Mitigating Partitioning Skew in MapReduce Computations

J. Berlińska · M. Drozdowski

Abstract In this paper we analyze handling partitioning skew in MapReduce computations. The basic MapReduce implementations strongly depend on the assumption that the data is partitioned evenly for reducing. However, in practical applications the data distribution is often skewed, what leads to decreasing MapReduce system performance. Using divisible load theory we analyze two methods of handling data skew in MapReduce computations. The proposed algorithms are evaluated in a series of computational experiments. To our best knowledge this is the first analytical study comparing mitigation of partitioning skew in two different stages of MapReduce applications.

1 Introduction

MapReduce is a parallel programming model for processing large data sets on big numbers of computers [10,17]. A MapReduce application consists of two stages: mapping and reducing. In the first stage a user-defined *Map* function processes the input dataset (e.g. a text or HTML file), and generates a set of intermediate $(key1, value1)$ pairs. In the second step the intermediate pairs are sorted by $key1$, and a *Reduce* function merges pairs with equal values of $key1$, to produce a list of result pairs $(key1, value2)$. As an example, consider constructing the inverted index for a set of documents, where all documents comprising certain *words* must be found [10]. The *Map* function parses the documents and generates a sequence of pairs (*word*, $docID$), where $docID$ is a document identifier (e.g., a URL of a web page). The *Reduce* function accepts all (*word*, $docID$) pairs containing a given *word*, sorts them and returns a (*word*, $list\_docIDs$) pair, where $list\_docIDs$ is a sorted sequence of $docIDs$.

Both map and reduce operations are performed in parallel in a distributed computer system. The intermediate data obtained by a computer in the mapping stage are

J. Berlińska
Faculty of Mathematics and Computer Science, Adam Mickiewicz University, Umultowska 87, 61-614 Poznań, Poland
E-mail: Joanna.Berlinska@amu.edu.pl

M. Drozdowski
Institute of Computing Science, Poznań University of Technology, Piotrowo 2, 60-965 Poznań, Poland
E-mail: Maciej.Drozdowski@cs.put.poznan.pl

divided into many parts, which are then assigned to the reducing processors by hashing. The obtained partitioning of the intermediate data has a big impact on the whole MapReduce application performance [13–15]. In the ideal case all reducing processors should finish computations at the same time. In reality the data distribution is often skewed and some processors work longer than others. Consequently, the total running time of the application increases. In this work we analyze this problem and present algorithms to solve it. Although practical methods of handling skew in MapReduce have been proposed earlier [13–15], to our best knowledge this is the first theoretical study dedicated to comparing approaches mitigating the skew in different stages of computations.

We propose a mathematical model of MapReduce based on the divisible load theory. Divisible load theory (DLT) was introduced in papers [1,9]. It is a model of distributed processing which assumes that the data to be processed, called load, can be continuously divided into pieces. There are no precedence constraints between these pieces, so that they can be processed independently on remote computers. Divisible load theory covers scheduling problems and performance modeling for various types of interconnection networks [9,11,8], systems with memory limits [16,3,5], and other. Surveys of divisible load theory, including applications, can be found e.g. in [2,7,12,18]. A divisible load model of MapReduce applications has been first proposed in [4,6]. In this work this model is expanded by relaxing the assumptions about the organization of communication between computers. We also provide a more detailed view of the reducing stage.

The rest of this paper is organized as follows. In Section 2 we present a mathematical model of MapReduce computations. The methods of handling partitioning skew in MapReduce applications are described in Section 3. The algorithms are compared experimentally in the following section. The last section is dedicated to conclusions.

2 Mathematical Model of MapReduce

In this section we present the mathematical model of MapReduce computations.

We assume that the application is run by a homogeneous computer system. We will use the terms computer, processor and worker interchangeably. Each computer comprises a CPU, some memory and an independent network interface (e.g. NIC and DMA). A processor can open at most one communication channel at a time, i.e. the so-called one-port model is used. The communication speed for two computers communicating in otherwise unused network is $1/C$. However, the bandwidth limit for concurrent channels in the whole network is l/C, so that at most l communication channels can be opened simultaneously without reducing the communication speed. We will call l the bisection width limit. If $n > l$ communications take place at the same time, then the bandwidth is shared equally between them, and the communication speed decreases to $l/(nC)$.

The total amount of load (input data) to be processed is V (e.g. bytes).

