Parallel Computing 72 (2018) 14–28

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Comparing load-balancing algorithms for MapReduce under Zipfian data skews

Joanna Berlińska[a,*], Maciej Drozdowski[b]

[a] Faculty of Mathematics and Computer Science, Adam Mickiewicz University in Poznań, Umultowska 87, Poznań 61-614, Poland
[b] Institute of Computing Science, Poznań University of Technology, Piotrowo 2, Poznań 60-965, Poland

ARTICLE INFO

Article history:
Received 29 September 2016
Revised 8 December 2017
Accepted 19 December 2017
Available online 19 December 2017

Keywords:
MapReduce
Load balancing
Scheduling
Performance evaluation

ABSTRACT

In this paper, we analyze applicability of various load-balancing methods in countering data skew in MapReduce computations. A MapReduce job consists of several phases: mapping, shuffling data, sorting and reducing. The distribution of the work in the last three phases is data-driven, and unequal distribution of the data keys may cause imbalance in the computation completion times and prolonged execution of the whole job. We propose algorithms of four different types for balancing computational effort in reduce-heavy MapReduce jobs and evaluate their performance under various degrees of data skew and system parameters. By applying an innovative method of visualizing algorithm dominance conditions, we are able to determine conditions under which certain load-balancing algorithms are capable of scheduling MapReduce computations well. We conclude that no single algorithm is a panacea and hybrid approaches are necessary.

1. Introduction

MapReduce is a paradigm for processing big volumes of data in parallel [14,26,34]. The name MapReduce comes from the two main steps of mapping and reducing. We will call the computers executing the first phase *mappers*, and the second phase *reducers*. Mappers read the input data and transform them into a set of intermediate $(key_1, value_1)$ pairs using a user-defined *Map* function. We will call the set of pairs with equal key_1 a *cluster*. The intermediate pairs are read by reducers which group them together by key_1 using, e.g., external sort and process them using a user-defined *Reduce* function to produce a set of final $(key_2, value_2)$ pairs. A MapReduce job is supervised by a *master* machine. The master initiates mapper and reducer tasks, distributes the work by providing mappers and reducers with locations of the files with data to process, re-executes computations which take too long time. A detailed description of the execution of a MapReduce job will be given in Section 2. Applications of MapReduce can be found in text, log, measurement and image processing, machine learning, simulation and graph algorithms [12,14,23,26,32,34]. Since we consider MapReduce in general, analysis of any particular MapReduce applications is out of scope of this paper.

The routing of work pieces (clusters) from mappers to reducers is driven by data, i.e. by the key_1 values. An unequal distribution of cluster sizes is called data *skew*. A skewed distribution of cluster sizes may result in an imbalance of work assignment and big differences in reducer completion times. In more detail, data skew may impact a MapReduce job in the following ways:

* Corresponding author.
E-mail addresses: joanna.berlinska@amu.edu.pl (J. Berlińska), maciej.drozdowski@cs.put.poznan.pl (M. Drozdowski).

https://doi.org/10.1016/j.parco.2017.12.003
0167-8191/

Table 1
Summary of notation.

a^{red}	reducer computing rate in seconds per byte;
a^{sort}	reducer sorting rate in seconds per byte;
a^{master}	the master computing rate in seconds per byte;
A	computing rate of the mapping phase in seconds per byte;
C	communication rate for reading the data by the reducers;
ε	size of master messages;
f_i	frequency of the ith key;
γ	mapper result multiplicity fraction;
γ_1	reducer result multiplicity fraction;
l	bisection width limit, expressed in parallel channels;
k	number of sub-partitions in the static algorithm;
k_1, k_2	number of sub-partitions in the static and dynamic phases of the mixed algorithm;
κ_i	value of the ith key;
m	number of mappers (computers);
Ω	space of keys (the set of all key_1 values);
Π_i	partition i, a set of keys;
R	number of partitions;
r	number of reducers (computers);
s_i	size of partition $i = 0, \ldots, R-1$;
σ_i	size of κ_i cluster in bytes;
$T(s)$	reducer complexity function in partition size s; $T(s) = a^{sort} s \log_2 s + a^{red} s$
V	initial load size, in bytes;
X	fraction of keys processed in the static phase of the mixed algorithm;
z	data skew scaling parameter, see equation (1);

1. A non-uniform distribution of keys key_1 between the mappers results in unequal sizes of mapper output files and imbalance in the shuffle phase.
2. Different frequencies of key_1s result in unequal sorting and reducing times.

There may also be other sources of load imbalance, both in the map and in the reduce step: keys not equally easy to process or volatility of computer and communication speeds. These are out of scope of this study. Many approaches to load-balancing MapReduce have been proposed and it is necessary to compare them and test their limitations. Our primary interest are reduce-heavy MapReduce jobs. Therefore, this paper is dedicated to the analysis of applicability of different algorithms mitigating effects of data skew in the reduce phase. Due to the scope of the system and application configurations studied, the analysis cannot be performed by measurement. We have to recourse to mathematical models and simulation. We depart from the idea of providing "one best" load-balancing algorithm, and instead, we map conditions under which certain types of algorithms prevail. The contribution of this paper can be summarized as follows:

- Models of MapReduce performance and data skew are proposed. Power-law (Zipfian) key frequency distributions are used as more demanding and realistic.
- Four algorithms differing in the mode of operation are proposed to mitigate partition skew and implemented in a MapReduce simulator.
- By applying an innovative way of visualizing algorithm dominance relationships in the space of system and data distribution parameters, conditions under which certain load-balancing algorithms prevail are determined.

Further organization of the paper is as follows. In the next section mathematical models of MapReduce and key frequency distribution are introduced. In Section 3 related publications are outlined. Section 4 comprises algorithms designed to counteract load imbalance in the reducing phase of the MapReduce computation. The efficacies of these algorithms are evaluated in Section 6 using a simulator presented in Section 5. The last section is dedicated to conclusions. The notation used throughout the paper is summarized in Table 1.

2. Model of MapReduce and data skew

In this section we define mathematical models of MapReduce performance and key distribution. Let m denote the number of mappers and r the number of reducers. As we explained in Section 1, a MapReduce job consists in extracting and processing (key_1, $value_1$) pairs. Let Ω be the space of key_1 values, and let $\kappa_i \in \Omega$ denote the ith key_1 value. It is assumed that $|\Omega|$ is much greater than the number of reducers. The set of all key-value pairs with the same key value κ_i will be called a cluster of κ_i. The size of κ_i cluster is denoted by σ_i (in bytes).

MapReduce starts with the map phase in which the input dataset is processed in many equal-size blocks called *splits*. Processing a split is a *map task* consisting of retrieving the split, computations on its data, and distributing the results to output files so that (key_1, $value_1$) pairs with equal key_1 land in the same file. The target output file for key_1 is identified using a *partitioning function*. A set of keys for which the partitioning function returns the same value will be called a *partition*, and the number of partitions will be denoted by R. Thus, partition Π_j, for $j = 0, \ldots, R-1$, is a set of some κ_i values. The size

s_j (in bytes) of partition Π_j is $s_j = \sum_{\kappa_i \in \Pi_j} \sigma_i$. It is required that $\cup_{j=0}^{R-1} \Pi_j = \Omega$ and $\forall i \neq j$, $\Pi_i \cap \Pi_j = \emptyset$. Hence, the partitioning function usually has the form $\texttt{hash}(key_1) \bmod R$. Each map task is executed sequentially on one machine and is managed independently of other map tasks by the MapReduce environment such as Hadoop. Many map tasks run simultaneously in parallel. Though splits have discrete nature, the influence of split size on the mapping speed is negligible (cf. the discussion in [7,8]). At the end of mapping phase each mapper holds R local output files.

Let V denote the size of the input dataset (in bytes). We assume that the input is distributed roughly equally between the mappers and each mapper processes volume V/m. Moreover, the mappers process the splits with equal average speed $1/A$ (A is in seconds per byte) and hence finish in time VA/m. It is assumed that for α bytes of input data, $\alpha\gamma$ bytes of output data are produced by the mappers. Thus, the size of the intermediate data produced by all mappers is $\sum_{j=0}^{R-1} s_j = V\gamma$, and at the end of computation each mapper holds roughly $V\gamma/m$ bytes in R files for reducing. Since the sizes of the files are in general unequal, mappers can report sizes of the output files to the master. In such a case the size of a message is ε per output file.

