (2,3,4) and (1,6,7), respectively, from the distributed objects. Algorithm 1 processes the λ values in the order of 1, 2, 3, 4, 6, 7. When 4 is processed, the cur value exceeds $m = 20$, the value 3.5 is computed as the bucket boundary at Line 9.

Algorithm 1: Computing Bucket Boundaries

Input : $\lambda_{i,j}, \mu_{i,j}$, $1 \le i \le t, 0 \le j \le s$
Output : Global boundaries $b[k]$, $0 \le k \le t$

1 Initialize: Create an empty priority queue Q; $\forall 1 \le i \le t$: $pastfd[i] = 0$; $next[i] = 0$; push $\langle \lambda_{i,0}, i, \mu_{i,0} \rangle$ into priority queue Q; $cfd = 0; pre = 0; cur = 0; k = 0; flag = 0$;
2 **while** $Q \neq \emptyset$ **do**
3 $\langle \lambda, i, \mu \rangle \leftarrow$ TopAndPop(Q); /*λ and μ from $\mathcal{M}_i$ */
4 **if** $flag == 0$ **then**
5 $b[k] = \lambda, k++, flag = 1$; /*first boundary*/
6 **if** $(\lambda - pre) \times cfd + cur < m$ **then**
7 $cur += (\lambda - pre) \times cfd$; /*keep count*/
8 **else**
9 $b[k] = (m - cur)/cfd + pre, k++$; /* new bucket */
10 $cur = (\lambda - pre) \times cfd + cur - m$; /* keep count for new bucket*/
11 $pre = \lambda$; /*update previous boundary*/
12 $cfd = cfd - pastfd[i] + \mu$; /* update cfd */
13 $pastfd[i] = \mu$; /*μ will be obsolete for $\mathcal{M}_i$*/
14 **if** $!next[i]$ **then**
15 push$\langle \lambda_{i,next[i]}, i, \mu_{i,next[i]} \rangle$into $Q, next[i]++$;
16 $b_t = \lambda$ **return** $b[k], 0 \le k \le t$;

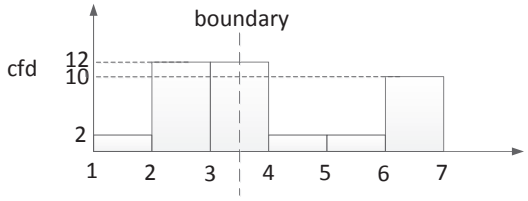


Fig. 2. Example of Combined Frequency Distribution (cfd): $n = 40$, $t = 2$, $s = 2$, $m = 20$, samples from $\mathcal{M}_1$ = (2,3,4), samples from $\mathcal{M}_2$ = (1,6,7)

Each while loop handles one sampled value λ_{ij}. There are at most t elements in Q, hence each while loop costs $O(\log t)$ time. The total time complexity is $O(st \log t)$ because of $t(s+1)$ rounds of the while loop.

We should point out that the complexity of Algorithm 1 is insignificant compared to the problem size. Utilization of computer cluster is justified only when the problem size is big, and from previous works such as [11], the size is in terms of billions of records and is 20 GB or more. Thus, the value of t is very small in comparison to n. In our experiments, the runtime for Round 2, including Algorithm 1, is found to be

negligible for all test cases.

2) Analysis: In the first round of SMMS, all machines are assigned equal workload. In Round 2, the workload is the $t(s+1)$ samples which is small compared to the input size, n. Hence, we need only analyze the workload distribution at Round 3. We aim for a bound on the maximum workload of a machine when compared to the even workload.

Theorem 2. *At Round 3 of SMMS sorting, the workload of each machine is bounded by* $(1+2/r+t^2/n)m$.

For example, if $n \geq 25M$, $r = 2$, and $t = 50$, then the workload for each machine is bounded above by $\approx 2m$, and it is 2-balanced. If $n \geq 75M$, $r = 6$, and $t = 50$, then this bound becomes $\approx 1.3m$, and SMMS is 1.3-balanced.