The execution of a MapReduce application begins with starting many copies of the program on a cluster of computers. A master computer assigns work to the other computers (workers). There are m processors executing map tasks (the mappers) and r processors executing reduce tasks (the reducers). The master starts processing a MapReduce application with dividing the input data into so-called splits and assigning them to the mappers. As the number of load splits is very big [10], the granularity of the input load is fine. Hence, according to the methodology of DLT, we assume that the load is arbitrarily divisible. Each mapper obtains the same amount of data.

A mapper reads the assigned input load splits and processes the data using the Map function. We perceive these operations as processing with rate A (e.g. sec/bytes) [4,6]. Each mapper receives load of total size V/m and processes it in time AV/m. The results of mapping are divided into r parts by the partitioning function (usually of the form $hash(key1) \bmod r$) and written to r files on the local disk. Each file is dedicated to a particular reducer. We assume that the amount of results produced by mappers is proportional to the input size. For α bytes of input, $\gamma\alpha$ bytes of output are produced. The information about the locations of the files containing the intermediate results is sent back to the master. We will denote the size of a single piece of information, such as the location of a file, by ε. Thus, the communication of all mappers with the master takes time $Cmr\varepsilon$, as each of the m mappers created r files. The master forwards it to the reducers (again in time $Cmr\varepsilon$, because each of the r reducers obtains the locations of m files).

When a reducer receives the information about the files locations, it reads the buffered data from the local disks of the mappers. In order to guarantee that no computer performs two communications at the same time, a special communication organization is necessary. 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 parts 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 given time, and the total time needed for sending load from the mappers to the reducers is $\max\{C, C\min\{m, r\}/l\}\gamma V/\min\{m, r\}$.

After reading all intermediate data, a reducer sorts it by the intermediate keys in order to group together all occurrences of the same intermediate key. The time needed to sort the data of size x is $a^{sort}x\log x$. Each key and the corresponding set of values are then processed by the Reduce function. The computing rate of a reducer will be denoted by a^{red}. Thus, the reducer execution time for input data of size x is $a^{red}x$. The output generated for each intermediate key is appended to a final output file for a given reducer. Thus, the final output of MapReduce is available in r output files located on the reducers.

In the ideal case, the partitiong function divides the space of intermediate keys so that each reducer obtains from each mapper data of the same size $\gamma V/(mr)$, i.e. the total load received by any reducer is $\gamma V/r$. Then, the running time of the MapReduce application is

$$\begin{aligned} T^* = {} & AV/m + 2Cmr\varepsilon + \max\{C, C\min\{m, r\}/l\}\gamma V/\min\{m, r\} + \\ & a^{sort}(\gamma V/r)\log(\gamma V/r) + a^{red}\gamma V/r. \end{aligned} \tag{1}$$

However, in reality the load partition is not ideally balanced and some reducers receive more data than the others. Such *partitionig skew* may lead to a performance drop, as the execution of the whole application finishes when the last reducer completes computations. The running time of the application depends in this case on many factors and cannot be presented by a simple formula similar to (1).

3 Algorithms

In this section we present two algorithms for mitigating partitioning skew in MapReduce: a static and a dynamic method. The static approach to the problem is to improve the load partitioning before sending the data from the mappers to the reducers. In the dynamic methods, the load distribution between the reducers can only be changed after the reducers obtained the data and started processing.

3.1 Static Method

A static method of load balancing, called fine partitioning, was proposed in [13]. The idea was to change the partitioning function so that it divides the space of the key values not into r parts, but into kr parts for some $k > 1$. This allows to decrease the impact of creating parts of different sizes by appropriately assiging the parts to the reducers. After the mappers finish their computations, they send the information about the obtained part sizes to the master computer. 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 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. The reducers obtain the information about the partitions they are to process, read the data from the mappers and process them.

We will now present a divisible load model of MapReduce with fine partitioning. In the first step, all mappers read and process the load with rate A, what takes time AV/m. Afterwards, each of the mappers sends the information about the sizes of kr partitions it created to the master. As we use 1-port model, the communications with the master must be sequential and the time necessary to complete them is $Ckrm\varepsilon$. The master computes the total sizes of the partitions, sorts them and assigns to the reducers, what 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 assignment is then sent sequentially to the reducers, in time $Ckrm\varepsilon$. Next, the reducers start reading the data from the mappers. After obtaining its load of size $\delta_j\gamma V$ (where $\sum_{j=1}^{r}\delta_j = 1$), reducer j sorts the data and performs reducing, in total time $a^{sort}(\delta_j\gamma V)\log(\delta_j\gamma V) + a^{red}\delta_j\gamma V$.

