Poznań University of Technology

Algorithms to Mitigate Partition Skew in MapReduce Applications

J.Berlińska, M.Drozdowski

Research Report RA-01/15

2015

Institute of Computing Science, Piotrowo 2, 60-965 Poznań, Poland

Algorithms to Mitigate Partition Skew in MapReduce Applications

J.Berlińska

Faculty of Mathematics and Computer Science,
Adam MickiewiczUniversity,
Poznań, Poland

M.Drozdowski

Institute of Computing Science,
Poznań University of Technology,
Poznań, Poland

Abstract

Algorithms for mitigating imbalance of the MapReduce computations are considered in this paper. MapReduce is a new paradigm of processing big datasets in parallel. A MapReduce job consists of two phases of mapping and reducing. In the latter phase computation completion times may become imbalanced due to unequal distribution of the data. We propose four algorithms for balancing computational effort in the reducing phase. The static algorithm improves the load distribution by constructing a few times more load partitions than the number of reducing computers. The multi-dynamic algorithm performs many load balancing operations during the reducing phase, whereas the single-dynamic algorithm uses simulation to balance the load in a single step in the reducing phase. The mixed algorithm works both before and during the reducing phase. The performance of the algorithms is evaluated and compared by simulation. We conclude that the distribution of keys in the input data and the complexity of sorting by the reducers determine which algorithm is best to use.

Keywords: MapReduce, Load balancing, Divisible loads, Scheduling, Performance evaluation

1 Introduction

MapReduce is a paradigm for processing big volumes of data in parallel [10, 21, 24]. A MapReduce job consists of two processing steps: mapping and reducing. We will call the computers executing the first phase *mappers*, and the second phase *reducers*. Let us assume that there are m mappers and r reducers. A MapReduce job is supervised by a master machine. The whole volume of initial input data is divided into many equal-size blocks called *splits* processed in parallel in the map phase. Mappers read the splits 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 key-value pairs with equal key_1 a *cluster*. On each mapper the pairs are scattered to local output files so that $(key_1, value_1)$ pairs with equal key_1 land in the same file. Let us assume that the output from the mappers is divided into R *partitions*. In general the number of partitions R is unrelated to the number of reducers r. The target output file is usually identified using a function of the form $\texttt{hash}(key_1) \bmod R$. Thus, basically Map function *partitions* the space of key_1 into R subspaces. At the end of mapping phase each mapper holds R local output files. When all mappers finish computation, then the MapReduce job progresses to the next phase. In a communication stage called *shuffle* the reducers pull the files from the mappers. All the key-value pairs copied to one reducer are grouped together by key_1 using, e.g., external sort and are processed to produce a set of new $(key_2, value_2)$ pairs. A cluster of some key_1 is processed by one reducer. 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. As an example of a MapReduce application consider analysis of the geographic trends in queries submitted to a search engine. Mappers read the submitted phrases *phrase* and IP addresses of the computers from the logs of the web-servers, translate the IP addresses into country locations *country* and emit a set of $(country, phrase)$ key-value pairs. The pairs are partitioned and routed to reducers on the basis of *country* in a $(country, phrase)$ key-value pair. Each reducer sorts its set of key-value pairs by *country* and produces a new key-value pair $(country, stats)$ where *stats* is an object comprising statistics on a given *country* including, e.g., the total number of queries, a histogram for a few most popular phrases, etc. If *stats* comprises a list of phrases submitted from a given *country* then it can be a subject of another MapReduce to compile a detailed phrase analysis for each country. Similar applications of MapReduce can be found in text, log, measurement, image processing, machine learning, and simulation [10, 18, 21, 24]. Recently MapReduce is becoming an element of the so-called NoSQL databases [14, 22].

Both map and reduce are blocking operations despite relaxations built in MapReduce implementations [18, 24]. Blocking means that a MapReduce job may proceed from mapping to reducing only if all mappers have finished. This is especially required by reducers which must first sort *all* the output from the mappers before proceeding to any further computation. Analogously, MapReduce completes only when all reducers finish. Any imbalance of completion times in the above operations degrades MapReduce performance. A dispersion of completion times may arise in many ways. For example:

1. A non-uniform distribution of keys key_1 between the splits results in unequal sizes of files created by different mappers, and imbalance in the shuffle phase.

2. Different frequencies of key_1s result in unequal file sizes in the shuffle phase and unequal workloads in reducing.

3. Keys are not equally easy to process.

4. Volatility of the environment: speeds of computers may change as a result of running background services, cluster sharing, degradation of the hardware, maintenance processes, applying energy-saving modes, etc. Speed of communication may change for similar reasons.

Usually mapping phase is relatively well protected against the computation time imbalance because splits have small size and each mapper is assigned splits in many iterations here called *waves* [16, 24]. A mapper which computes faster gets more splits. The fact that splits are processed in many waves has randomizing effect and e.g. unequal distribution of keys between splits has little effect on load imbalance. Consequently, granularity of mapping imbalance can be considered small. Implementations of MapReduce have provisions for a failure or prolonged processing of one split in mapping or one partition in reducing. Namely, if such an operation fails or progresses too slowly, then it can be re-executed speculatively. This resiliency comes at the cost of additional computation, communication and storage. Unfortunately, the distribution of keys and hence sizes of partitions submitted to reducers are application- and data-dependent and are hard to predict. Inevitably some reducers get more data to process than the others. Such a situation of unequal partition sizes is called data or partition *skew*. Speculative re-execution is not helpful here because a partition with skewed data will result in exactly the same prolonged computation on another computer. Since it is hard to balance reducers work in advance (see more details in the

next section), reducing computations must be re-balanced online. This paper is dedicated to algorithms mitigating partition skew in the reduce phase of MapReduce. The contribution of this paper is as follows:

- Four algorithms are proposed to mitigate partition skew in MapReduce computations: static, multi-dynamic, single-dynamic and mixed.
- For the purpose of comparing the algorithms a performance model of MapReduce computations is proposed.
- The above algorithms are evaluated by simulation against changes of various system parameters for two reducing complexity functions: linear and loglinear (a.k.a. linearithmic).
- Algorithm performance sensitivity to the selection of control parameters is evaluated.
- Robustness of the algorithms to the volatile environment performance is tested.

The remaining part of this paper is organized as follows. In the next section related publications are outlined. In Section 3 scheduling and performance model of MapReduce computations is introduced. Section 4 comprises algorithms designed to counteract load-imbalance in the reducing phase of MapReduce computation. The efficacies of these algorithms are evaluated in Section 5. The last section is dedicated to conclusions. The notation used throughout the paper is summarized in Table 1.

2 Related Work

MapReduce has been introduced in the seminal paper [10]. However, Hadoop [2] is more known currently as a MapReduce implementation. Hadoop is a set of open-source software modules for distributed processing of big datasets on parallel clusters. A detailed description of MapReduce in Hadoop can be found in [24]. Examples of MapReduce applications can be found, e.g., in [8, 10, 18, 21, 24]. Advantages and limitations as well as possible development strategies for MapReduce are discussed in [18].