The master decides on assigning partitions to the reducers so that a cluster of some key_1 is processed by one reducer. Typically, processing a partition is one sequential *reduce task* managed by the master of the MapReduce job. The master directs the reducers to execute reduce tasks. In the next step, called *shuffle*, reducers read the output files of the mappers. Shuffle involves communication over the shared communication network. We assume that each machine has bandwidth $1/C$, the communication network has bisection width l, and bandwidth is shared. This means that a single communication channel in the otherwise unloaded network has speed $1/C$ (C is in seconds per byte). A computer can open at most one communication channel at a time, i.e. the so-called one-port model is used. This requires a special communication organization that will be described in Section 4. Bisection width l is the number of concurrent communication channels which can be open in the communication network between pairs of different computers without bandwidth degradation. If the number of concurrently open channels exceeds l, then the bandwidth is shared. For example, if x machines simultaneously open communication channels, then the bandwidth perceived by a machine is $\min\{1, l/x\}/C$.

After reading the assigned partitions, reducers start processing them. Processing a partition consists in sorting key-value pairs by key_1, processing them and then storing the results. For α bytes of reducer input data, $\alpha\gamma_1$ bytes of final results are produced. We assume that for a partition of size s (in bytes) the reduce task has loglinear runtime $T(s) = a^{sort} s \log_2 s + a^{red} s$ representing, e.g., standard merge-sort. Parameters a^{sort}, a^{red} are processing rates (in seconds per byte) of reducer sorting and computing, respectively. Note that for the complexity function greater than linear even small differences in load distribution may easily escalate the differences in the reducer completion times. Migrating data of a running sort is a cumbersome and time consuming operation. Therefore, we assume that the sorting operation is essentially nonpreemptive and not transferable. If a partition assigned to a reducer were to be split and its data redistributed, then it can be done only after sorting. The sets of key-value pairs (key_2, $value_2$) created by the reducers are stored in the distributed file system for subsequent use, possibly by yet another MapReduce job.

In this paper, we assume that key frequency distribution follows power law. In more detail, the key ranked d most frequent among $|\Omega|$ different keys has expected frequency

$$f(d) = \frac{1/d^z}{\sum_{i=1}^{|\Omega|} 1/i^z}, \tag{1}$$

where $z \geq 0$ is a scaling parameter controlling the skew of the frequencies. Note that for $z = 0$ equation (1) describes the uniform distribution, and for $z > 1$ we obtain Zipfian distribution. Thus, by assuming that $z \geq 0$ our frequency distribution model extends the Zipf's law into the area of scaling parameters $z \leq 1$ and is applicable to small, medium, as well as big data skew. For conciseness of presentation we will be saying that key frequency distribution is Zipfian. Let us note that the distribution of keys and key frequencies f_i in the input data are not known in advance. In a sense, the skew mitigating algorithms have to discover them. In consequence, the value of z cannot be used by the algorithms assigning load to reducers.

3. Related work

MapReduce has been introduced in the seminal paper [14]. One of the most popular MapReduce implementations is Apache Hadoop [6,34]. MapReduce is becoming an industry-standard component of the so-called NoSQL databases [2,18,29] and libraries for big data processing [3,4].

In order to characterize MapReduce workloads, logs from 171,079 Hadoop jobs run by 31 different users were analyzed in [20]. It has been found that while most of the MapReduce jobs have relatively equal runtimes of map tasks or reduce tasks, a significant fraction of jobs exist where runtimes are unequal. Moreover, runtime dispersion was greater in case of reducing. An attempt at predicting MapReduce job runtime resulted only in a limited success because large prediction errors appeared ($> 230\%$ mean relative prediction error for successful jobs) due to performance problems in the application or specific input data [20]. Authors of [25,28] argue that practical workloads very often have Zipf-like or power law distributions of object frequencies and sizes. This signifies the need for reacting online to data skew and other performance problems.

Implementations of MapReduce have provisions for a failure or prolonged processing of one split in mapping or one partition in reducing. If such a task fails or progresses too slowly, then it can be re-executed speculatively. This mechanism

Table 2
Numbers of instances in which certain algorithm dominates [10].

Algorithm	Hard, loglinear sort	Easy, loglinear sort	Hard, linear sort	Easy, linear sort	Total
static(2)	10	10	215	29	264
static(10)	62	14	**309**	0	385
mixed(0.5)	50	**550**	50	223	**873**
mixed(0.9)	**488**	27	30	3	548
multi-dynamic	0	9	6	**355**	370

can be used for improving MapReduce performance in heterogeneous systems [35]. Unfortunately, speculative re-execution is not helpful in the case of data skew because re-execution of the partition with skewed data will result in exactly the same prolonged computation on another computer.

The performance of MapReduce has been studied analytically in [8] using so-called divisible load model [1,13,15]. Methods of calculating effective load partitioning have been proposed, assuming arbitrary flexibility in partitioning keys and taking into account system performance parameters. It has been concluded that the total MapReduce execution time is determined by the balance of computational demand between mapping and reducing. When the amount of data produced by mappers and shifted to reducers is big, then the time of shuffling and reducing easily dominates the whole computation time. This approach has been generalized in [9] to calculate optimum schedules for iterative MapReduce jobs. The scheduling method developed in [9] is useful for discovering effective scheduling patterns which can be implemented in heuristics, but it has high computational complexity and hinges on reliable performance indicators, whose availability is a strong limitation considering the nature of shared computer clusters and ever-changing input datasets. Hence, it is hard to build an optimum schedule for MapReduce job in advance and data skews must be overcome online.

Algorithms for load-balancing MapReduce have been proposed for the map phase [22,27] and the reduce phase [10,16,22]. In this work we concentrate on the reduce stage. An algorithm called fine partitioning has been proposed in [16]. It divides the space of the key values into kr partitions, where $k > 1$ is the algorithm control parameter. On completion of their computations mappers send to the master machine information about sizes of the partitions they created. The master assigns partitions to reducers on the basis of the Longest Processing Time (LPT, see e.g. [15]) algorithm. In another algorithm, called dynamic fragmentation, a mapper can initiate a split of a partition if its expected computational cost exceeds the average cost by some factor e. On completion of mapping, partition sizes are reported to the master. On the basis of reducing time estimations, the master decides which new partition to keep and which one to ignore. Let Π_i denote a partition on some mapper i. If partition Π_i has been split into Π_i', Π_i'' on mapper i, and has not been split on some other mapper j, then all the keys from Π_j must be copied both to the reducer processing Π_i', and to the reducer processing Π_i'', while the excessive keys from Π_j must be pruned. Due to this additional complication in dataset handling we do not consider dynamic fragmentation method in this study. In [22] a SkewTune method mitigating skew dynamically has been proposed. SkewTune manages data skew both in mapping and in reducing. We describe here only the reducing phase. Each time some reducer finishes computation, SkewTune redistributes the work from a *straggler* reducer, i.e. the one which has the latest expected completion time. Recipients of the work are free reducers and the reducers expected to finish before the partition being redistributed is completed. SkewTune uses range-partitioning of keys in a partition to allow only concatenation of outputs from mitigating machines without more complicated merging.

Load-balancing a MapReduce application has similar challenges to parallel sorting [33]. On the one hand, a typical MapReduce job comprises a sorting step. Both MapReduce and parallel sorting efficiency depends on accommodating to the distribution of keys. Hence, approximate histograms of the key distribution are constructed iteratively in the histogram-based parallel sorting algorithms [19,31]. On the other hand, the sort operation in MapReduce is just one part in the key-value processing workflow, and the other parts may reduce the impact of sorting on the overall performance. Quite often, the output from a MapReduce job does not have to obey the order between the keys, i.e., that processor j holds keys smaller than processor $j+1$, which is required in parallel sorting [19]. In many sorting algorithms tacit assumptions are made, e.g., that all keys are unique, the number of keys per processor (here $|\Omega|/r$) is large. Here, we depart from these assumptions and will consider the option of splitting a key cluster, albeit at additional cost of merging the reducer results.