3.2 Dynamic Method

A system allowing for MapReduce skew mitigation in a dynamic way, called SkewTune, was described in [15]. The goal of SkewTune was to handle different types of skew, in both mapper and reducer stage of computations. As partitioning skew only is the subject of this work, we propose here a simpler approach. While SkewTune tries to balance the load distribution each time when a reducer finishes computations, we stop the computations to redistribute the load only once.

In our algorithm mapping and sending load from the mappers to the reducers proceeds as described in Section 2. Each reducer starts sorting (and then reducing) as soon as it receives all data. It notifies the master when it finishes sorting and when it finishes reducing. As soon as all reducers completed sorting and at least one reducer processed all its load, the master sequentially stops the r_1 reducers which still have some load to process, in time $Cr_1\varepsilon$. Afterwards, each of these reducers informs the master about the amount of its remaining load, in total time $Cr_1\varepsilon$. 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. According to the methodology of DLT, the master assumes that it is possible to divide the load into parts of any sizes, although in practice some rounding may be necessary, as records with the same key must be processed by the same reducer. If balancing the load assignment in this way 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 proportional to the number of analyzed cases. 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 load sizes if necessary. The execution of the application is completed when the last reducer finishes computing.

4 Computational Experiments

In this section we analyze the influence of system parameters on the improvement of MapReduce performance achieved by the proposed algorithms. In all test instances the number of keys in the input data is 1E6. The key frequencies are obtained in the following way. For each key i a number f_i is selected randomly with uniform distribution from interval $(1-\Delta, 1+\Delta)$. The frequency of i is set to $f_i/\sum_j f_j$. Thus, the key frequencies are not very unbalanced, as the maximum possible value is $1+\Delta$ times greater than the average.

Two types of instances will be analyzed. In the easy instances the partitioning function is consctructed by assigning the keys to the reducers randomly in such a way that each reducer receives the same number of keys. Hence, the load partitioning for the reducers will be rather balanced. In the hard instances each reducer obtains the same number of keys, but reducer 1 receives the keys with the greatest frequencies, reducer 2 the remaining keys with the greatest frequencies, and so on. Thus, the load distribution will be unbalanced.

Unless stated otherwise, we assume the following 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 rate parameters in order to emphasize the reducing phase, for which load balancing is crucial. The number of partitions per reducer created by the static algorithm is $k = 10$.

The algorithms performance will be measured by the speedup of the whole MapReduce application with load balancing (dynamic or static) in comparison to executing MapReduce as described in Section 2. Each point on the charts presents the average value over 10 instances.

We will first study the number of processors in the system executing MapReduce application. Fig. 1 presents the influence of the mapper number m on the performance of a system with $r = 100$ reducers. It can be seen that the speedup improves with growing m for both algorithms and both instance sets. This can be explained by the fact that when the number of mappers grows, then the time of mapping decreases.

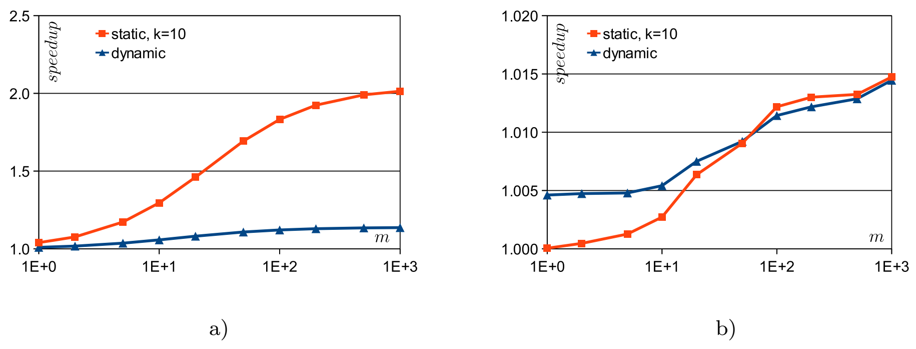


Fig. 1 Speedup vs. m. a) Hard instances, b) easy instances.