Logs from MapReduce applications were analyzed in [16]. It has been found that in most cases mappers receive splits in many waves (iterations), which eases load-balancing of mappers. On the contrary, a vast majority of MapReduce jobs had fewer than 2 reduce waves. It can be speculated that most of the users follow the recommendation (e.g. in [24]) that the

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;
Δ	spread of key distribution (see Section 5);
δ	spread of sorting and reducing rate changes (see Section 5.3);
ε	size of master messages;
γ	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 key_1 values);
Π_i	partition i, a set of clusters;
s_i	size of partition $i = 0, \ldots, R-1$;
R	number of reduce partitions;
r	number of reducers (computers);
σ_i	size of κ_i cluster in bytes;
$T(z)$	reducer complexity function in partition size z; we consider either linear $T(z) = (a^{sort} + a^{red})z$, or loglinear $T(z) = a^{sort} z \log_2 z + a^{red} z$
X	fraction of keys processed in the static phase of the mixed algorithm;
V	initial load size, in bytes;

number of partitions should be close to the number of cluster computers. Thus, $r \approx R$, which exacerbates inequalities in data partitioning for reducers. While most of the jobs have relatively equal mapping task runtimes and reducing task runtimes, a significant fraction of jobs exist where runtimes are unequal [16]. Moreover, runtime dispersion was greater in case of reducing. An attempt in 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 application performance problems or diversified application-specific input data [16]. This signifies reacting online to data skew and other performance problems.

Imbalance of the mapping phase has been discussed in [17, 20], which appears to be a problem in itself. In this paper, however, we study mitigation of partition skew for reducing. Load-balancing the reducing phase has

been studied in [13, 15, 17]. An algorithm called fine partitioning has been proposed in [13]. 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 information about sizes of the partitions they created. The master sorts the partitions according to non-increasing partition processing time estimates and assigns them one by one to the reducers with the smallest total load. Thus, partitions are assigned on the basis of the Longest Processing Time (LPT) algorithm. Another algorithm called dynamic fragmentation is derived from fine partitioning. A mapper can initiate a split of a partition if its expected computational cost exceeds the average cost by some factor e. Factor e and the number f of new parts created from one partition are control parameters of the method. On completion of mapping, partition sizes are reported to the master. The master decides on the basis of reducing time estimations which new partition to keep and which one to ignore. Let Π_i denote partition on some mapper i. Note that if some 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 [15] algorithm LEEN has been proposed which assigns clusters one by one to computers for reducing on the basis of keys locality and total size of data stored on the machines. For this purpose a score is calculated which takes into account the number of keys held by a computer, the size of each key cluster held on a computer and the distance of the held data size from the mean data size of all machines. Note that LEEN has quite high complexity $O(|\Omega|r)$, where $|\Omega|$ is the number of different keys. LEEN requires $O(m|\Omega|)$ messages to communicate detailed key distribution data to the master. Moreover, it is not taking into account prolonging execution of some reducing task arising as a result of varying speed of a computer, or differences in the difficulty of the keys. Hence, we do not discuss LEEN in the further text. In [17] a SkewTune method mitigating skew dynamically has been proposed. Though SkewTune manages data skew both in mapping and in reducing, we consider 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.

Divisible load theory (DLT) is a scheduling and performance model for

data-parallel computations. It has been proposed in the late 1980s to schedule parallel computations in distributed sensor networks [9] and clusters of workstations [1]. In the following years DLT developed in many directions to cover various communication strategies, network topologies, memory limitations and cost optimizations [7, 11, 12, 19, 23]. Since MapReduce is a data-oriented and data-parallel processing paradigm, it naturally fits DLT methodology. Performance of MapReduce has been studied in [6] using divisible load model. 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. This means that the amount of data produced by mappers and shifted to reducing plays important role. When it is big, then the time of shuffling the data to reducers and sorting it easily dominates the whole computation time. This approach has been generalized in [3] to calculate optimum load distribution to mappers and reducers. The scheduling model [3] is useful for discovering effective schedule patterns which can be implemented in heuristics. Unfortunately, the proposed method has high computational complexity and requires reliable performance data which may be hard to obtain in a shared system with the ever-changing input datasets. Hence, it is hard to build optimum schedule for MapReduce job in advance and data skews must be overcome online.

3 Mathematical Model of MapReduce

In this section we define scheduling and performance model of MapReduce. On the modeling side it builds on DLT, on the practical side it refers to the Hadoop implementation of MapReduce.

The computation of MapReduce consists in processing $(key_1, value_1)$ pairs. Note that key_1 itself has values which are something different than $values_1$. In our example from the introduction key_1 is *country* which may have values {Afghanistan,Albania,Algeria,...}. Let Ω be the space of key values, and let $\kappa_i \in \Omega$ denote some key value. 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 σ_i (in bytes). A *partition* Π_j, for $j = 0, \ldots, R-1$, is a set of key values such as e.g. κ_i. It is required that $\cup_{j=0}^{R-1}\Pi_j = \Omega$ and $\forall i \neq j, \Pi_i \cap \Pi_j = \emptyset$. Let s_j denote size of partition Π_j (in bytes). The size of a partition is the sum of cluster sizes for the key values the partition comprises, i.e. $s_j = \sum_{\kappa_i \in \Pi_j} \sigma_i$.

MapReduce starts with the map phase in which the input dataset is processed in split units. Processing a split is a *map task* consisting of retrieving

the split, computations on its data, and distributing the results to R different output files on the basis of key_1, where R is the number of partitions. Map tasks are executed sequentially and managed independently by the MapReduce environment such as Hadoop. Though splits have discrete nature, we assume that the influence of split size choice has negligible impact on the map phase duration. A more detailed discussion of this model and impact of split size on mapping duration can be found in [4, 6]. We assume that mappers process the splits with speed $1/A$ (A is in seconds per byte). Let V denote the size of the input dataset (in bytes) and m the number of mappers (computers). The input dataset is distributed roughly equally between the mappers and each mapper processes volume V/m. The mapping phase duration is VA/m. We assume that for α bytes of input data $\alpha\gamma$ bytes of output data are produced by the mappers. Hence, 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 report sizes of output files to the master and the size of a message is ε per output file.

The master decides on assigning partitions to the reducers, possibly using a load-balancing algorithm (discussed in the next section). In this work we assume that key clusters are not moved between partitions at this stage. It means that all clusters contained in a given partition are read by one reducer. Thus, unlike in the dynamic fragmentation algorithm [13] mentioned earlier, there is no proactive partition sizing by mappers. 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 shuffle phase reducers read files from the mappers. Shuffle involves communication over the cluster communication network. We assume that each machine has bandwidth $1/C$, the cluster 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 channels which can be open in the cluster between pairs of different computers without bandwidth degradation. If the number of concurrently open channels exceeds l, then the cluster bandwidth is shared. For example, if x machines simultaneously open communication channels, then the bandwidth perceived by a machine is $\min\{1, l/x\}/C$. In the simulations described in Section 5 we trace the time moments when machines start and finish communications in order to determine the number of simultaneously open channels and the resultant bandwidth perceived by the communicating processes.

After reading the assigned partitions reducers start processing them. Processing a partition consists in two operations: sorting key-value pairs by key_1 and then processing them and storing the results. We assume that for a partition of size z (in bytes) the reduce task has runtime $T(z)$. We will consider two types of reducer complexity functions: $T(z) = (a^{sort} + a^{red})z$ for linear-time sorting such as radix sort or bucket sort, and loglinear $T(z) = a^{sort}z\log_2 z + a^{red}z$ representing 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 reconstructed and its data redistributed, then it can be done after sorting.