Four alternative load-balancing algorithms have been proposed in [10]. The performance of the algorithms has been compared in four settings: easy or hard key partitioning and linear or loglinear (i.e. $O(n\log n)$ for n units) key sorting cost. In the experiments the keys had flat frequency distribution. In the easy partitioning the keys were distributed to reducers randomly, which resulted in rather flat partition sizes. In the hard partitioning the first partition had $1/r$th of the most frequent keys, the second partition comprised $1/r$th of the next less frequent keys, etc. The robustness of the algorithms has been tested against volatile platform and application performance parameters. Table 2 provides a subset of the results from [10] showing the number of tests in which certain load-balancing algorithm provided the shortest schedule. It can be seen in Table 2 that certain algorithms win in certain settings, but none dominates overall. Study [10] leaves at least two open questions: why certain algorithms prevail, and what is their performance under more demanding key distributions. We answer these questions in this paper.

4. Load-balancing algorithms

In this section algorithms applying different approaches to balancing partition skew will be presented. For practical reasons these algorithms must have low runtime. Thus, in the cases when solving a hard computational problem such as bin-packing is necessary, we have to recourse to heuristics.

4.1. Reference distribution algorithm

The reference distribution algorithm will be used to compare with the performance of the other load-balancing algorithms. The reference algorithm represents a default organization of a MapReduce job in optimistic performance conditions with orderly conducted communications. In this algorithm the partitioning function places key κ_i in partition $\Pi_{i \bmod R}$, one partition is created per reducer ($R = r$), and no load balancing is conducted. The master assigns partition Π_j to reducer $j+1$, for $j = 0, \ldots, r-1$. We impose a sequencing of reducers reading data from the mappers in order to obey the 1-port assumption, that is that no computer performs two communications at the same time. If $m \le r$, then mapper i communicates consecutively with reducers $i, i+1, \ldots, r, 1, \ldots, i-1$. If $m > r$, then reducer j communicates consecutively with mappers $j, j+1, \ldots, m, 1, \ldots, j-1$. Similar sequencing has been proposed in [31]. Each communication starts without unnecessary delay and no computer performs more than one communication at the same time. In the ideal case, when sizes of data sent from mapper i to reducer j are equal for all i, j, there are always $\min\{m, r\}$ communications taking place at a time. If the load distribution is unequal, then the number of concurrent communications may become smaller in some intervals, because some computers may have to wait until other machines finish communicating. Each reducer starts processing its partition as soon as it obtains data from all mappers.

For better exposition of the reference distribution algorithm, let us estimate the schedule length. Mappers run in time AV/m and communicate sizes and locations of the output files to the master in time $Cmr\varepsilon$ because the reads of the master are sequential. The master immediately assigns the r partitions and sends information on the mr file locations to r reducers in time $Cmr\varepsilon$. In the optimistic case partitions have equal size $\gamma V/r$, and each mapper produced r files of size $\gamma V/(mr)$, so that the shuffle time is $C \max\{1, \min\{m, r\}/l\}\gamma V/(mr)\max\{m, r\}$. Finally, reducers execute reduce tasks in time $T(\gamma V/r)$. Thus, the total schedule length is

$$T^* = AV/m + 2Cmr\varepsilon + \frac{C\gamma V}{mr} \max\{1, \min\{m, r\}/l\} \max\{m, r\} + T(\gamma V/r). \tag{2}$$

Eq. (2) is an optimistic estimation given for better exposition how the classic MapReduce works for a performance-predictable application. In practice, partitions may be unequal, the communications may be less regular, and the pattern of congestion may be more complicated. In such cases communication and computation times will be calculated according to the actual partition sizes.

4.2. Static algorithm

The static algorithm is based on fine partitioning [16] also known under the name of overdecomposition [24]. The main idea is to create more than r partitions and assign them to reducers in such a way that their loads are balanced. The key space partitioning is determined before the shuffle phase and cannot be improved during reducing.

The number of partitions is $R = kr$ for some integer $k > 1$. After processing the input splits, the mappers inform the master about their output file sizes sequentially, in time $Ckrm\varepsilon$. In order to obtain the optimum load assignment for the reducers, it is necessary to solve an NP-hard bin-packing problem. Therefore, the master uses a heuristic approach similar to LPT algorithm. It sorts the partitions according to the non-increasing sizes and assigns them one by one to the reducers with the smallest total load. This can be done in total time $t^{master} = a^{master}(krm + kr\log_2(kr) + kr\log_2 r)$, where a^{master} is the computing rate of the master. The information about the locations of the assigned files on the mapper disks is then sent sequentially to the reducers, in time $Ckrm\varepsilon$. The reducers read the data from the mappers, following the pattern of communication described in Section 4.1. Each reducer starts processing as soon as it gathers data from all mappers. Since sorting has greater than linear complexity, the many partitions assigned to a reducer are sorted and processed separately, to decrease sorting time.

4.3. Multi-dynamic algorithm

In contrast to the static approach, the dynamic method consists in improving load distribution during the reducing phase. The multi-dynamic algorithm performs multiple load balancing operations and is based on the idea similar to work stealing (see [11] and the references mentioned therein) and SkewTune [22]. Namely, reducers which become idle take over parts of load from the reducers with the greatest remaining computation time.

The number of partitions created by the mappers is $R = r$ and the job execution proceeds as in the reference algorithm (Section 4.1) until all reducers finish sorting and at least one reducer finishes computations. When a reducer becomes idle, it notifies the master, which then asks the other reducers about their expected processing time. The master chooses the most loaded reducer to share a part of its unprocessed load with the idle reducer. The amount of data to be sent to the

idle reducer is computed so that both computers finish processing at the same time. However, the key clusters cannot be continuously divided between different processors and hence, the amount of transferred load usually differs from the ideal value computed by the master. In order to meet the desired size, the overloaded reducer sends key clusters one by one, in the order in which they are stored on its disk (starting from the last one), as long as the total amount of transferred data does not exceed the limit computed by the master. In the worst case of a single indivisible cluster no load is transferred. This procedure is repeated every time some reducer becomes idle.

4.4. Mixed algorithm

The mixed algorithm is meant to combine advantages of the static and the dynamic approaches. Hence, it works both before shuffle, balancing reducer loads like the static algorithm, and during the reducing phase, when reducers which become idle obtain new partitions to process. We will call the two stages of the mixed algorithm the static part and the dynamic part, respectively.

The relation between the sizes of load distributed in the two phases is controlled by parameter X ($0 < X < 1$). A separate partitioning function is used to construct partitions for each part of the algorithm. A subset of arbitrary $\lfloor X \times |\Omega| \rfloor$ keys is chosen to be distributed in the static part of the algorithm. This subset is divided into $k_1 r$ partitions for some integer $k_1 \geq 1$, so that key κ_i is included in partition $\Pi_{i \bmod k_1 r}$. The remaining keys are divided into $k_2 r$ partitions for an integer $k_2 \geq 1$, to be distributed dynamically in the reducing phase. The partition containing key κ_i is now determined as $i \bmod k_2 r$. Thus, the total number of partitions is $R = (k_1 + k_2)r$. When mapping is finished, the $k_1 r$ partitions are distributed to the reducers in the same way as in the static algorithm (see Section 4.2). The remaining $k_2 r$ partitions are not yet assigned to any computers. The reducers read and process the data distributed statically. When a reducer becomes idle, it sends the master a request for more data. The master assigns it one of the remaining partitions to be distributed dynamically. The reducer reads the corresponding files from the mappers in the order described in Section 4.1 and processes them. This procedure is continued until processing all partitions.

4.5. Divide-keys algorithm

Divide-keys method is designed for arbitrary key distributions, including very skewed ones. Since the key distribution is in general unknown, the partitioning function is created at runtime. A similar problem is confronted in parallel sorting. To this end, key distribution is first determined by mappers reading the input data volume. On the basis of the key distributions obtained from mappers, the master constructs a partitioning function which is beneficial for equal reducer workload. Classically one key cluster is processed by one reducer. Here we depart from this assumption and allow to divide one key cluster between many reducers. The partitioning function is distributed to the mappers and the MapReduce proceeds in the standard way. If some key is partitioned, then it means that it will be processed by several reducers (computers). Let us call a *key* κ_i *leader* the reducer with the smallest index in the range of the reducers processing key κ_i. After processing the divided key(s) is finished, the reducers send the results of processing the key to the key leader, which merges the results into one file. For each α bytes of input data received by the reducers, they pass $\alpha\gamma_1$ bytes of output data to the key leader.