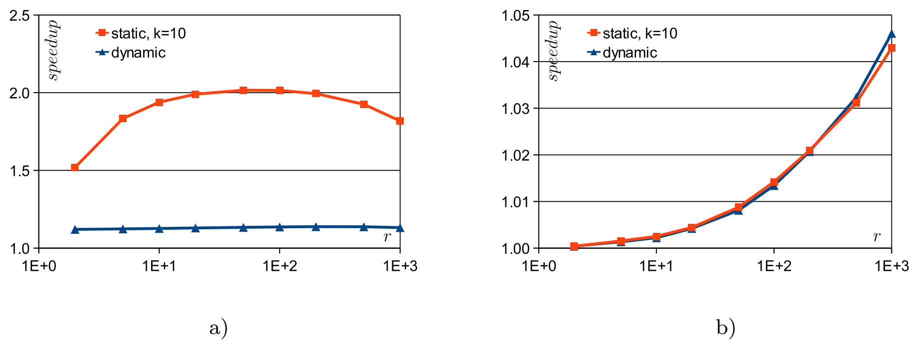
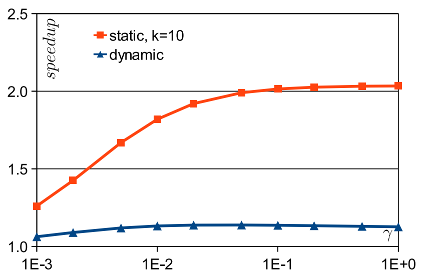


Fig. 2 Speedup vs. r. a) Hard instances, b) easy instances.

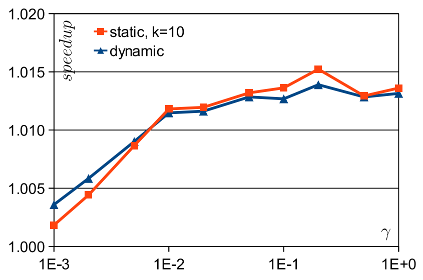
Therefore, changes in the reducing time, which are caused by the load balancing algorithms, have a greater impact on the whole schedule length. The speedup is much larger for hard instances than for the easy ones. This is natural, as in the easy instances the initial load distribution is already balanced. Therefore, there is less room for improvement.

For hard instances the static algorithm performs much better than the dynamic method, because balancing the load dynamically can only start after all reducers finish sorting. If the load distribution is unbalanced, many processors have to wait idle until the last reducer sorts all its load. The situation is different for easy instances, for which the dynamic algorithm performs better for small m. In this case, the amounts of load sent from a mapper to a reducer are big. In consequence, the time between the beginning of computations on the first and on the last reducer is long. Therefore, it is better to create unbalanced load distribution, such that the reducers which start processing earlier obtain more data. This can be done by the dynamic method, but not by the static algorithm. When m gets larger, the reducers start computing around the same time and this effect disappears.

The influence of the reducer number r on the application speedup is shown in Fig. 2. In the instances with $r > 100$ the bisection width limit was increased to $l = r$, so that



a)



b)

Fig. 3 Speedup vs. γ. a) Hard instances, b) easy instances.

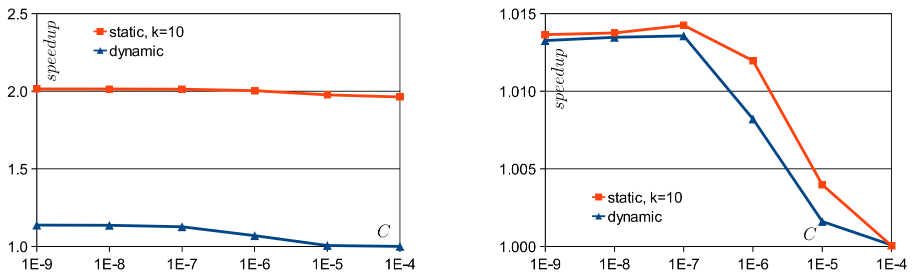
it was not limiting the communication speed. For the hard instances, the performance of both algorithms first increases, and then decreases. When the number of reducers is very small, the load distribution found by the heuristic in the static algorithm is often not very balanced. Hence, the speedup is smaller than for larger reducer numbers. When there are many reducers ($r > 100$), the load obtained by a single reducer, and hence the reducing time, decreases. Thus, we observe decreasing speedup, because the contribution of reducing to the total application running time is smaller. A similar effect can be seen for the dynamic algorithm, but its range is not so significant.