4 Load-balancing Algorithms

In this section algorithms for balancing partition skew will be presented. For practical reasons these algorithms must have low runtime. Thus, in the cases of solving a hard computational problem, such as bin-packing, we have to recourse to heuristics. Let us remind that partitions are built on the basis of key clusters and one cluster is processed by one reducer.

4.1 Reference Distribution Algorithm

Our algorithms will be compared to a standard MapReduce execution with one partition per reducer ($R = r$) and no additional load balancing. We always use a simple partitioning function which places key κ_i in partition $\Pi_{i \bmod R}$. In the reference distribution algorithm the master assigns partition Π_j to reducer $j+1$, for $j = 0, \dots, r-1$. The reducers read data from the mappers using a special communication pattern which guarantees that no computer performs two communications at the same time. Namely, if $m \leq r$, then mapper i communicates consecutively with reducers $i, i+1, \dots, r, 1, \dots, i-1$. If $m > r$, then reducer j communicates consecutively with mappers $j, j+1, \dots, m, 1, \dots, j-1$. 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.

Let us finish this subsection with a calculation of the schedule length for the reference distribution algorithm in the ideal case when all partitions happen to have equal size. Mappers run in time AV/m and communicate sizes of the output files to the master in time $Cmr\varepsilon$ because the reads of the master are sequential. The master immediately calculates distribution of the r partitions and sends information on the mr file locations to r reducers in time $Cmr\varepsilon$. Each mapper produced r files of size $\gamma V/(mr)$ and each partition has size $\gamma V/r$, 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

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

Let us remind that (1) provides only an optimistic estimation because very often partitions are unequal. In such cases communication and computation times will be calculated according to the actual load sizes.

4.2 Static Algorithm

The first load balancing algorithm is based on the idea of fine partitioning [13]. We call it a static algorithm, because the load 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 non-increasing sizes and assigns them one by one to the reducers with the smallest total load. This can be done in total time $t^{master} = a^{master}(krm + kr\log(kr) + kr\log r)$, 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. In the early version of this algorithm presented in [5] a reducer sorted all its data (from

different partitions) merged together. Here, the partitions are processed separately and sequentially. This decreases the total sorting time if the sorting complexity is higher than linear.

4.3 Multi-dynamic Algorithm

In contrast to the static approach, the dynamic methods we propose for mitigating skew in MapReduce consist in improving the load distribution during the reducing phase. The multi-dynamic algorithm performs multiple load balancing operations and is based on the idea similar to SkewTune [17]. The number of partitions created by the mappers is $R = r$ and the application execution is the same as in the reference algorithm (Section 4.1) until the moment when all reducers finish sorting and at least one reducer finishes processing. 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 other reducer is computed so that both computers finish processing at the same time. Let us remind that key clusters cannot be divided between different processors. Therefore, the amount of transferred load usually differs from the value computed by the master. In order to approximate the desired size the 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 data they contain 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 a reducer becomes idle.

4.4 Single-dynamic Algorithm

A disadvantage of the multi-dynamic algorithm is that a reducer can start receiving data from more loaded computers only after it processed all its load. Thus, it has to wait idle until new data arrive. Therefore, we propose a single-dynamic algorithm which predicts the requests to balance the load in order to start transferring the data earlier. Again, there are $R = r$ partitions and the algorithm starts when all reducers finish sorting and at least one reducer finishes processing. At this moment the master sequentially stops the reducers which still have some load to process. Each of them informs the master about the amount of its remaining load. Based on this data, the master computes the expected time of finishing the application. Then, it analyzes the scenario in which the most loaded reducer sends part of its remaining data to the least loaded reducer (which already finished processing

its part), so that they finish computations at the same moment. The master assumes in simulation that it is possible to divide the load into parts of any sizes, although in practice rounding to cluster sizes is necessary (as explained in Section 4.3). If balancing the load assignment between the selected pair of reducers is profitable, the master checks the next scenario, in which two most loaded processors hand over parts of their data to the least loaded reducers. This process continues as long as it is possible to decrease the total processing time. The running time of master computations is $a^{master}(r \log r + Nr)$, where N is the number of analyzed iterations. After choosing the best option, the master notifies the reducers to resume computing and informs them about the load parts they should send to different computers. The reducers perform the operations indicated by the master, rounding the transferred load sizes to whole clusters as in the multi-dynamic algorithm.

Let us remark that the pattern of moving clusters between the partitions obtained by the single-dynamic algorithm is much simpler than what can be achieved by the multi-dynamic method, as each reducer sends or receives additional load at most once. This approach was chosen for practical reasons, in order to avoid running complicated and time-consuming simulations by the master as well as to avoid repartitioning the clusters and transferring data many times. However, in the case of big imbalance in the initial load distribution it may be disadvantageous.

4.5 Mixed Algorithm

The mixed algorithm is meant to combine the static and the dynamic approach to skew mitigation. Hence, it works both before shuffle and during the reducing phase. We will call these two stages of the mixed algorithm the static part and the dynamic part, correspondingly. 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 $\lfloor X|\Omega| \rfloor$ 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 by $i \bmod k_2 r$. Thus, the total number of partitions is $R = (k_1 + k_2)r$. When mapping is finished, the $k_1 r$ partitions to be distributed statically are assigned 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 data from the mappers and performs computations. These operations are repeated until all partitions are processed.

5 Evaluation of the algorithms

In this section we compare the effectiveness of the load-balancing algorithms against changing system parameters. We created a program in C++ that simulates the execution of MapReduce with load balancing methods described in Section 4. In order to precisely compute the schedule length, the simulator maintains a queue of system events, such as the start or end of a communication between two processors or computations on some processor. Let us note that the events may depend on one another in many ways. For example, starting a new communication may lead to decreasing the speed of all data transfers taking place at the same time and hence, to delaying their completion. In such a case all affected events are properly updated by the simulator.

We will now describe our method of generating test instances. In all test instances the number of keys in the input data is $|\Omega| = 1E6$. The key frequencies are obtained in the following way. For each key κ_i a number f_i is selected randomly with uniform distribution from interval $(1 - \Delta, 1 + \Delta)$. The frequency of κ_i is set to $f_i / \sum_j f_j$. Thus, the key frequencies are not very unbalanced, as the maximum possible cluster size is $1 + \Delta$ times greater than the average. We distinguish two distributions of the keys between the partitions: *hard* distribution and *easy* distribution. For the simplicity of presentation let us assume that the number of keys per reducer $|\Omega|/r$ is an integer. Let us remind that in the reference distribution algorithm keys are assigned to partitions based on their numbers, i.e. key κ_i belongs to partition $\Pi_{i \bmod R}$. In the hard case we renumber the keys after selecting their frequencies, so that the first partition contains $|\Omega|/r$ most frequent keys, the next partition contains $|\Omega|/r$ of the remaining most frequent keys, etc. The last partition contains the keys which are the least frequent. Thus, in the hard instances the partitions are unbalanced. In the easy case the numbering of keys is not changed. This can be seen as assigning keys to the partitions randomly. Hence, it can be expected that on average the sizes of the partitions are not very imbalanced in the easy instances.