Two k-balanced algorithms are comparable when they have similar operations at each machine. SMMS and Terasort both involve only sorting and distribution of data as the major steps. In comparison, Terasort has up to 60% imbalance empirically, which is higher than the above bound for SMMS.

The global boundaries are related to quantiles of an ordered sequence of data values. The ϕ-quantile is the element α with rank $\lfloor \phi N \rfloor$, where N is the given number of values [1], [4]. We adopt a two phase approach as in [1]. However, our problem is for boundaries instead of ranks, which allows us to apply linear interpolation in the computation.

III. Skew Join

In the recent development of the Apache Pig system on top of MapReduce, it has been noted that data skew in join is a significant and challenging problem [3]. In this section, we focus on the problem of join when data is skew.

We consider the problem of skew join for two tables S and T with an equality join condition of $S.\rho = T.\rho$ for a certain join key ρ. As in [8], we model the join result by means of a $|S| \times |T|$ **join-matrix** Γ as shown in Figure 3(b). In this matrix, S and T are sorted by the join key into ordered lists $\overrightarrow{S} = s_1, s_2, ..., s_{|S|}$, and $\overrightarrow{T} = t_1, t_2, ..., t_{|T|}$. In Figure 3(b), the key values for $s_1, ..., s_{|S|}$ are b, d, d, d, d, f, correspondingly. The matrix entry $\Gamma(i, j)$ is true (shaded) iff $s_i.\rho = t_j.\rho$. The join result for a certain join key k form a shaded rectangular region in Γ, we call this region the **join result** for k, or simply *result*(k). For example, in Figure 3(b), the join result for key d, denoted by $result(\mathrm{d})$, is the shaded rectangle of size $4 * 3$.

Suppose k is a join key, we say that the **size of the join result** for k is $M \times N$ if M and N are the number of tuples with key k from S and T, respectively. For example, in Figure 3(b), the join result for key d has size 4×3, which is the cross product of tuples 2 to 5 from S and 2 to 4 from T. Next we define the skew factor to indicate how large the join result size is compared with the total size of S and T, where size is measured by the number of tuples.

Definition 1 (Join Skew Factor σ)**.** *The skew factor of the join, $S \bowtie T$, of two tables S and T is given by σ if $|S \bowtie T| = \sigma(|S| + |T|)$.*

A. StatJoin - A Deterministic Algorithm

In this section we introduce a deterministic algorithm **StatJoin** for handling the skew join problem The major idea for StatJoin is the partitioning of data based on *statistical* information.

1) Statistics Collection: In Algorithm StatJoin, we first collect statistics from the two tables S and T. For this purpose, we apply a parallel sorting algorithm such as Terasort or SMMS for each of S and T, allowing for repeated keys. After sorting, each $\mathcal{M}_i$ contains sorted portions or buckets $\mathcal{P}_i^S$ and $\mathcal{P}_i^T$ of S and T, respectively. All occurrences of the same join key will be collected at one single machine. Then each machine calculates the sizes of the join results for different join keys, and the total join result size that will be generated from $\mathcal{P}_i^S$ and $\mathcal{P}_i^T$. The result sizes are measured in number of tuples. Based on such statistics, a task distribution algorithm is applied on all the join tasks.

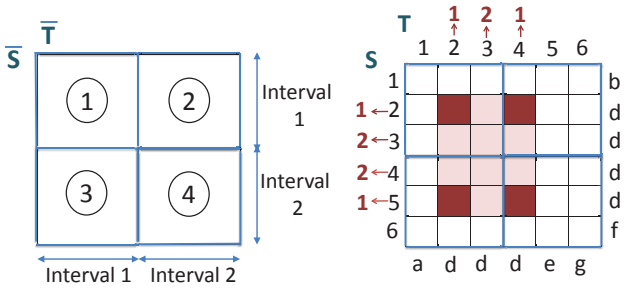
Let W be the total join result size. A join result of a key with a size greater than W/t is called a **big join result**, otherwise, it is called a **small join result**. Note that the biggest size of a small join result is W/t. We decide on the task distribution by first considering the big join results, followed by the consideration of the small join results.