In more detail, divide-keys algorithm is defined as follows (cf. Fig. 1):

1. Mappers read splits as in the classic map stage to determine locally key frequencies $f_i, i = 1, \ldots, |\Omega|$, in time approx. AV/m.
2. Mappers report key frequencies to the master in time $|\Omega|C\varepsilon$ each, where ε is the size of master message. Since also the master obeys 1-port assumption, the communication is sequential.
3. The master merges frequencies reported by the mappers to determine key frequencies globally and sorts the frequencies non-increasingly. Let us call the ith key *frequent* if $rf_i > 1$, and *non-frequent* in the opposite case. Each frequent key cluster is split into $\lfloor rf_i \rfloor$ parts which require processing capability of 1 reducer each, and one part requiring processing capability of $rf_i - \lfloor rf_i \rfloor$ reducers. Thus, the key cluster is divided between $\lceil rf_i \rceil$ reducers. The non-frequent key clusters are never split, in order to avoid unnecessary overhead of merging final results. The key cluster parts are then assigned one by one to the reducers with the greatest remaining processing capability. Note that key parts are not necessarily assigned in the order of nonincreasing processing capability requirements, as all parts of a frequent key cluster are assigned to reducers consecutively, before the non-frequent key clusters. Thanks to this approach, if the data is not very highly skewed, then each reducer has at most one split key, which simplifies gathering final results by key leaders. Precisely, no reducer receives more than one divided key if the total frequency of all frequent keys does not exceed 1/2. In the opposite case a reducer may be assigned many split keys. The obtained partitioning function is encoded as an array giving for each key the reducer indices and fractions of their processing capabilities assigned to the key. Its length $F \geq |\Omega|$ depends on the key frequencies, but does not exceed $|\Omega| + r$. Computing the partitioning function takes time $a^{master}(|\Omega| \log_2 |\Omega| + r + F(\log_2 r + 1))$ because keys need sorting, and they are assigned to reducers using a priority queue.
4. The master sends the partitioning function to the mappers in time $mCF\varepsilon$.
5. As soon as some mapper receives the partitioning function, it starts mapping. If key κ_i cluster is divided between several reducers, each key-value pair containing key κ_i is assigned randomly to one of them with probability proportional to the assigned processing capability.

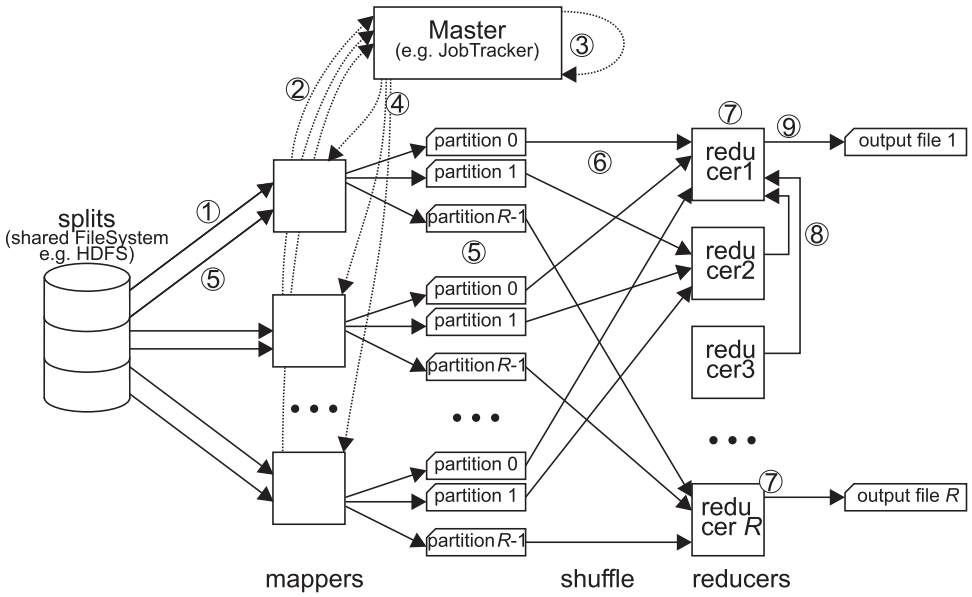

Fig. 1. Scheme of divide-keys operation. Mapper and reducer initiation messages have been omitted.

6. Shuffle stage proceeds as described in Section 4.1.
7. Reducing is performed in the standard way.
8. If some key κ_i has been divided between several reducers, then its leader reducer reads output files from the other reducers processing this key to merge the results. Key κ_i leader pulls the files from the other reducers as soon as possible, even if they are still processing some other keys.
9. Key leaders merge the files for each divided key. Merging p files of total size x takes time $a^{red}x\log_2 p$. Thus, merging the results for key κ_i that was processed by $p>1$ reducers takes time $a^{red}\sigma_i\gamma_1\log_2 p$.

5. Events simulator

In this study we will analyze the influence of diverse system parameters and data skew degree on the efficacy of load-balancing algorithms. Due to the extent of the study and breadth of analyzed parameter ranges it is not possible to measure performance directly. Hence, it is necessary to recourse to analytic and simulation approaches. We need detailed communication timings in the MapReduce job schedule to compare the load-balancing algorithms. Coarse representation at the datacenter level (see e.g. [30]) is insufficient in our case. Although MapReduce simulators already exist [5,17,21], they do not conform with our MapReduce model and skew mitigating algorithms. Therefore, we created a program in C++ that simulates the execution of MapReduce with load balancing methods described in Section 4. The simulator is publicly available at https://github.com/mr-skew/mapreduce-balancing-simulator.

Our program computes a schedule of MapReduce execution for each proposed load balancing algorithm. Although simulating computation stages, like reducing a given amount of data, is generally simple, computing communication schedules requires more effort. In order to simulate the shuffle phase, we track concurrent communications and maintain a queue of the remaining required communications for each reducer if $m \geq r$, or for each mapper if $m < r$. The simulator processes a queue of communication events, such as the start or the end of some communication. The first event is shuffle start, when data transfers begin. In each step the simulator finds the next event to occur. Then, the amounts of data remaining to be sent in all communications are updated and a new bandwidth perceived by the communicating processes is computed, because it depends on the number of simultaneously open channels. When a communication finishes, new events may be appended to the queue, since the processors may start new data transfers. As we observe one-port model, it may happen that a queued data transfer is suspended until both processors taking part in it finish their previous communications. The simulator processes events until all planned transfers complete.

The dynamic phase of the mixed algorithm, load balancing in the multi-dynamic algorithm and results merging in divide-keys algorithm are simulated in a similar way. A simulator extension for these algorithms consists in including additional event types, as completing computations or finishing processing of a key cluster by a reducer may also initiate a communication.

Assigning keys to partitions in the static algorithm and the static part of the mixed algorithm is done by the simulator in the same way as the master would do this in reality. A custom partitioning function for divide-keys algorithm is also computed exactly as described in Section 4.5. The number of steps (i.e. the length F of the sequence encoding the partitioning

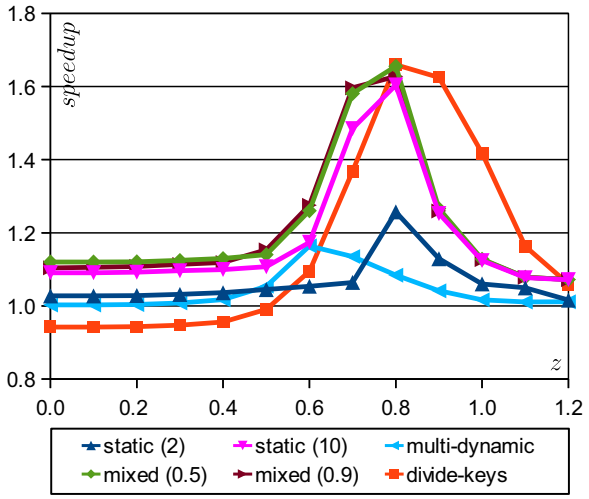

Fig. 2. Speedup vs. z.

function) is recorded in order to compute the exact time the master needs for computation and communication with the mappers.