Unless stated otherwise, we assume the following system and application parameter values: $V = 1E12$, $\varepsilon = 8$, $m = 1000$, $r = l = 100$, $\gamma = 0.1$, $C =$ 1E-8, $A = a^{master} = a^{sort} =$ 1E-7, $a^{red} =$ 1E-6, $\Delta = 1$. The reducing rate

a^{red} is greater than the remaining computation rates in order to emphasize the reducing phase, for which load balancing is crucial. For each tested setting 10 instances were generated. The measure of the algorithm quality is the average speedup in comparison to the reference distribution algorithm (Section 4.1).

Our experiments can be divided into three groups. First, in Section 5.1, we tune the algorithm parameters and choose the best values of k for the static algorithm and X, k_1, k_2 for the mixed algorithm. The impact of the system and application parameters on the performance of the algorithms is analyzed in Section 5.2. In Section 5.3 we study the robustness of the algorithms to the volatile speed of sorting and reducing. As the number of parameters is high, it is not possible to evaluate all possible configurations. Therefore, we use a simple one-factor experiment design.

5.1 Algorithm Parameter Sensitivity Analysis

Let us start with analyzing the performance of the static algorithm with $k = 2, 3, \dots, 10$, for the reference system configuration. Note that in the static algorithm each mapper creates kr output files on its disk. Too big values of k may be impractical or may decrease the mapping speed. This is why we chose $k = 10$ as the maximum value. The results are presented in Fig. 1. Let us first remark that the speedup values are much greater for the hard instances (Fig. 1a) than for the easy instances (Fig. 1b). The reason is that in the easy tests the initial load distribution is already rather balanced and there is not much place for improvements. This is visible for all algorithms in all our experiments. The speedup obtained for loglinear sort complexity is greater than for linear sorting. This is a result of the fact that dividing the load into smaller parts before sorting decreases the sorting time for complexities higher than linear and has no effect for linear complexity. The performance of the static algorithm for the linear sort is roughly the same for $k = 2, \dots, 10$, for both types of instances. This means that $k = 2$ is sufficient to balance the load distribution. An advantage of such a small value is that the costs of operating on kr files by each mapper and sending the information about kr partition sizes are not big for $k = 2$. However, larger k allows for shortening the processing time in the case of loglinear sorting (see Fig. 1b). Therefore, in the further experiments we will use both $k = 2$ and $k = 10$.

For the mixed algorithm we tested the following parameter ranges: $X \in \{0.1, 0.2, \dots, 0.9\}, k_1, k_2 \in \{1, 2, 5, 10\}$. All combinations of the above values were analyzed. Selected results are presented in Fig. 2 and 3. Fig. 2a shows the results for hard instances and linear sorting, for $k_1 = 1$. This value means

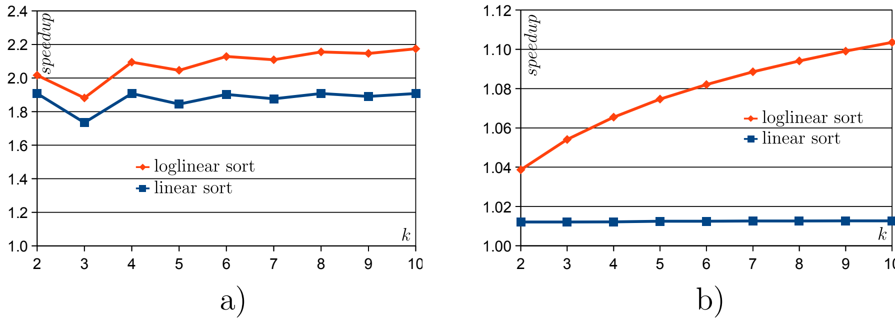

Figure 1: Speedup of the static algorithm vs. k. a) Hard instances, b) easy instances.

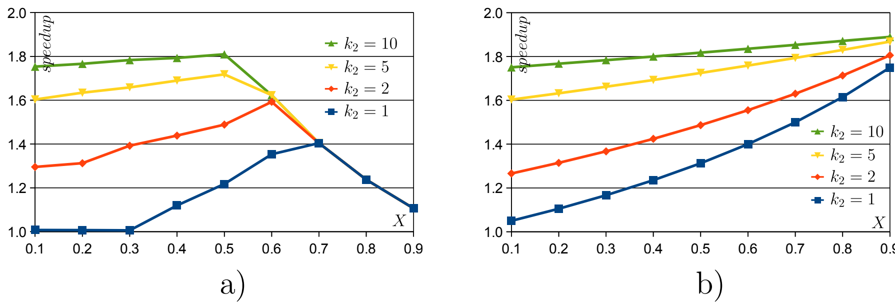

Figure 2: Speedup of the mixed algorithm for linear sort complexity, hard instances vs. X, k_2. a) $k_1 = 1$, b) $k_1 = 10$.

that no load balancing is performed in the first, static part of the algorithm. Therefore, it is not profitable to process too much load in this phase and the performance decreases when X gets big. The results for $k_1 = 10$ are presented in Fig. 2b. Let us note that the speedup values obtained for $k_1 = 2, 5$ (not shown here) are very similar to the ones for $k_1 = 10$. This conforms with the observation that $k = 2$ is enough for the static algorithm when sorting is linear. It can be seen in Fig. 2b that the performance increases with growing X and k_2. This means that most of the load should be carefully balanced in the static part of the algorithm. Only a small part of data should be processed in the dynamic phase.

Fig. 3 presents the results for the easy instances with loglinear sorting. The key to achieve good speedup is now to decrease sorting time. The best strategy is to create many partitions of similar sizes. Thus, if $k_1 < k_2$, then most of the load should be partitioned dynamically (it is good to choose small X in Fig. 3a for $k_2 = 5, 10$). Similarly, if $k_1 > k_2$, then X should be large (see $k_2 = 1, 2$ in Fig. 3b). Finally, for $k_1 = k_2$ it is best to choose $X = 0.5$.

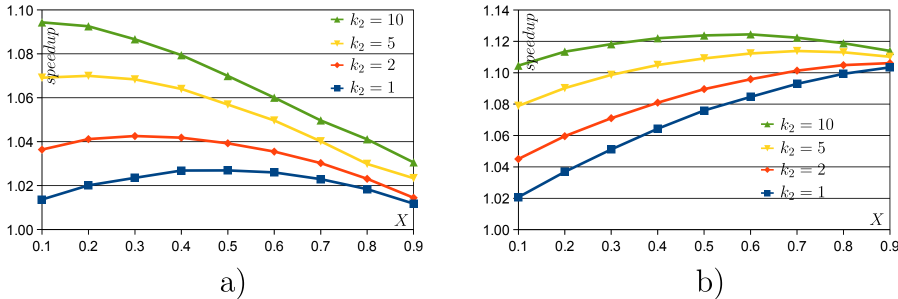

Figure 3: Speedup of the mixed algorithm for loglinear sort complexity, easy instances vs. X, k_2. a) $k_1 = 1$, b) $k_1 = 10$.

Selecting big k_1 and k_2 at the same time was profitable for all generated instances. The best results for $k_1 = k_2 = 10$ were obtained for $X = 0.5$ or $X = 0.9$, depending on the type of the instance and the complexity of sorting. Thus, in the further experiments we will analyze two versions of the mixed algorithm: with $X = 0.5, k_1 = k_2 = 10$ and with $X = 0.9, k_1 = k_2 = 10$.