The speedup of both algorithms does not decrease for the easy instances. As the number of keys in the input data is constant, for larger r the number of keys per reducer is smaller. Hence, the load distribution between the reducers tends to be more unbalanced, and hence, the algorithms have better opportunity to fix it. This effect is stronger than the impact of decreasing the reducing time. For hard instances it is not so significant because the load distribution for small r is already skewed.

In Fig. 3 we present the speedup for different values of the multiplicity γ of results created by the mappers. As could be expected, increasing γ leads to better speedup for both algorithms and both instance sets. When γ is larger, the reducers receive more load and the significance of the reducing time, which is shortened by the algorithms, increases.

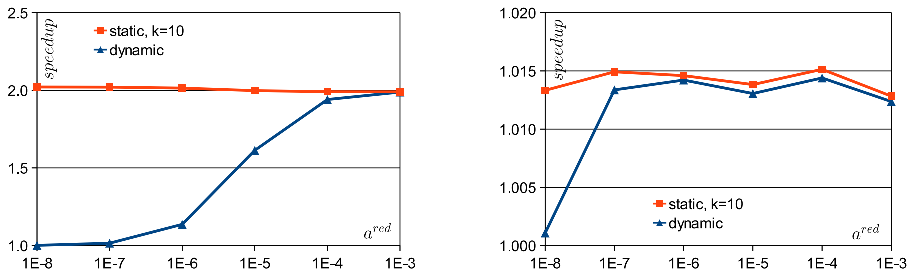
The results of experiments with changing communication rate C are presented in Fig. 4. Note that increasing C 10-fold is roughly equivalent to decreasing l also 10-fold at fixed C. Thus, Fig. 4 demonstrates also the effect of changing l on the performance. When C grows, the speedup for both algorithms decreases. This effect is stronger for the dynamic method, because it performs additional transfers of large amounts of data between the reducers. When C becomes very big, moving load between the reducers is unprofitable and the speedup achieved by the dynamic method decreases to 1. In the static method the only additional communication in comparison to MapReduce without skew handling is sending some information from the mappers to the master and from the master to the reducers. These messages are short and do not contribute much to the application running time. Hence, the results obtained by this method for the hard instances and big C are still good.

The main disadvantage of the dynamic algorithm is that it starts working after the reducers finish sorting data. If sorting dominates in the reducing time, this may



a) b)

Fig. 4 Speedup vs. C. a) Hard instances, b) easy instances.



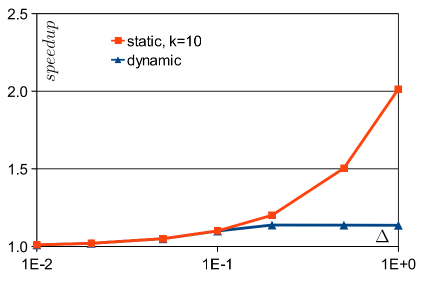
a) b)

Fig. 5 Speedup vs. a^{red}. a) Hard instances, b) easy instances.

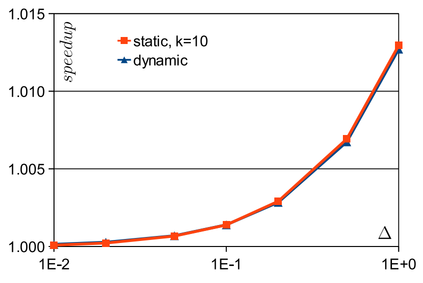
be too late to significantly improve the execution time of the application. In Fig. 5 we present the application speedup for different values of a^{red}. Indeed, the performance of dynamic load balancing improves with growing a^{red} and reaches the same level as the static algorithm, because sorting time becomes less important. However, for the tested parameter range the dynamic method does not outperform the static one.

In the next set of experiments we expose the impact of parameter Δ (cf. Fig. 6). Increasing Δ makes the key frequencies more diversified and the load distribution more distorted. Therefore, both algorithms have more opportunities for introducing improvements and achieve better speedup.