6. Evaluation of the algorithms

In this section we compare the effectiveness of the load-balancing algorithms against changing system parameters. Let us first describe our method of generating test instances. The number of different keys in the input data is always $|\Omega| =$ 1E6. To obtain key frequencies for a given parameter z controlling input data skew we choose 1E8 keys randomly from the corresponding Zipfian distribution. Unless stated otherwise, we assume the following reference system and application parameter values: $V =$ 1E12 B, $\varepsilon = 8$ B, $m = 1000$, $r = l = 100$, $\gamma = 0.1$, $\gamma_1 = 1$, $C =$ 1E-8 s/B, $A = a^{master} = a^{sort} =$ 1E-7 s/B, $a^{red} =$ 1E-6 s/B. Value $C =$ 1E-8 s/B corresponds with the bitrate of Gigabit Ethernet. At the current I/O speeds, $A =$ 1E-7 s/B, $a^{red} =$ 1E-6 s/B represent a compute-intensive application. In general the range of possible parameter values is very broad, depending on the application, hardware and software platform, and background load. By analyzing wide ranges of these parameters we avoid focusing on the features of one particular application and platform configuration. Let us note that $\gamma_1 = 1$ means that the data size does not decrease during reducing and hence, the amounts of reducer results to merge in divide-keys algorithm can be quite big. Such a choice of γ_1 makes our tests more difficult for divide-keys algorithm. The reducing rate a^{red} is greater than the remaining computation rates in order to represent MapReduce jobs with complex Reduce functions. For each tested setting 10 workload instances were generated. The measure of the algorithm quality is the average speedup in comparison to the reference distribution algorithm (Section 4.1).

The performance of the static algorithm depends on the value of parameter k. The efficiency of the mixed algorithm is ruled by the values of X, k_1, k_2. The computational experiments conducted to tune these parameters are described in [10]. We decided to use $k = 2$ and $k = 10$, and the corresponding variants of the static algorithm will be denoted static(2) and static(10). For the mixed algorithm we also selected two settings: $X = 0.5$, $k_1 = k_2 = 10$ and $X = 0.9$, $k_1 = k_2 = 10$. These algorithm variants will be called mixed(0.5) and mixed(0.9), correspondingly.

6.1. Data skew

Let us start with comparing the performance of our algorithms for the reference system configuration and different values of parameter z controlling data skew (see Fig. 2). The range of z can be divided into three intervals. When $z \leq 0.4$, the data skew is very small and balancing the reducer loads is not really necessary. Therefore, the multi-dynamic algorithm has speedup close to 1. However, as the reducers sort data in nonlinear time, it is beneficial to divide them into a larger number of smaller partitions. Hence, algorithms mixed(0.5) and mixed(0.9), which create $20r$ instead of just r partitions, perform best. The static(10) algorithm, which creates $10r$ partitions, achieves speedup close to the mixed algorithms, but static(2) is significantly worse, which shows that $2r$ partitions are not enough. Algorithm divide-keys has speedup smaller than 1 because it introduces a big overhead for computing the partitioning function.

When $0.5 \leq z \leq 0.7$, the data skew is moderate, there are more opportunities to improve the load distribution, and hence, the speedups of all algorithms increase. The best results are still obtained by the mixed algorithms. The multi-dynamic algorithm outperforms static(2), which creates too few partitions to balance the load well. Let us note that for $z \leq 0.7$ the greatest expected key frequency is still smaller than $1/r = 0.01$, so that algorithm divide-keys does not split any key clusters between reducers. However, while creating a partitioning function it balances the load similarly to the static algorithm, but

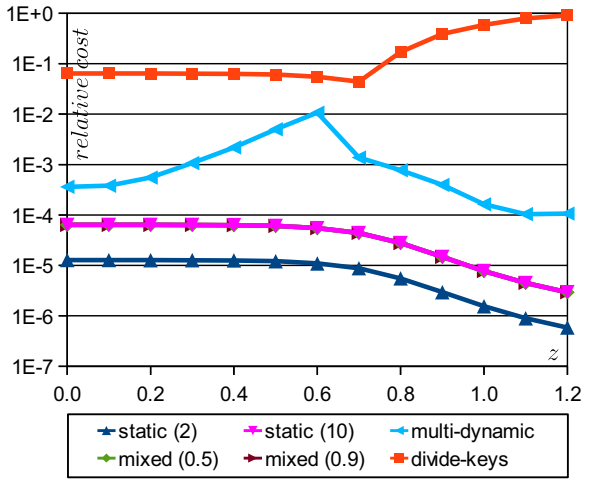

Fig. 3. Relative load balancing cost vs. z.

operating on single keys instead of larger partitions. Thus, it balances the load so well that the profit is greater than the cost of creating a custom partitioning function. Hence, the speedup of divide-keys algorithm grows fast with increasing z.

When data skew becomes big, i.e. $z \geq 0.8$, the greatest expected key frequency exceeds $1/r$ and still grows with z. The static, mixed and multi-dynamic algorithms cannot divide key clusters, and the time of processing the most frequent key dominates the schedules they generate. Hence, the speedups of these algorithms decrease. Algorithm divide-keys splits processing the frequent keys between several reducers and performs best for $0.8 \leq z \leq 1.1$. Still, the costs of merging the results obtained for a single key on several reducers are quite high and grow with z. Indeed, for $z = 0.8$ the most frequent key cluster is expected to be divided between 2, for $z = 1$ between 7, and for $z = 1.2$ between 9 reducers, and the number of parts to be merged affects the merging time. Thus, the speedup of algorithm divide-keys also decreases. For $z = 1.2$ the costs of merging final results are so high that divide-keys is again outperformed by both mixed algorithms. Let us remind that the time of merging final results in divide-keys algorithm depends on the value of γ_1. In the presented experiment $\gamma_1 = 1$, which is high. For the applications with data size decreasing during reducing ($\gamma_1 < 1$) the performance of divide-keys will be analyzed in the further text (see Fig. 11).

It is an important question, whether the costs of load balancing can offset the benefits. In the static algorithm, the cost of load balancing is the time spent sending partition sizes to the master, assigning the partitions to reducers using the LPT-like algorithm, and informing the reducers about files to be read. Similarly, the load balancing cost of the mixed algorithm is the time of master communications and computations in the static phase. In the multi-dynamic algorithm, the load balancing cost is the total time of redistributing the data between the reducers. The divide-keys algorithm has two components of the cost. The first is the time spent on computing key frequencies by the mappers, transferring them to the master, constructing a partitioning function and sending this function back to the mappers. The second component is the time of collecting and merging the results obtained for the split keys.

The average relative load balancing costs for changing z are shown in Fig. 3. A relative cost of the load balancing is the ratio of the cost defined above and the length of the schedule built by the reference distribution algorithm. The costs of the static and the mixed algorithms depend on the numbers of partitions kr, $k_1 r$, respectively. Since $k_1 = 10$ in both mixed(0.5) and mixed(0.9), these algorithms have the same load balancing costs as static(10) algorithm (the corresponding lines overlap in Fig. 3). The costs of static(2) are approximately 5 times smaller, as the number of partitions is only $2r$. It can be seen in Fig. 3 that the relative costs of the static and the mixed algorithms decrease with z. This is caused by the increasing length of the reference algorithm schedule. In the absolute terms, load balancing costs of the static and mixed algorithms do not depend on z. In the multi-dynamic algorithm the costs are higher than for the mixed algorithm, because large amounts of data are transferred between the reducers. For $z \leq 0.6$ these costs grow with increasing z, as more data need to be shifted when the data skew becomes bigger. However, the multi-dynamic algorithm starts working only after all reducers finish sorting. If the data skew is big, the most loaded reducer finishes sorting late, and the multi-dynamic algorithm has fewer opportunities to transfer data. Therefore, the costs decrease for $z \geq 0.6$. Since the costs in the multi-dynamic algorithm are directly connected with balancing reducer loads, the best speedup for this algorithm coincides with the greatest costs (cf. Figs 2, 3). The divide-keys algorithm is characterized by the largest load balancing costs. When $z \leq 0.7$, no keys are split between reducers, and only the first cost component, connected with computing a custom partitioning function, is borne. In the absolute terms, this cost does not depend on z, but since the reference schedule gets longer, the relative cost slightly decreases with z. For $z \geq 0.8$, some keys are split, and the results need to be merged. Then, the costs grow fast with growing z. For $z = 1.2$ they are almost as big as the length of the schedule obtained by the reference distribution algorithm.