5.2 System Parameter Sensitivity Analysis

In this section we analyze the impact of system parameters on the performance of the algorithms. For the sake of brevity we will refer to the static algorithm with some value of parameter k as to static(k) and the mixed algorithm with $k_1 = k_2 = 10$ and some value of parameter X as to mixed(X).

We will first study the influence of the number of reducers r. Note that with increasing r the average amount of load to be sorted and processed by a reducer decreases. The results for r between 2 and 1000 are presented in Fig. 4. In the instances with $r > 100$ the bisection width limit was increased to $l = r$ so that it would not affect the communication speed. Let us start with the observation that the differences between speedups of the mixed, static and dynamic algorithms are much larger for loglinear sorting (Fig. 4a,c) than for linear sorting (Fig. 4b,d). When sort complexity is higher than linear, then sorting data in several portions separately takes less time than sorting them all together. The single-dynamic and multi-dynamic algorithm divide the load into r partitions. The static(2) and static(10) algorithms produce $2r$ and $10r$ partitions, respectively. The mixed(0.9) algorithm creates $10r$ "big" and $10r$ "small" partitions, and the mixed(0.5) algorithm produces $20r$ partitions of similar sizes. Hence come the relations between the algorithm quality. With nonlinear sorting, the influence of dividing load obtained by a reducer into smaller parts is much stronger than that of load

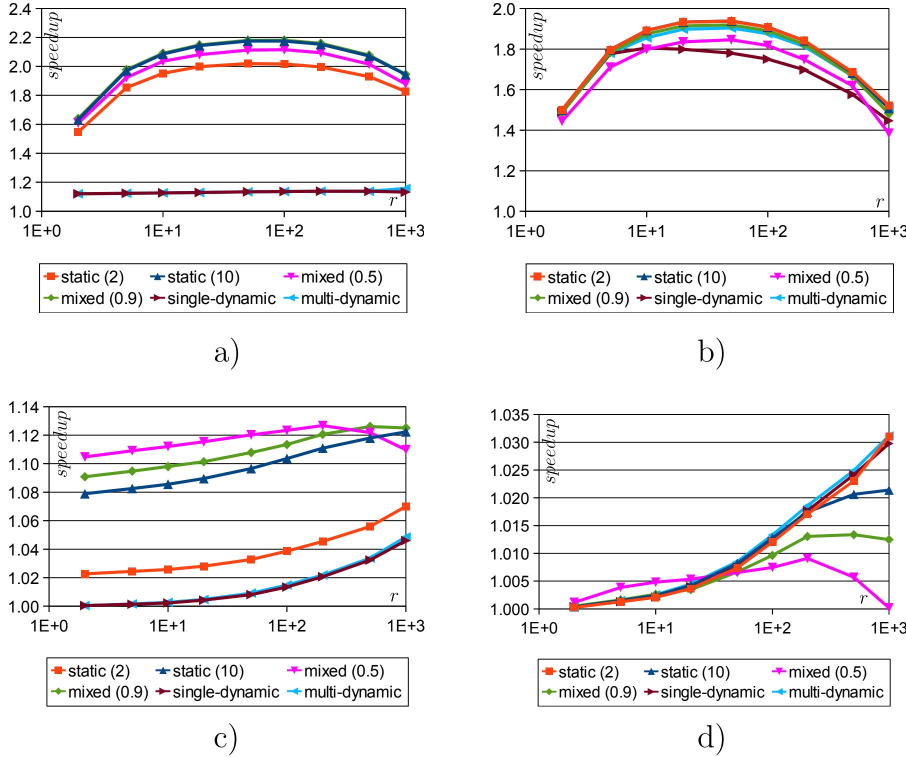

Figure 4: Speedup vs. r. a) Hard instances, loglinear sort, b) hard instances, linear sort, c) easy instances, loglinear sort, d) easy instances, linear sort.

balancing. This effect will be visible in all experiments presented in this section. For the linear sort, the only factor is the balance of load distribution.

We will now analyze how the speedups change when r increases. For the hard instances (Fig. 4a,b) the speedup obtained by all algorithms first increases and then decreases. When the number of reducers is very small, it may be hard to find a well balanced load distribution. The situation gets better with increasing r. However, when the number of reducers is very big, the sorting and reducing phases become shorter in comparison to mapping. As load balancing does not affect mapping time, the speedup of the whole application obtained by improved load balancing decreases. For loglinear sort, the best speedup is achieved by mixed(0.9) and static(10) algorithms, because they generate many partitions for each reducer. The performance of mixed(0.5) is worse, although it creates $20r$ partitions of similar sizes. It seems that transferring half of the load only on request of idle reducers significantly damages performance. For linear sort, the best results are obtained by static(2) and static(10) algorithms, which balance

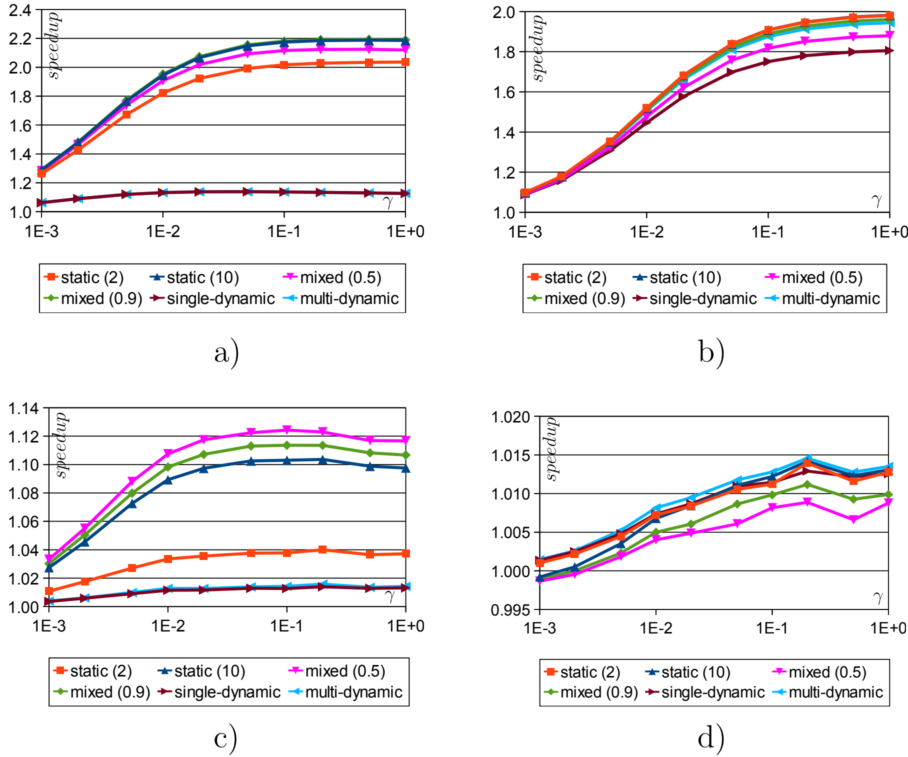

Figure 5: Speedup vs. γ. a) Hard instances, loglinear sort, b) hard instances, linear sort, c) easy instances, loglinear sort, d) easy instances, linear sort.

the load very well (the lines overlap in Fig. 4b). The single-dynamic and mixed(0.5) algorithm perform worst, although the overall difference is not very big.