Although the statistics collection requires a sorting of the input datasets, this overhead is insignificant when compared to the overall runtime, because for skew join the input size is small when compared to the result size.

2) Big Join Results: We consdier the big join results one at a time, in an arbitrary order. Let B be a big join result with a size of $M \times N$, where $(j-1)W/t < MN \leq jW/t$. We apply a **result-to-machine** mapping method for B with the number of machines set to j. Without loss of generality, let the machines assigned be $\mathcal{M}_1, ..., \mathcal{M}_j$. The result of the mapping is that each machine $\mathcal{M}_i$ will be mapped to a rectangular region in the join result B. Each rectangular region is defined by a quadruple $\langle l_s^i, h_s^i, l_t^i, h_t^i \rangle$, where l_s^i, h_s^i are two tuple id's in table S, where $l_s^i < h_s^i$, and l_t^i, h_t^i are two tuple id's in table T, where $l_t^i < h_t^i$. A tuple in table S with id in $[l_s^i, h_s^i]$ is assigned to $\mathcal{M}_i$. Similarly, a tuple in T with id in $[l_t^i, h_t^i]$ is assigned to $\mathcal{M}_i$. For example, in Figure 3 (b), suppose we divide the join result horizontally into 2 equal sized rectangles. The top rectangle is defined by $\langle 2, 3, 2, 4 \rangle$. Suppose this rectangle is assigned to machine $\mathcal{M}_2$. Then tuples 2 and 3 of S, and tuples 2, 3, and 4 of T will be assigned to $\mathcal{M}_2$.

We divide the MN result tuples among j machines by partitioning the longer side of the rectangle B into j intervals as evenly as possible. Without loss of generality, assume $M \geq N$. Then M is divided into j intervals. Each of the j intervals and the side of size N of region B form a rectangle in B. Hence B is partitioned into j such rectangles. We call these rectangles the **mapping rectangles**. There are two possible cases for the size of MN:

1) $MN = jW/t$. In this case, the j mapping rectangles are of the same size, W/t. The output of each mapping rectangle are assigned to one of j machines that have



(a) (b)

Fig. 3. (a) machine matrix A for 4 machines ($t = 4$), $a = b = 2$. (b) join matrix Γ and randomized tuple-to-interval mapping

not been assigned any big join result so far. We send the N tuples on the T side of B and tuples along interval i, $1 \leq i \leq j$, on the S side of B, to $\mathcal{M}_i$.

2) $MN < jW/t$. Since we partition the longer side of B (with M tuples) as even as possible, each interval has either $\lceil M/j \rceil$ or $\lfloor M/j \rfloor$ tuples. Thus, the smallest mapping rectangle R_{min} has a size smaller than W/t. For each of the $j-1$ mapping rectangles other than R_{min}, the corresponding tuples are processed as in Case (1) above, so that their output are assigned to $j-1$ machines. For R_{min}, it is treated as a small join result, which is to be processed as described in the next subsection. We call R_{min} a **residual join result**.

3) Small Join Results: After the big join results are assigned to the machines, we deal with the **result-to-machine** mapping for the small join results. The small join results include those residual join results. We consider small join results for different join keys one by one, each time we assign the next join result to the machine with a smallest assigned workload.

The work assignment resembles the greedy bin-packing algorithm and hence we have the following result.