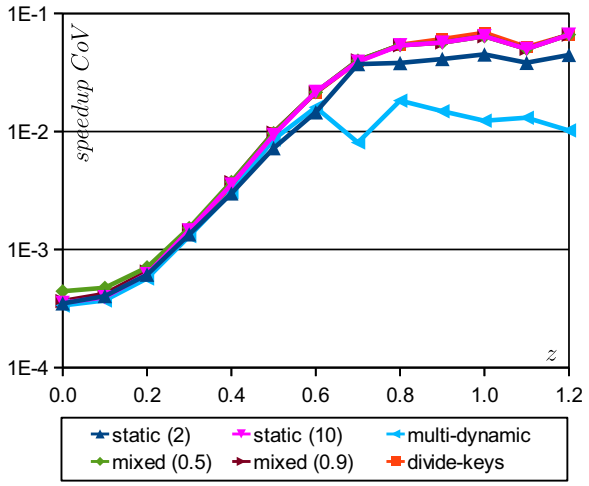

Fig. 4. Speedup coefficient of variation vs. z.

It can be concluded that the static and mixed algorithms have very low load balancing costs. The costs of the multi-dynamic algorithm are medium, and for the divide-keys algorithm the costs are the biggest. Only in the divide-keys algorithm, for small or very big skews, can the costs of load-balancing offset the benefits (see Figs 2, 3). Hence, the divide-keys algorithm should be used only for heavily skewed key distributions. For other algorithms, their limitations in dealing with big data skew and the advantages of creating smaller partitions are more influential than the costs of load balancing.

In the next experiment we assessed the stability of results delivered by our skew mitigating algorithms. For each set of 10 test instances generated for a given experimental setting, we computed the coefficients of variation (CoV) of the speedups obtained by different algorithms. The average coefficients over 45 settings (described in Sections 6.1 – 6.4) are presented in Fig. 4. It can be seen that parameter z strongly affects the coefficients of variation of the speedups. The higher data skew is, the more diversified the results returned by all algorithms. For small z the stability of all algorithms is comparable. When z becomes big, the coefficients of variation of both mixed algorithms, divide-keys and static(10) remain very similar (the corresponding lines overlap in Fig. 4). Algorithm static(2) has a significantly smaller coefficient of variation, and the multi-dynamic algorithm delivers the most stable results. Thus, the algorithms constructing shorter schedules are characterized by bigger differences between the speedups obtained for various instances. Still, the performance of all algorithms may be considered stable, since all the average coefficients of variation are smaller than 0.1.

In the following text we will study the impact of the system parameters on the performance of the skew mitigating algorithms. As we have already shown that parameter z is crucial for the performance, in each experiment we will analyze different values of z and some other system parameter. In the following figures z is always shown along the horizontal axis. For each pair of parameter values we will present the dominating algorithm, i.e. the one which achieved the best average speedup.

6.2. Sizes of input and intermediate data

We will now study the influence of parameter γ, which controls the relationship between the sizes of input and intermediate data. The higher γ is, the larger the contribution of sorting and reducing in the application execution time. In Fig. 5 we can again clearly see the areas with small ($z \leq 0.4$), moderate ($0.5 \leq z \leq 0.7$) and big ($z \geq 0.8$) data skew. When the skew is small or moderate, γ does not affect the dominating algorithm. For small skew the best algorithm is mixed(0.5), which creates $20r$ partitions of similar sizes and hence, is best at shortening the sorting time. When the skew is moderate, algorithm mixed(0.9) becomes better, since working on smaller partitions in the second stage of the mixed algorithm allows better load balancing. The influence of γ is visible for high data skew. When both γ and z are big, the time necessary for processing the largest key cluster on a single reducer is very long. Thus, it is best to avoid it by using algorithm divide-keys. When γ gets smaller, sorting and reducing data becomes less important in comparison to mapping. The profit from dividing a key cluster between reducers is smaller, whereas the costs of reading input data twice in order to create a custom partitioning function remain high. Therefore, divide-keys performs worse than the mixed algorithms for small γ and big z. Since none of the mixed algorithms can cope with the load imbalance caused by a single frequent key, the relationship between mixed(0.5) and mixed(0.9), in this area of γ and z, is determined by the decrease of sorting time. Although for high skew a larger number of keys in a partition does not necessarily mean a larger partition, algorithm mixed(0.5) is more probable to create $20r$ balanced partitions than mixed(0.9), constructing, e.g., $10r$ big and $10r$ small partitions. Thus, sorting takes shorter time in mixed(0.5) and this algorithm wins for big data skew. It is worth noting that the lower part of the area where divide-keys performs best is triangle-shaped. This is caused by two facts. On the one hand, for $z = 0.8$ and big

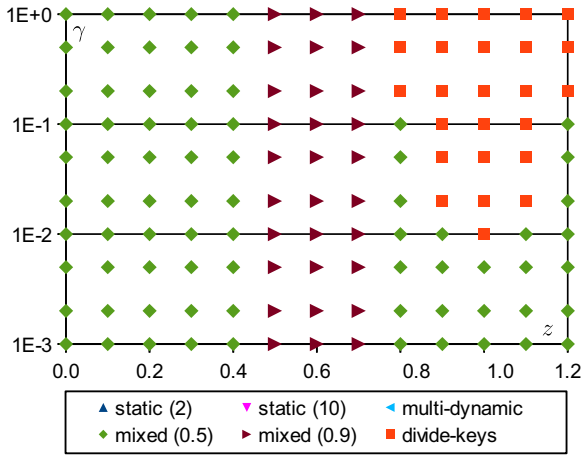

Fig. 5. Dominating algorithm vs. z and γ.

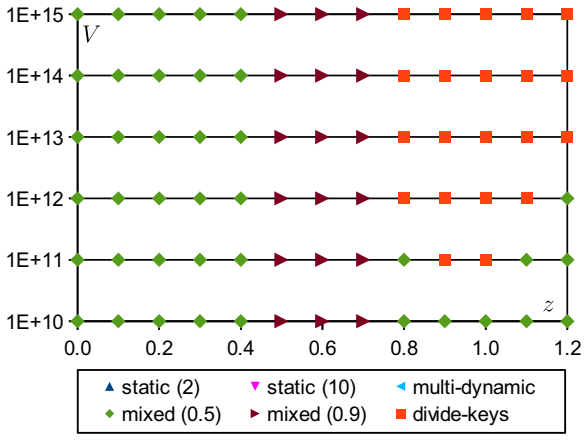

Fig. 6. Dominating algorithm vs. z and V.

γ the difference between the divide-keys and mixed algorithms is not as big as for $z = 1$, because data skew is not so high. On the other hand, when $z = 1.2$ and γ is big, the cost of merging final results by divide-keys is very large, and again the difference between divide-keys and mixed algorithms gets smaller. Thus, $z = 0.8$ and $z = 1.2$ require smaller change of γ to make mixed algorithms better than divide-keys, than in the case of $z = 1$. Let us note that the influence of m on the dominating algorithms (not shown here) is very similar to the results for changing γ, since smaller m makes mapping longer and more important in comparison to sorting and reducing.

In Fig. 6 it can be seen that the effect of changing V is very similar to changing γ. However, the reasons for algorithm mixed(0.5) outperforming divide-keys for big z and small V are different than in Fig. 5. Let us remind that in our experiments the number $|\Omega|$ of different keys is fixed. Thus, decreasing V means that the number of bytes per key decreases. The time of mapping and reducing depends on V, but the time of constructing a partitioning function by divide-keys depends on $|\Omega|$. Gathering key frequencies from all mappers sequentially needs time $m|\Omega|C\varepsilon$, which is very large in comparison to the other MapReduce stages for small V. Hence, when $V = 1E10$, algorithm divide-keys performs poorly and it is better to use mixed(0.5).

6.3. Reducer parameters

The results of the analysis on the changing number of reducers r are presented in Fig. 7. The areas in which certain algorithms dominate in Fig. 7 are slanted rather than vertical, but similar to the results of previously described experiments. They show once more that algorithms mixed(0.5), mixed(0.9), divide-keys and again mixed(0.5) are best to use for small, moderate, high and very high data skew, correspondingly. In order to explain the slope of the borders between the algorithm dominance areas, let us observe that the crucial parameter determining partition skew in MapReduce is not the highest key frequency $\max\{f_i\}$, but the ratio $\max\{f_i\}/r$. For example, if $\max\{f_i\} = 0.015$ and $r = 100$, then the size of the corresponding key cluster exceeds the desirable reducer load, and the cluster will be split by divide-keys algorithm. But for $\max\{f_i\} =$

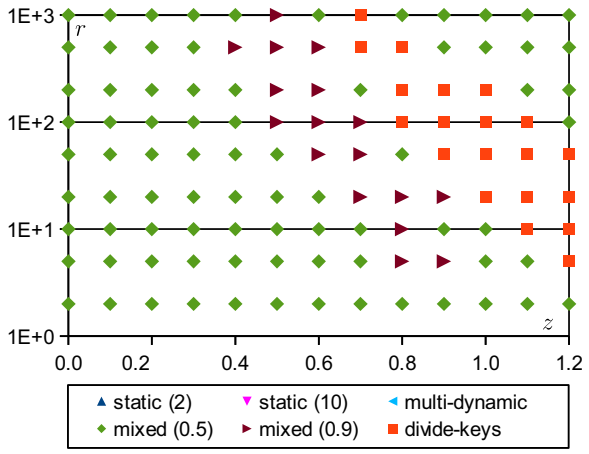

Fig. 7. Dominating algorithm vs. z and r.