Different tendencies may be observed for the easy instances (cf. Fig. 4c,d). The quality of the static and dynamic algorithms grows with growing r. As the range of speedup is smaller for easy instances, the effect of shortening the reducing phase is weaker. However, the costs of managing many partitions become visible and result in the performance drop of the mixed algorithms and slower growth of the static(10) speedup. This is especially evident for the mixed(0.5) algorithm, in which a big portion of load is transferred to reducers quite late, after they finished processing the load distributed statically. For loglinear sort, it is still profitable to use the algorithms that create many partitions when r is large, due to shorter sorting time. For linear sort and big r the algorithms creating many partitions are worse than single-dynamic, multi-dynamic and static(2) algorithms, although the overall difference in speedup is again small.

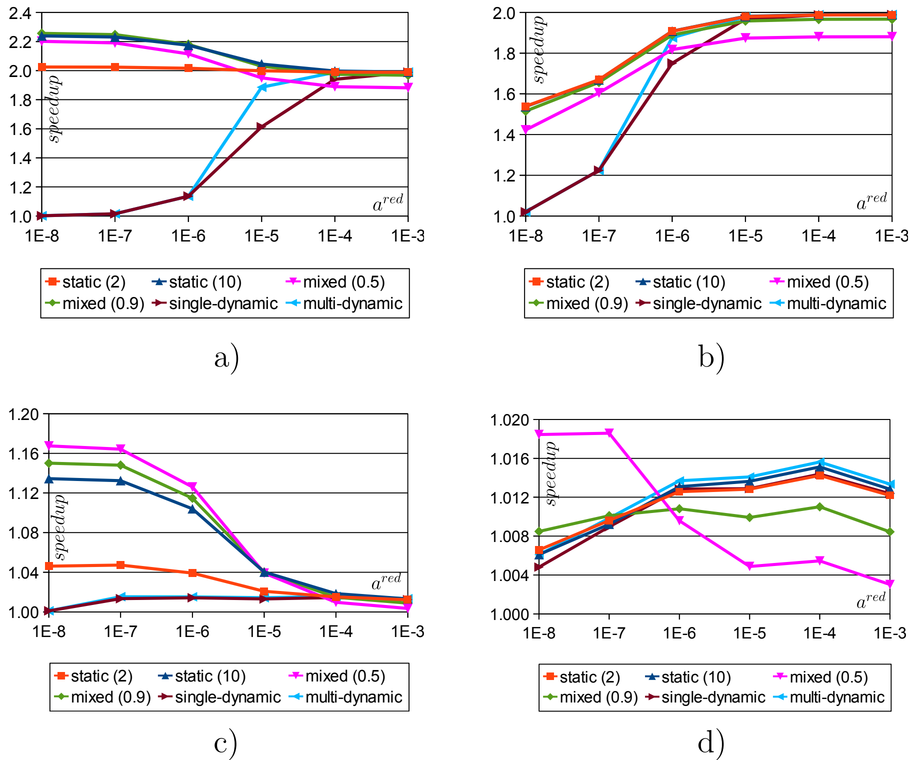

Figure 6: Speedup vs. a^{red}. a) Hard instances, loglinear sort, b) hard instances, linear sort, c) easy instances, loglinear sort, d) easy instances, linear sort.

In the next series of experiments we analyzed parameter γ which controls the amount of load that is transferred from the map phase to the shuffle and reduce phase. Thus, growing γ increases the contribution of sorting and reducing in the total application running time. As a consequence, the speedup obtained by all algorithms also increases (see Fig. 5). It seems that γ does not affect the relationships between different algorithms. The order of the algorithms, from the greatest to the smallest speedup, is similar as in Fig. 4 with $r = 100$, for both instance types and sorting complexities.

The speedups for different values of reducing rate a^{red} are presented in Fig. 6. Let us remind that increasing a^{red} makes the reducing phase more computationally demanding compared to mapping, shuffle and sorting. On the one hand, it exposes the need for balancing a long reducing stage and hence allows for achieving better speedup of the whole application. On the other hand, for loglinear sort complexity, the static and mixed algorithms achieve good speedup mostly in the sorting phase, which becomes less im-

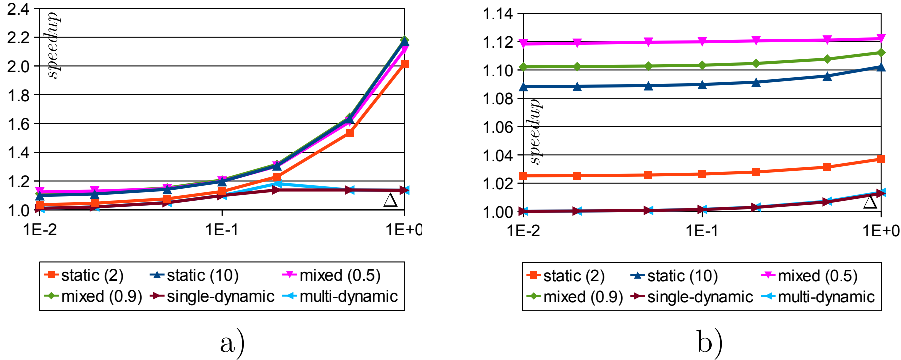

Figure 7: Speedup vs. Δ. a) Hard instances, loglinear sort, b) easy instances, loglinear sort.

portant when a^{red} is very big. Hence, the speedups of the static and mixed algorithms increase in Fig. 6b (linear sort), but decrease in Fig. 6a,c (loglinear sort). The single-dynamic and multi-dynamic algorithms do not lose performance when sorting becomes less important. Thus, their performance improves with growing a^{red} in all cases. An interesting phenomenon can be seen in Fig. 6d, where the performance of mixed(0.5) algorithm significantly decreases with growing a^{red} for easy instances with linear sorting. This is a similar situation as in Fig. 4d. The maximum possible speedup is small due to the initial good load balance and hence, the time when processors wait idle for more load in the dynamic part of the mixed algorithm becomes important and leads to decreasing performance.

Fig. 7 shows the impact of key distribution dispersion parameter Δ on performance of the algorithms. The smaller Δ is, the smaller differences between key frequencies. We present the speedups for loglinear sort only because for linear sort (not shown in Fig. 7) the differences between the algorithms for $\Delta < 1$ are negligible. As could be expected, the speedups of all algorithms grow, as larger unbalance in the data gives more space for improvements. The relations between the results of different algorithms are determined by the numbers of created partitions, which affect the sorting time.

We also tested the impact of the remaining system parameters m, C, l, V on the performance of the algorithms, which we report in short only. As increasing the number of mappers m means shortening the mapping phase compared to the remaining parts of processing, it has a similar effect as increasing γ (cf. Fig.5). Increasing parameter C means slower communication, and hence it decreases the speedup of all algorithms. It has the strongest ef-

Table 2: Numbers of instances for which particular algorithms obtained best results.