Theorem 3. *Let the total join results size be W. With StatJoin, the total size of the join results generated by any machine is at most $2W/t$.*

B. RandJoin- A Randomized Algorithm

In this subsection, we introduce our randomized algorithm, RandJoin, for handling skew join. In this algorithm we assign tuples to machines in a randomized approach. A randomized algorithm is proposed in [8] which maps square regions of the join result to machines. However, since the join result may not be a perfect square, the mapping does not ensure equal probabilities of assignments, and results in a maximum workload imbalance factor of 4. Here we propose another mapping technique where each tuple has equal expected probabilities for the assignments. Our method is based on a conceptual machine matrix.

1) Machine Matrix A: Let the number of machines be t, we determine two integers a and b such that firstly, $a \times b = t$ and secondly, among all a, b satisfying $a \times b = t$, $a|T| + b|S|$ is minimized. We shall see that $a \times b = t$ is a sufficient condition for our workload balancing guarantee. The minimization of $a|T| + b|S|$ can lead to some minor improvement for load balancing related to the join input size to each reducer. The reason for this choice will be explained later. With the values of a and b, we form a $a \times b$ matrix A called the **machine matrix**. For matrix A, we call the first dimension $\overline{S}$ and the second dimension $\overline{T}$. Each $A[i, j]$ is assigned a unique machine. We say that $A[i, j]$ lies on **interval** i of $\overline{S}$ and **interval** j of $\overline{T}$.

Example 1. *Fig.3(a) shows the machine matrix A given 4 machines. The two dimensions of $\overline{S}$ and $\overline{T}$ each consists of 2 intervals, i.e., $a = b = 2$. Machines $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$, and $\mathcal{M}_4$ are assigned to $A[1, 1]$, $A[1, 2]$, $A[2, 1]$, and $A[2, 2]$, respectively.*

2) Tuple-to-Interval Mapping: We assign tuples to machines by a randomized algorithm. For each tuple in S we randomly select an integer i in $1, ..., a$ and map the tuple to interval i of $\overline{S}$ in the machine matrix A. For each tuple in T, We randomly select an integer j in $1, ..., b$ and map the tuple to interval j of $\overline{T}$ in A. Then each tuple is assigned to the machines as follows: if an S tuple x is mapped to interval i of $\overline{S}$ in matrix A, then x is sent to each of the b machines assigned to $A[i, 1]$, $A[i, 2]$, ..., $A[i, b]$. If a T tuple y is assigned to interval j of $\overline{T}$ in A, then y is sent to each of the a machines assigned to $A[1, j]$, $A[2, j]$, ..., $A[a, j]$. Each machine computes the cross-product of all the S tuples and T tuples that it has received for a single join key. Hence, the join result for tuples x and y, if any, will be uniquely generated by the machine assigned to $A[i, j]$.

Example 2. *: In Figure 3(b), we show the join matrix for the tables S and T. Each table contains 6 tuples. We show that tuples 2,3,4,5 of S are randomly assigned interval numbers 1,2,2,1. Then the second tuple of S will be mapped to the first interval on $\overline{S}$ in matrix A in Figure 3(a), and it will be sent to machines $\mathcal{M}_1$ and $\mathcal{M}_2$. The join result in the join matrix for the darker shaded area will be generated by machine $\mathcal{M}_1$.*

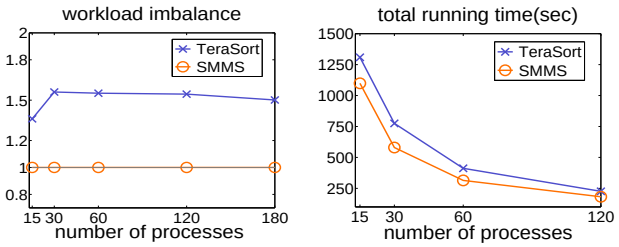
From the above tuple-to-interval mapping, each tuple in S is assigned to b machines, and each tuple in T is assigned to a machines. By selecting a and b that minimize $a|T| + b|S|$ we minimize the total input size to the machines in the number of tuples.

The following theorem establishes that RandJoin is 2-balanced with high probability.