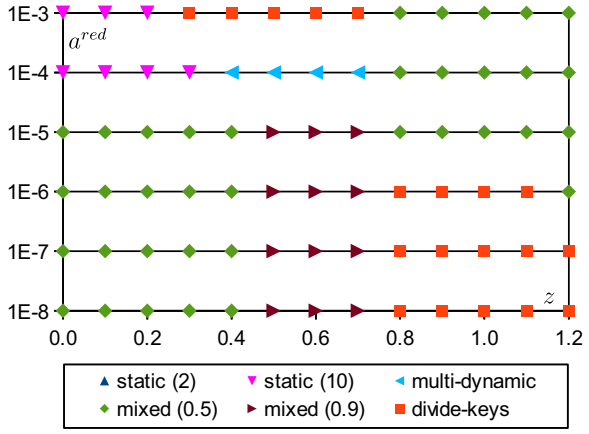

Fig. 8. Dominating algorithm vs. z and a^{red}.

0.015, $r = 10$, such a key cluster is much smaller than the average load per reducer and the cluster will not be split. Then, the divide-keys method will not be able to exploit its competitive features. Thus, with decreasing r the scaling parameter of skew z has to increase to make divide-keys algorithm start dividing the key clusters. On the other end of the winning area, divide-keys algorithm is defeated due to the results merging cost. The more reducers we have and the bigger the skew parameter z is, the earlier the merging costs outweigh the gains from splitting a key cluster.

Fig. 8 presents the dominating algorithms for different values of reducing rate a^{red}. Large a^{red} makes reducing more important in comparison to mapping, shuffling and sorting. It can be seen that this parameter influences the performance of the algorithms for all values of z. When both z and a^{red} are small, the best algorithm is mixed(0.5), as the partitions it creates result in the fastest sorting. However, when a^{red} grows, balancing the load in the reducer stage becomes more important. Hence, it is better to use algorithm static(10), which not only shortens sorting, but also balances the reducer loads better. For moderate skew, algorithm mixed (0.9) is the best when a^{red} is not very big. Still, when $a^{red} = 1\text{E-}4$ s/B, sorting time is less important than balancing reducer loads. The static and mixed algorithms, which work on partitions containing many keys, are outperformed by the multi-dynamic algorithm, which can move single key clusters between reducers after sorting is completed. When a^{red} is even bigger, the profit from assigning single key clusters to reducers before shuffle phase is so big that algorithm divide-keys wins, although it introduces a big overhead in the map phase. However, for big data skew and big a^{red} it is not profitable to use divide-keys anymore. Indeed, let us remind that parameter a^{red} controls not only reducing time, but also merging the final results by divide-keys algorithm. Thus, when a^{red} is big and z is large enough for merging to occur, divide-keys is worse than mixed(0.5).

6.4. Communication parameters

The influence of communication rate C on the algorithm dominance relationships is shown in Fig. 9. When z is small and C is very big ($C = 1\text{E-}4$ s/B), it is more important not to introduce too many additional communications than to balance

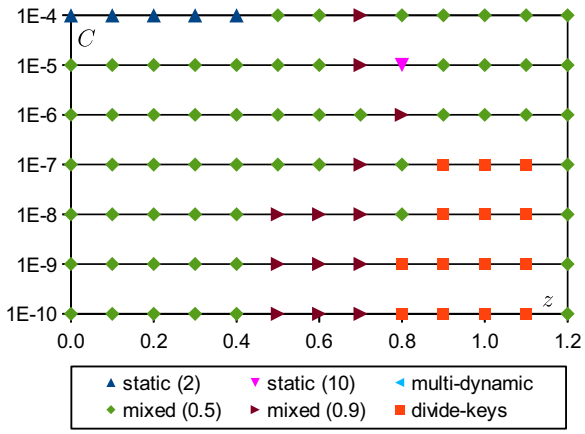

Fig. 9. Dominating algorithm vs. z and C.

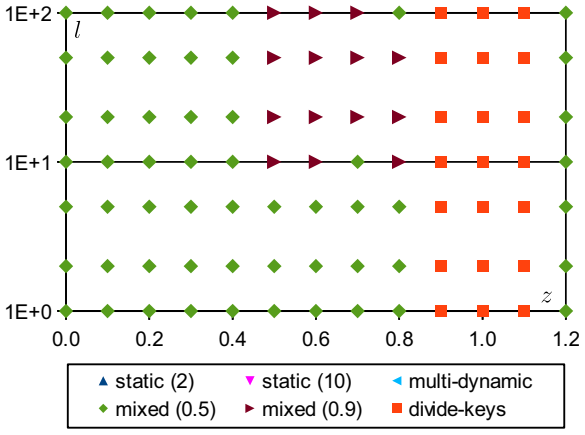

Fig. 10. Dominating algorithm vs. z and l.

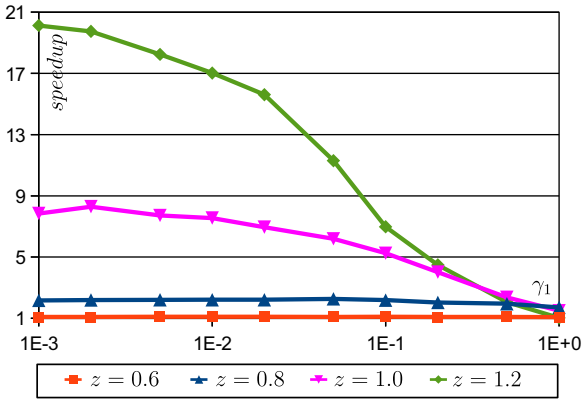

Fig. 11. Speedup of divide-keys algorithm vs. z and γ_1.

the load carefully. Therefore, algorithm static(2) achieves the best results. For moderate skew and large C both algorithms mixed(0.5) and mixed(0.9) win in some cases, and for $z = 0.8, C =$ 1E-5 static(10) is the best. The three algorithms obtain similar results at $z = 0.8$, $C =$ 1E-5. When z and C are big, slow communication damages the performance of gathering key frequencies and merging final results in divide-keys algorithm. Under such conditions it is better to use mixed(0.5) than divide-keys algorithm.

Changes of the bisection width l (see Fig. 10) modify the earlier observed dominance relationships only when more than l pairs of processors are communicating at the same time. In particular, l does not influence sequential communications

with the master. Note that in the static phase of the mixed algorithm there are many parallel communications, while in the dynamic phase the number of concurrent communications is usually small. Therefore, for moderate skew and small l algorithm mixed(0.5) becomes better than mixed(0.9), because it has shorter static and longer dynamic phase.

6.5. Size of final results

Let us now present the influence of parameter γ_1, determining size of the final results, on the performance of divide-keys algorithm (cf. Fig. 11). As we already noted, in some applications γ_1 may be smaller than 1. Its impact on the performance of divide-keys depends on z. When z does not exceed 0.7, there are no final results to merge and γ_1 has no impact at all. When $z \geq 0.8$, smaller γ_1 means faster merging and its influence grows with growing z. In particular, the speedup obtained for $z = 1.2$ and $\gamma_1 = 1\text{E-}3$ is greater than 20. Thus, for strong data skew and small γ_1 algorithm divide-keys will obtain much better results than in our earlier experiments with $\gamma_1 = 1$, and it will more often outperform the other algorithms.

7. Summary and conclusions

In this work we analyzed the problem of mitigating the data skew in MapReduce computations. We proposed four methods of different genre for handling the skew. The static algorithm adjusts the load distribution before the shuffle phase, using the idea of fine partitioning (a.k.a. overdecomposition). The multi-dynamic algorithm performs many simple load balancing operations in the reducing phase. The mixed algorithm processes a part of data like the static algorithm, and sends the chunks containing the remaining data on requests from idle reducers. Finally, the divide-keys algorithm computes a partitioning function adjusted to the input data distribution and allows processing a key cluster on several reducers.