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
single-dynamic	0	0	0	0	0
multi-dynamic	0	9	6	**355**	370

fect on the single-dynamic and multi-dynamic algorithms because they transfer large amounts of data between reducers. The speedup of the algorithm static(2) almost does not change, because the total size of data sent in the network is very close to the size of messages sent in the reference distribution algorithm. Decreasing l also decreases the communication capabilities of the network, but the speedup is only slightly affected. This can be attributed to the fact that the additional messages used by the balancing algorithms very rarely use all the available bisection width. For sequential communication (e.g. sending messages to and from the master) the bisection width limit l is meaningless. The total load size V does not affect the effectiveness of load balancing.

In total, 610 test instances with various combinations of system parameters were solved for each type of key distribution and sorting complexity. It appears that no algorithm dominates in all possible cases. In some situations the differences are minor. Therefore, we summarize algorithm performance in Table 2 as the number of instances for which certain algorithm constructed the best solution. For the hard instances and loglinear sort complexity the best algorithm is mixed(0.9), as it creates the greatest number of partitions and distributes only a small amount of load dynamically. When the complexity of sorting is linear, dividing data into many parts is not so important. For hard instances and linear sort the best results are obtained by static(10) and static(2) algorithms. The easy instances are best solved by the algorithms that balance the load dynamically, in many steps: multi-dynamic (linear sort) and mixed(0.5) (loglinear sort). It is worth noting that the mixed(0.5) algorithm won the greatest total number of instances. The single-dynamic algorithm did not win for any instance we solved. This means that balancing the load by creating pairs of reducers exchanging parts of data is too simplistic.

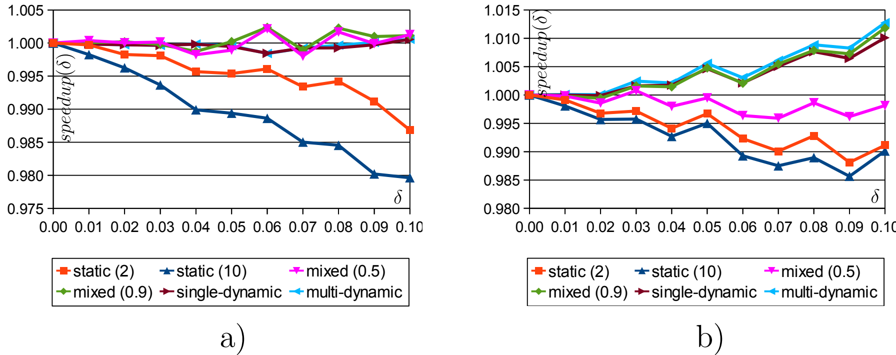

Figure 8: Speedup(δ) vs. δ. a) Hard instances, loglinear sort, b) easy instances, loglinear sort.

5.3 Robustness

The assumption that computing speed is constant is unrealistic in many cases. Even small changes in the machine performance may lead to a significant dispersion in processor running times and longer application execution. In this section we study the robustness of the load-balancing algorithms to changing parameters a^{sort} and a^{red}. In the following experiments values of these parameters changed every 100 seconds and were chosen randomly, independently for each reducer, from ranges $(a^{sort}(1-\delta), a^{sort}(1+\delta))$ and $(a^{red}(1-\delta), a^{red}(1+\delta))$, correspondingly. Parameter δ is controlling the variability of computing speed. For example, $\delta = 0$ means that the processing rates are constant, and $\delta = 0.1$ means that the processing rates can differ by at most 10% from the expected value.

We will start with analyzing the influence of changing computer speed on the speedup of load balancing algorithms. For clarity of presentation we will divide the speedup obtained for given value of parameter δ by the speedup obtained in a system with no computer speed changes. We will denote this measure as speedup(δ). Thus, speedup(0) always equals 1. The results are shown in Fig. 8. We discuss loglinear sorting only, as the results obtained for both sort complexities were very similar. It can be seen that the static algorithms are the least robust to the changing computing speed. Indeed, they try to divide the load equally between reducers and have no information about the real speed of processing. Thus, they cannot react when some reducers are slower than others. The dynamic and mixed algorithms have knowledge about the actual times when processing by some reducers finished and use it to better balance the further computation. Both dynamic algorithms and the mixed(0.9) algorithm achieve speedup(δ) greater than

1 for the easy instances and big δ. This means that they can counteract the drop of performance that occurs for the reference algorithm when the environment becomes volatile. Still, it should not be forgotten that though static algorithms loose ≈2% in speedup due to varying processing speed (Fig.8), their reference performance for hard instances with loglinear sort is by 100% better than, e.g., the single- and multi-dynamic algorithms (Fig.4a-6a).

Besides the speedup changes, we studied the influence of parameter δ on the dispersion of the reducer completion times. Let us define for $i = 1, \dots, r$ value $\tau_i = t_i/T$, where t_i is the completion time on reducer i and T is the total application running time. Thus, τ_i represents the fraction of the schedule length in which reducer i is active. As reducer completion times dispersion measures we used the standard deviation and the interquartile range (IQR) of values τ_i. Since the results were very similar in both cases, we present the IQR only. However, to better understand the influence of speed changes, let us first report on the dispersion of completion times in a system with no computing speed changes, for different values of a^{red}, shown in Fig. 9. It turned out that the sorting complexity does not affect these results strongly, so we present only loglinear sort. All proposed algorithms decrease the dispersion of values τ_i compared to the reference algorithm. Only the performance of the multi-dynamic algorithm strongly depends on a^{red}. Let us remind that this algorithm starts working after all reducers finish sorting and every time when a reducer becomes idle it performs only one balancing operation. Thus, if a^{red} is very small, the multi-dynamic algorithm does not have enough time to balance the load. However, when a^{red} becomes bigger, the algorithm uses multiple balancing operations and obtains the best IQR, close to 0. In most cases the single-dynamic and the static algorithms give smaller IQR than the mixed algorithms. It seems that sending portions of data on requests from idle reducers, as the mixed algorithms do in the dynamic stage, causes larger differences in reducer completion times. Still, the IQR of mixed algorithms is less than 5%.

Let us now return to the experiment with changing speeds. Fig. 10 shows the changes in IQR of values τ_i for growing δ. We present the results for linear sort only, because the observed tendencies are similar for both sort complexities, and the range of IQR changes is greater in the case of linear sort. For clarity we show the IQR for given δ divided by the IQR obtained in the system with no speed changes. This measure is denoted by IQR(δ). The results confirm our observations about the robustness of the algorithms based on Fig. 8. The relative completion time dispersion IQR(δ) of the static algorithms strongly increases with growing system volatility. The single-dynamic algorithm is also affected (the lines for static(2) and single-dynamic

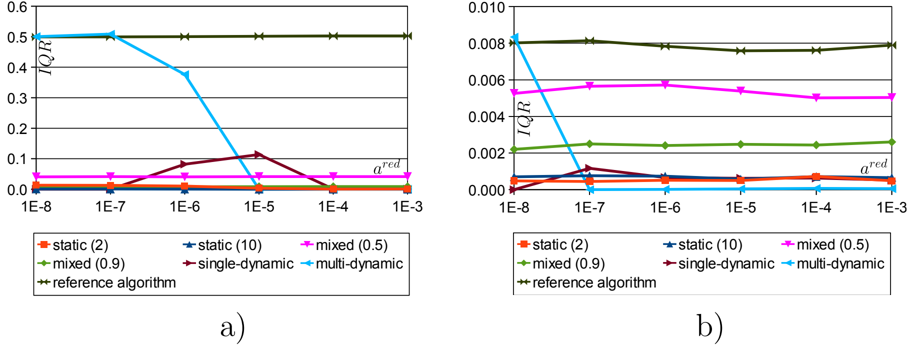

Figure 9: IQR of τ_i vs. a^{red}. a) Hard instances, loglinear sort, b) easy instances, loglinear sort.