Theorem 4. *If the join results for each join key is either an empty set or a set with size $M \times N$ where $M/a \geq 300$ and $N/b \geq 300$, then the probability that the workload of any machine is less than twice the even workload is more than $1 - 1.2 \times 10^{-9}$.*

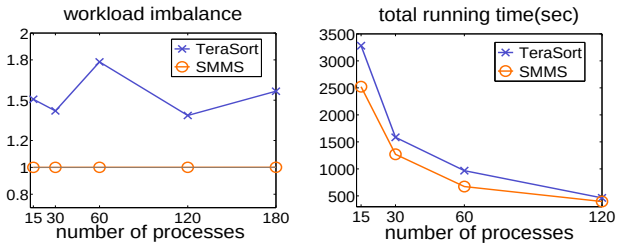
IV. Experimental Results

Our experiments for the parallel algorithms have been conducted on a 16 machine cluster with a master machine and 15 slave machines. The master is a Dell R720 Server with



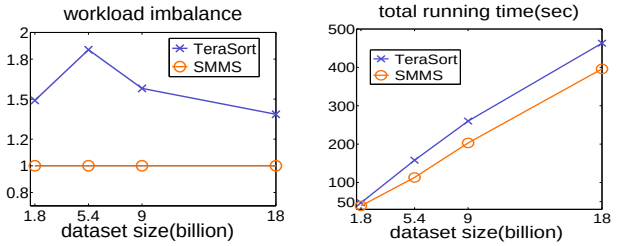
(a) workload (input size) (b) run time

Fig. 4. Sorting for real dataset LIDAR, workload imbalance = maximum workload / optimal workload



(a) workload (input size) (b) run time (sec)

Fig. 5. Comparing SMMS and Terasort for Random Dataset with 18 billion objects (199.3 GB)



(a) workload (input size) (b) run time (sec)

Fig. 6. Sorting results for Random Datasets of different sizes with 120 processes

Dual 6-core Xeon E2620 2.0GHz, 192GB RAM and 4x 3TB SAS Hard Disk. Each slave machine is a Dell R620 Server - Dual 6-core Xeon E2620 2.0GHz, with 48GB RAM and 2x 300GB SAS Hard Disk. All machines are connected by a 1GB-ethernet switch. We have installed Hadoop (version 1.2.1) on the cluster for MapReduce algorithms. There are 6x2x15 = 180 cores in the slaves, we can activate up to 180 workers in parallel for Hadoop mappers or reducers.

We have implemented the sorting algorithms (SMMS and Terasort) based on MPI, and the join algorithms RandJoin and StatJoin based on Hadoop MapReduce. We have set the DFS dfs.replication factor to 3. The fs.block.size is set to 64MB. Other Hadoop parameters are set to the default values. The computer cluster consists of 15 worker machines each with 8 cores that share 2 hard disks. We shall call the parallel computational units *processes* instead of *machines* in our experiments.

We evaluate our algorithms by two measurements: the workload distribution and the runtime. The workload is measured by the input size for sorting and by the result size for join. The sizes are given in the number of tuples unless otherwise specified. We examine the **workload imbalance** which is given by the ratio of the maximum workload on a machine versus the even workload. The **runtime** is given by the longest runtime taken by any process.

A. Results for Sorting

We evaluate the sorting algorithms of SMMS and Terasort on a real dataset LIDAR and also on a synthetic dataset. For SMMS, we set the value of r to 1 so that each process samples t objects. We vary the number of processes from 15 to 180, and measure both the workload distribution and the runtime performance.

Real Data: We use the real dataset LIDAR[1] for experiments on sorting, which has been used for the sorting experiments in [11]. LIDAR contains 8.27 billion records, each of which is a 3D point representing a location in North Carolina. We sort the records by the first dimension. The dataset size is 123GB. The input data is distributed sequentially to the machines.