Finally, let us analyze the influence of parameter k on the performance of the static algorithm. On the one hand, big k gives us greater freedom in choosing the load sizes assigned to the reducers. On the other hand, too large value of k means creating many files, sending a lot of information from the mappers to the master and from the master to the reducers, and longer master computation time. The results of the experiments with changing k, for different values of r, are presented in Fig. 7. As before, for $r > 100$ we set $l = r$ in order to avoid influencing the speedup by the communication speed. The lines for $r = 1000$ end with $k =$ 1E3, because for $k =$ 1E4 there would be more partitions than keys in the input data. It can be seen that the speedup for $k = 2$ is

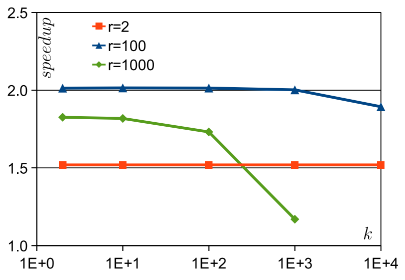


a)

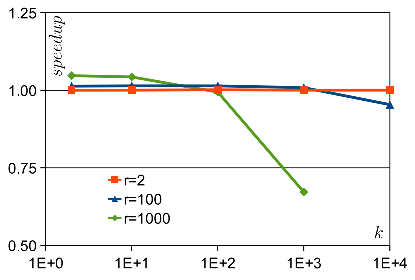


b)

Fig. 6 Speedup vs. Δ. a) Hard instances, b) easy instances.



a)



b)

Fig. 7 Speedup for the static algorithm vs. k. a) Hard instances, b) easy instances.

almost the same as for $k = 10$ (the numerical results show that $k = 10$ is slightly better for $r = 2, 100$). Thus, big values of k are not necessary to obtain good speedup. The effect of increasing k depends on the number of reducers r. When $r = 2$, the number kr of partitions created by the static algorithm is not very big even for $k =$1E4. The time needed for sending the information about the partition sizes to the master and for assigning the partitions to the reducers is short, and hence, the application speedup does not change. When the number of reducers grows, the impact of increasing the number of partitions per reducer becomes more visible. For the easy instances the cost of sending additional messages and assigning the partitions to the reducers is greater than the profit from load balancing.

5 Conclusions

In this work we analyzed the partitioning skew in MapReduce applications. A mathematical model of MapReduce based on divisible load theory was presented. Two methods of mitigating the skew were proposed. Both of them are easy to implement in practice and do not rely on any specific assumptions about the executed application. The static algorithm creates many data partitions on the mappers and then assigns them to the reducers so that the load distribution becomes balanced. The dynamic algorithm transfers parts of data between the reducers, but only after sorting the intermediate data.

Both algorithms were tested in a series of computational experiments. It was shown that mitigating the partitioning skew is an important issue, as for many cases it allows for decreasing the running time of MapReduce more than twice. For most of the test instances the static algorithm performed better than the dynamic method. The disadvantage of the dynamic algorithm is that it has to wait with load balancing until all reducers finish sorting. If the time needed for reducing is not greater than the time of sorting, then sorting unbalanced data precludes obtaining good application execution time. However, there are situations in which the dynamic algorithm achieves significantly better results. Namely, for a system with a particular combination of parameters it may be necessary to divide the load between the reducers unequally in order to achieve the best performance. The dynamic algorithm can handle this situation, because it refers to the running time of the reducers, whereas the static algorithm operates on data sizes and therefore cannot construct good load partitioning in this case.

We also showed that the influence of skew mitigating depends on a combination of the system parameters. The effects of load balancing are most significant when the time of reducing dominates in the total running time of the application.

Acknowledgements This research has been partially supported by Polish National Science Centre grant N N519 643340.

References

1. R. Agrawal, H.V. Jagadish, Partitioning Techniques for Large-Grained Parallelism, IEEE Transactions on Computers 37, 1627-1634 (1988)
2. O. Beaumont, H. Casanova, A. Legrand, Y. Robert, Y. Yang, Scheduling divisible loads on star and tree networks: results and open problems, IEEE Transactions on Parallel and Distributed Systems 16, 207-218 (2005)
3. J. Berlińska, M. Drozdowski, M. Lawenda, Experimental study of scheduling with memory constraints using hybrid methods, Journal of Computational and Applied Mathematics 232, 638-654 (2009)
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, Heuristics for multi-round divisible loads scheduling with limited memory, Parallel Computing 36, 199-211 (2010)
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. J. Błażewicz, M. Drozdowski, Scheduling divisible jobs on hypercubes, Parallel Computing 21, 1945-1956 (1995)
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. 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)
15. 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)
16. 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)
17. J. Lin, C. Dyer, Data-Intensive Text Processing with MapReduce. Morgan & Claypool (2010)
18. T.Robertazzi, Ten reasons to use divisible load theory, IEEE Computer 36, 63-68 (2003)