The algorithms were tested in a series of computational experiments. It was shown that the performance of the algorithms depends on parameter z controlling data skew, as well as the system parameters. Our results on the algorithm dominance conditions can be summarized as follows.

1. If data skew is small ($z \leq 0.4$), the best choice is in most cases algorithm mixed(0.5).
2. When the skew is moderate ($0.5 \leq z \leq 0.7$), algorithm mixed(0.9) usually obtains the best results.
3. When the skew is big, the winning algorithm depends more on the system parameters.
   (a) If a^{red}, γ_1, C are small, and γ, m, V are big, then algorithm divide-keys is the best.
   (b) In the opposite case it is better to use mixed(0.5).

Thus, divide-keys should be used if data skew is big, and the costs of computing a partitioning function and merging final results are not very large compared to the other components of MapReduce. If this is not the case, the mixed algorithm, possibly with some tuning of parameter X, is a good choice. It can be also recommended as a safe strategy to begin with in the lack of the initial knowledge on the workload. We also found some settings for which even static(2), static(10) or multi-dynamic algorithms perform best. Overall, it can be concluded that no single "best" load balancing algorithm for MapReduce can exist. Depending on the type of workload or system features, some types of algorithms may be better than the others. Choosing a good load balancing method requires performance analysis of the MapReduce application to get the knowledge on the system, application and dataset parameters. Thus, hybrid approaches seem the inevitable solution.

Future work on this subject may include analyzing different types of skew, like unequal key distribution between mappers, or taking into account machine failures. Another direction is to adjust the algorithms to obtain a given, not necessarily equal, load distribution between the reducers. This may be useful in optimizing the overlap of communications and computations in chains of MapReduce applications. It is also a valid question, whether the analysis and algorithms proposed here can be ported to different platforms like Apache Spark with MapReduce-type applications. In this study we assumed that the skew in key frequencies is the main source of load imbalance. Load imbalance may also arise in other ways, e.g., the keys may not be equally easy to process. Our algorithms will still be able to work in such cases, because the number of bytes in a key cluster is a proxy to processing time function. Availability of additional information on the workload, e.g., the average processing times for particular keys, can be used to fine-tune the algorithms. Again, this leads to the earlier conclusion that knowing the workload is essential. Yet, it would also induce additional layers of complexity in load balancing. Hence, the performance of the algorithms should be compared in a practical environment such as Hadoop or Spark.

References

[1] R. Agrawal, H.V. Jagadish, Partitioning techniques for large-grained parallelism, IEEE Trans. Comput. 37 (1988) 1627–1634.
[2] Apache Software Foundation, Apache CouchDB, 2015. http://couchdb.apache.org/.
[3] Apache Software Foundation, Apache MRQL, 2015. https://mrql.incubator.apache.org/.
[4] Apache Software Foundation, Giraph - Welcome To Apache Giraph!, 2015. http://giraph.apache.org/.
[5] Apache Software Foundation, Mumak: Map-Reduce Simulator, 2015. https://issues.apache.org/jira/browse/MAPREDUCE-728.
[6] Apache Software Foundation, Welcome to Apache Hadoop, 2014. http://hadoop.apache.org/.
[7] J. Berlińska, M. Drozdowski, Dominance properties for divisible mapreduce computations, research report RA-09/09, institute of computing science, poznań university of technology, 2009. http://www.cs.put.poznan.pl/mdrozdowski/rapIIn/ra0909.pdf.
[8] J. Berlińska, M. Drozdowski, Scheduling divisible mapreduce computations, J. Parallel. Distrib. Comput. 71 (2011) 450–459.
[9] J. Berlińska, M. Drozdowski, Scheduling multilayer divisible computations, RAIRO Oper. Res. 49 (2015) 339–368.
[10] J. Berlińska, M. Drozdowski, Algorithms to mitigate partition skew in mapreduce applications, research report RA-01/15, institute of computing science, Poznań University of Technology, 2015. http://www.cs.put.poznan.pl/mdrozdowski/rapIIn/RA-01-2015.pdf.

[11] R.D. Blumofe, C. E.Leiserson, Scheduling multithreaded computations by work stealing, J. ACM 46 (1999) 720–748.
[12] S. Chen, S. W.Schlosser, Map-reduce meets wider varieties of applications, intel research report IRP-TR-08-05,. http://www.cs.cmu.edu/~chensm/papers/IRP-TR-08-05.pdf.
[13] Y.-C. Cheng, T. G.Robertazzi, Distributed computation with communication delay, IEEE Trans. Aerosp. Electron. Syst. 24 (1988) 700–712.
[14] J. Dean, S. Ghemawat, Mapreduce: Simplified Data Processing on Large Clusters, in: OSDI'04: Sixth Symposium on Operating System Design and Implementation, CA, San Francisco, 2004, pp. 137–150.
[15] M. Drozdowski, Scheduling for Parallel Processing, Springer, 2009.
[16] B. Gufler, N. Augsten, A. Reiser, A. Kemper, Handling data skew in mapreduce, in: CLOSER '11: Proceedings of the 1st International Conference on Cloud Computing and Services Science, 2011, pp. 574–583.
[17] S. Hammoud, M. Li, Y. Liu, N. Alham, Z. Liu, MRSim: A discrete event based mapreduce simulator, in: Seventh International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), Vol. 6, 2010.
[18] HBase and MapReduce, Apache Software Foundation, 2014. https://hbase.apache.org/book/mapreduce.html.
[19] L. V.Kalé, S. Krishnan, A comparison based parallel sorting algorithm, in: Proceedings of the 22nd International Conference on Parallel Processing, 1993, pp. 196–200.
[20] S. Kavulya, J. Tan, R. Gandhi, P. Narasimhan, An analysis of traces from a production mapreduce cluster, technical report CMU-PDL-09-107, Carnegie Mellon University, Parallel Data Laboratory, 2009.
[21] W. Kolberg, P.d. Marcos, J.C.S. Anjos, A.K.S. Miyazaki, C.R. Geyer, L.B. Arantes, MRSG - a mapreduce simulator over simgrid, Parallel Comput. 39 (2013) 233–244.
[22] Y. Kwon, M. Balazinska, B. Howe, J. Rolia, Skewtune: Mitigating skew in mapreduce applications, in: SIGMOD '12: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 2012, pp. 25–36.
[23] K.H. Lee, Y.J. Lee, H. Choi, Y. D.Chung, B. Moon, Parallel data processing with mapreduce: a survey, ACM SIGMOD Record 40 (4) (2011) 11–20.
[24] J. Lifflander, S. Krishnamoorthy, L. V.Kale, Work stealing and persistence-based load balancers for iterative overdecomposed applications, in: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC'12), 2012, pp. 137–148.
[25] J. Lin, The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce, in: Proceedings of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval, 2009.
[26] J. Lin, C. Dyer, Data-Intensive Text Processing with MapReduce, Morgan & Claypool, 2010.
[27] M. Lin, L. Zhang, A. Wierman, J. Tan, Joint optimization of overlapping phases in mapreduce, Perform. Eval. 70 (2013) 720–735.
[28] C. Loboz, Cloud resource usage - heavy tailed distributions invalidating traditional capacity planning models, J. Grid Comput. 10 (2012) 85–108.
[29] MongoDB Inc., Mongo DB Manual 2.4, 2014. http://docs.mongodb.org/manual/core/map-reduce/.
[30] Z. Niu, B. He, F. Liu, Not all joules are equal: towards energy-efficient and green-aware data processing frameworks, in: Proceedings of IEEE International Conference on Cloud Engineering (IC2E), 2016, pp. 2–11.
[31] E. Solomonik, L. V.Kale, Highly scalable parallel sorting, in: IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010, pp. 1–12.
[32] S.J. Plimpton, K.D. Devine, Mapreduce in MPI for large-scale graph algorithms, Parallel Comput. 37 (2011) 610–632.
[33] Sort Benchmark Home Page, 2016. http://sortbenchmark.org/.
[34] T. White, Hadoop: The definitive guide, o'reilly media, 2012,.
[35] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, I. Stoica, Improving MapReduce Performance in Heterogeneous Environments, OSDI'08: Sixth Symposium on Operating System Design and Implementation, 2008, pp. 29–42.