Synthetic Data : We have generated 4 sets of random data, with 1.8 billion objects, 5.4 billion objects, 9 billion objects and 18 billion objects. The sizes of these datasets are 19.9 GB, 59.9 GB, 99.8 GB and 199.3 GB, respectively. The key of each data object in a dataset is a randomly generated number in the range of $[1, 12 \times 10^6]$. We generate unique objects in each machine.

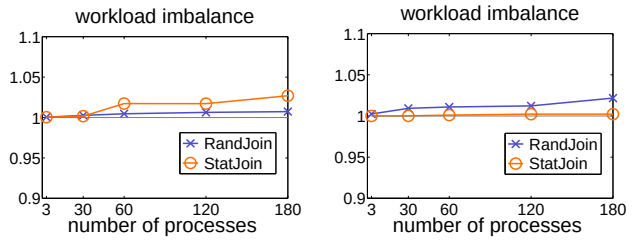
The results of workload imbalance are shown in Figures 4(a), 5(a), and 6(a). In all cases, SMMS distributes the workload very evenly and the imbalance is close to the optimal value of 1. TeraSort has comparably much larger workload imbalance, in most cases the maximum workload of a process is above 1.5 of the optimal load. Similar results are reported in [11]. The imbalance affects the performance in runtime. However, the effect is mitigated by the fact that we have a star cluster with a bottleneck at the master node, as shown in Figures 4(b), 5(b), and 6(b). The improvement in runtime by SMMS is expected to be more significant in a decentralized setting. The figures also show that SMMS achieves almost linear speedup.

B. Results on Skew Join

For the Skew Join experiments the dataset consists of two input tables S and T. We adopt two different methods to form a dataset with skew join keys. The first method is to generate tables with attributes drawn from the Zipf distribution, maintaining the same distribution for both tables so that each key has the same frequency in both of the input tables. We vary the Zipf skew parameter θ between 0 (skew) and 1 (uniform), i.e., $Z(r) \propto 1/r^{(1-\theta)}$, where r is a frequency rank, $Z(r)$ is the frequency of the item with rank r.

The second kind of skew data is generated as described in [2]. For a table with n tuples, the join key has a domain of $[n, 2n)$. The special join key n appears in a fixed number of tuples, while the remaining tuples are randomly assigned a

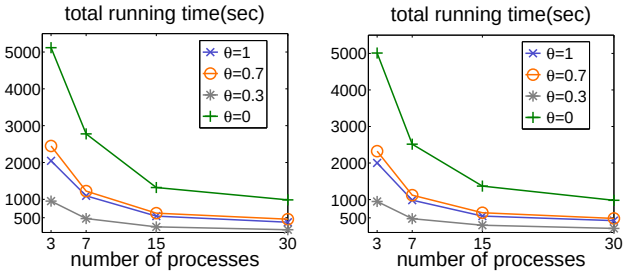
[1]Downloadable from http://www.ncfloodmaps.com



(a) θ = 0.7, $|S| = |T|$=5M output size = 147GB (29.4×10^9 tuples, σ=2940)

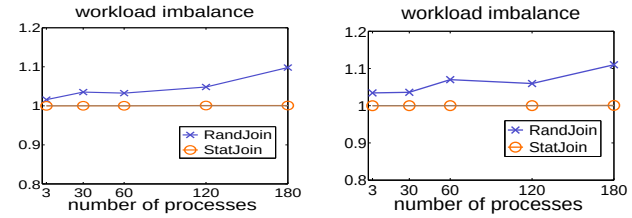
(b) $\theta = 0.3$, $|S| = |T|$=1.5M output size = 59GB (11.8×10^9 tuples, σ=3900)

Fig. 7. Workload distribution of RandJoin and StatJoin for Zipf distributions: (θ = 1 : uniform key distribution).