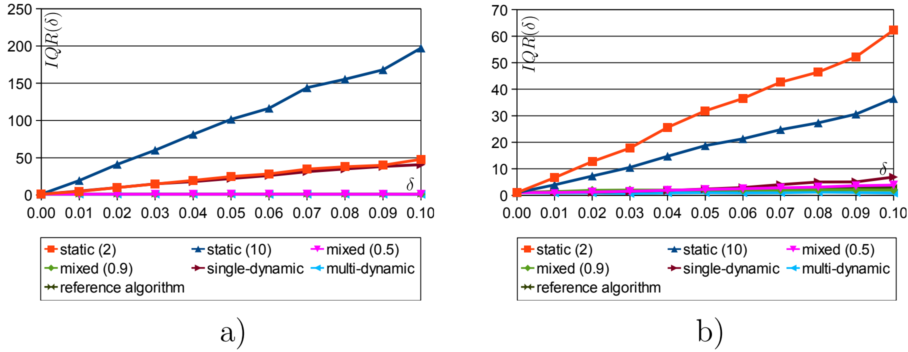

Figure 10: IQR(δ) of τ_i vs. δ. a) Hard instances, linear sort, b) easy instances, linear sort.

algorithm overlap in Fig. 10a). Although it has information about the actual sorting finish times, it cannot react to changing reducing speeds. Thus, the static and single-dynamic algorithms have small IQRs in a system without speed changes, but they can get very bad in a volatile system. The multi-dynamic and the mixed algorithms adjust the load balance many times, using the information about the real processing times. Hence, they are most robust and δ has almost no impact on them.

6 Conclusions

In this work we analyzed the problem of mitigating the partition skew in MapReduce computations, using the divisible load model. We proposed four methods of handling the skew. The static algorithm adjusts the load dis-

tribution before the shuffle phase, using the idea of fine partitioning. The multi-dynamic algorithm performs many simple load balancing operations in the reducing phase. The single-dynamic algorithm simulates the application execution in order to handle the skew in one complex operation. Finally, 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.

The algorithms were tested in a series of computational experiments. It was shown that the performance of the algorithms depends on the complexity of sorting by the reducers. If the sorting complexity is loglinear, then it is profitable to divide the data into a big number of partitions. Hence, the mixed algorithms and static(10) perform best. For linear sorting, the best results are obtained by the static algorithms for hard instances, and multi-dynamic and mixed(0.5) for easy instances. The single-dynamic algorithm is clearly outperformed by the other methods. Table 2 shows types of workloads in which certain algorithm dominates.

We also tested the robustness of the algorithms to the changing sorting and reducing speeds. The static algorithms operate on load sizes, not processing times, and hence their performance decreases in a volatile system. All the algorithms decrease the dispersion in reducer completion times in comparison to the standard MapReduce organization. If reducing is slow, then the multi-dynamic algorithm achieves the smallest dispersion.

Overall, it can be concluded that no single algorithm is a panacea. Depending on the type of workload or system features some types of algorithms may be better than the others. It seems that the single-dynamic algorithm is the weakest choice. A mixed algorithm, possibly with some tuning of parameter X, is a good compromise.

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 aim at a given, not necessarily equal load distribution between the reducers. This may be useful for optimizing the execution time of chains of MapReduce applications producing data one for another.

References

[1] R. Agrawal, H.V. Jagadish, Partitioning Techniques for Large-Grained Parallelism, IEEE Transactions on Computers 37, 1627-1634 (1988)

[2] Apache Software Foundation, Welcome to Apache Hadoop, 2014, `http://hadoop.apache.org/`

[3] J. Berlińska, M.Drozdowski, Scheduling Multilayer Divisible Computations, RAIRO Operations Research, vol.49, No.2, 2015, 339-368.

[4] 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

[5] J. Berlińska, M. Drozdowski, Mitigating Partitioning Skew in MapReduce Computations, Proceedings of the 6th Multidisciplinary International Scheduling Conference: Theory & Applications, Ghent 2013, 80-90.

[6] J. Berlińska, M. Drozdowski, Scheduling Divisible MapReduce Computations, Journal of Parallel and Distributed Computing 71, 450-459 (2011)

[7] V.Bharadwaj, D.Ghose, V.Mani, T.Robertazzi, Scheduling divisible loads in parallel and distributed systems. IEEE Computer Society Press, Los Alamitos, CA (1996)

[8] 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

[9] Y.-C.Cheng, T.G.Robertazzi, Distributed computation with communication delay, IEEE Transactions on Aerospace and Electronic Systems 24, 700-712 (1988)

[10] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004)

[11] M. Drozdowski, W. Głazek, Scheduling divisible loads in a three-dimensional mesh of processors, Parallel Computing 25, 381-404 (1999)

[12] M.Drozdowski, Scheduling for Parallel Processing. Springer (2009)

[13] B. Gufler, N. Augsten, A. Reiser, A. Kemper, Handling Data Skew in MapReduce, CLOSER '11: Proceedings of the 1st International Conference on Cloud Computing and Services Science, 574-583 (2011)

[14] HBase and MapReduce, Apache Software Foundation, 2014, https://hbase.apache.org/book/mapreduce.html

[15] S.Ibrahim, H. Jin, L. Lu, S. Wu, B. He, L. Qi, LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud, CloudCom '10: Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science, 17-24 (2010)

[16] S.Kavulya, J.Tan, R.Gandhi, P.Narasimhan, An Analysis of Traces from a Production MapReduce Cluster, Carnegie Mellon University, Parallel Data Laboratory, Technical Report CMU-PDL-09-107, 2009

[17] Y. Kwon, M. Balazinska, B. Howe, J. Rolia, SkewTune: Mitigating Skew in MapReduce Applications, SIGMOD '12: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 25-36 (2012)

[18] K.-H.Lee, Y.-J.Lee, H.Choi, Y.D.Chung, B.Moon, Parallel data processing with MapReduce: a survey, ACM SIGMOD Record, Vol.40, Issue 4, December 2011, 11-20.

[19] X.Li, V.Bharadwaj, C.C.Ko, Processing divisible loads on single-level tree networks with buffer constraints, IEEE Transactions on Aerospace and Electronic Systems 36, 1298 - 1308 (2000)

[20] J.Lin, The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce, Proceedings of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval, 2009

[21] J. Lin, C. Dyer, Data-Intensive Text Processing with MapReduce. Morgan & Claypool (2010)

[22] Mongo DB Manual 2.4, MongoDB Inc., 2014, `http://docs.mongodb.org/manual/core/map-reduce/`

[23] T.Robertazzi, Ten reasons to use divisible load theory, IEEE Computer 36, 63-68 (2003)

[24] T.White, Hadoop: The Definitive Guide, O'Reilly Media 2012.