(a) RandJoin (b) StatJoin

Fig. 8. Running time for Zipf skew datasets (in sec)



(a) $M = 10^5$, $N = 2 \times 10^4$ (b) $M = 2 \times 10^5$ $N = 10^4$

Fig. 9. Workload distribution for scalar skew data.

join key from $[n, 2n)$. The output tuple size is 95 bytes. The skew key $k_0 = n$ is generated in both tables S and T, and it occurs M times in S and N times in T. By adjusting M and N we can control the expected output join sizes. This kind of test data is called "scalar skew" in [12] and is also used in the study in [10].

Zipf distributed dataset: We aim to compare the effect of skewness on similar join output size. However, Zipf distributions would vary the output size for the same input size. Therefore we vary the input table sizes accordingly. Following the design of [8] for skew key distribution, each tuple contains a 4 byte join key with a domain of $[1000, 1999]$.

Scalar skew dataset : We tested on two sets of scalar skew data. As in [2], we fix an output size and vary the values of M and N to examine the effect of different key skewness in the two given tables. For the first dataset, we set $M = 10^5$, and $N = 2 \times 10^4$. For the second set, we set $M = 2 * 10^5$ and $N = 10^4$. The output size of the join of S and T for both datasets is 190GB. In both datasets, $|S| = |T| = 1.5M$, and the skew factor σ is 600.

1) Runtime Analysis: The total runtimes for Zipf skew data are shown in Figure 8. The results for scalar skew data are similar. It can be seen that there is almost linear speedup up to 15 processes. due to the highly even workload distribution. The speedup effect beyond 15 processes is discounted by the overhead in the file replication of Hadoop HDFS. The file replication factor is 3, hence, with 34 hard disks, there is good speedup effects with up to 15 processes.

2) Workload Imbalance: The results of workload distribution are shown in Figures 7 and 9. For the scalar skew dataset, RandJoin did not distribute the workload as evenly when the number of processors is large, this is because the values of M/a and N/b are too small to satisfy the condition in Theorem 4. StatJoin achieves near optimal results in all cases.

V. Conclusion

We study the problems of sorting and skew join in a computer cluster environment. We propose algorithms that achieve the best known theoretical guarantees on even workload distribution. Extensive empirical study shows that our sorting algorithm performs better than the state-of-the-art method of TeraSort. All our algorithms achieve near optimal workload distribution in all test cases.

Acknowedgements

This research was supported by GRF CUHK412313 and Direct grant 2050497. The authors would like to thank the reviewers for helpful comments, James Cheng for the use of the computer cluster, Yanyan Xu, Yi Lu, Wenqing Lin and Yingyi Bu for sharing their experiences with MPI and Hadoop, the authors of [11] for their source code and a pointer to a dataset, and the authors of [8] for explaining their source code.

References

[1] K. Alsabti, S. Ranka, and V. Singh. A one-pass algorithm for accurately estimating quantiles for disk-resident data. In *VLDB*, 1997.
[2] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, 1992.
[3] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: The pig experience. In *VLDB*, 2009.
[4] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, 2001.
[5] S. Huang and A. W.-C. Fu. (a, k)-minimal sorting and skew join in mpi and mapreduce. In *CoRR (arXiv)*, 2014.
[6] T. Jones. A note on sampling a tape file. *Commun. ACM*, 5(6):343, 1962.
[7] D. E. Knuth. *The Art of Computer Programming, Volume 2 Seminumerical Algorithms 3rd Ed.* Addison Wesley, 1997.
[8] A. Okcan and M. Riedewaid. Processing theta-joins using mapreduce. In *SIGMOD*, 2011.
[9] O. O'Malley. Terabyte sort on apache hadoop. In *Technical Report, Yahoo*, 2008.
[10] E. Omiecinski. Performance analysis of a local balancing hash-join algorithm for a shared memory multiprocessor. In *VLDB*, 1991.
[11] Y. Tao, W. Lin, and X. Xiao. Minimal mapreduce algorithms. In *SIGMOD*, 2013.
[12] C. Walton, A. Dale, and R. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *VLDB*, 1991.