MALiBU: Metaheuristics Approach for Online Load Balancing in MapReduce with Skewed Data Input

Matheus H. M. Pericini[1], Lucas G. M. Leite[1] and Javam C. Machado[1]

[1]LSBD – Departamento de Computação – Universidade Federal do Ceará (UFC)
Campus do Pici – Bloco 952 – 60.020-181 – Fortaleza – CE – Brazil

{matheus.pericini,lucas.goncalves,javam.machado}@lsbd.ufc.br

Abstract. *MapReduce is a parallel computing model where a large dataset is split into smaller parts and executed on multiple machines. When data are not uniformly distributed, we have the so called partitioning skew, where the allocation of tasks to machines becomes unbalanced, either by the distribution function splitting the dataset unevenly or because a part of the data is more complex and requires greater computing effort. To solve this problem, we propose a function based on Simulated Annealing metaheuristic which finds a partitioning that results in a better load balancing.*

1. Introduction

As technologies related to computational clouds, sensors, and scientific computing advance, larger volumes of data are generated out of those applications, demanding efficient techniques to store and analyze such data in order to extract relevant information from them. To meet this need, the MapReduce model was created as a parallel computing model that distributes a large dataset to run on multiple commodity machines, avoiding the need of using machines with high processing power, making processing more robust and scalable. The standard structure of MapReduce is divided in two steps: Map and Reduce. In the Map phase, the master node takes an input, splits it into smaller subproblems, and distributes it to worker nodes. A worker node processes its subproblem creating intermediate data, stores that data and signals the master node. In the Reduce phase, the master node distributes the intermediate data to the workers according to a partition rule. The workers collect the intermediate data, combine and process them to form an output to the original problem. This divide-and-conquer process allows MapReduce to process large data sets on distributed clusters. A MapReduce user only needs to write the map and reduce functions, which effectively hides the operation of large clusters, providing a highly scalable and fault-tolerant solution for large data-processing applications. Many companies, such as Google, Amazon, Facebook, and Yahoo! have been using MapReduce to process large volumes of data [Zhihong Liu 2016].

Although MapReduce has several advantages, its simplicity may imply in difficulties when processing specific applications or specific data. In most studies involving MapReduce, it is considered that the data to be processed are balanced, so that its distribution between the worker nodes using a partitioning function based on its hash code is balanced in an acceptable way. However, in many real-world applications, data split between worker machines is not uniformly distributed and the partitioning function based on hash can not guarantee that the data allocation to the worker nodes will be balanced. This phenomenon is called Partitioning Skew, which is a problem that occurs when the distribution of the data among machines is unbalanced or a part of the data needs more computational power to execute. As a result, in the reduce step, some of the workers become overloaded while others become idle.

The load balancing problem with skewed data has been investigated in several recent works [Ramakrishnan et al. 2012, Gufler et al. 2011, Kwon 2012]. Most approaches treat the problem offline, that is, wait for the mapping phase to complete to obtain the statistics related to the generated intermediate data, or perform a preprocessing on a sample of the data prior to the map phase to estimate the distribution of the data and execute an appropriate allocation strategy. However, these approaches lead to a network overhead and I/O excessive costs. Another strategy is to reallocate the data that requires more computational power to machines with more resources [Kwon 2012]. Although this strategy improves the performance of large data, it generates a large overhead, mainly when one considers small amount of data.

In this article, we balance the data by replacing the default MapReduce partitioning function with a function that also considers both the load already allocated to each machine and the size of the data that is waiting for allocation. Our strategy is to minimize a function that measures the imbalance of the load allocated to each machine. To find a partition of the data that approaches the ideal balancing, we will use a strategy based on metaheuristics because we want to solve this problem in an online manner without prior knowledge about the behavior of the data and it would also be computationally expensive to explore the entire solution space. Our strategy, called *MALiBU*, is based on Simulated Annealing metaheuristic, which is able to find solutions close to the global optimum while having relatively small complexity.

The remainder of this paper is organized as follows. Section 2 will give a more detailed view of the MapReduce model and the partitioning skew problem, motivating this work. Section 3 describes some of the related works in the literature. In Section 4 our contribution is presented, detailing MALiBU the partitioning algorithms used. Finally, Section 5 will show the results of the experimental evaluation and Section 6 will present our conclusions on this work.

2. MapReduce and Partitioning Skew

MapReduce splits dataset into smaller parts in order to process them on multiple machines [Dean and Ghemawat 2008]. The results of processing these parts are aggregated in order to generate the solution to the original dataset. In MapReduce the map tasks are responsible for receiving parts of the dataset, applying a user-defined mapping function and returning a set of key-value pairs, called intermediate data. The intermediate data are then distributed to the reduce tasks according to a partitioning function. A reduce task receives a partition and performs a reduce function to generate the solution for the intermediate data in this partition. MapReduce is able to significantly reduce the processing time of a large amount of data because it divides these data into smaller parts and processes them in parallel across multiple machines.

The MapReduce model was written in different programming languages [Isard et al. 2007, He et al. 2008, Ranger et al. 2007]. For example, Apache Hadoop [Hadoop 2006], one of its most popular free implementations, is based on a master-worker architecture, where a master node makes scheduling decisions and multiple worker nodes run tasks dispatched from the master.

Figure 1 shows the architecture of the Hadoop MapReduce model.

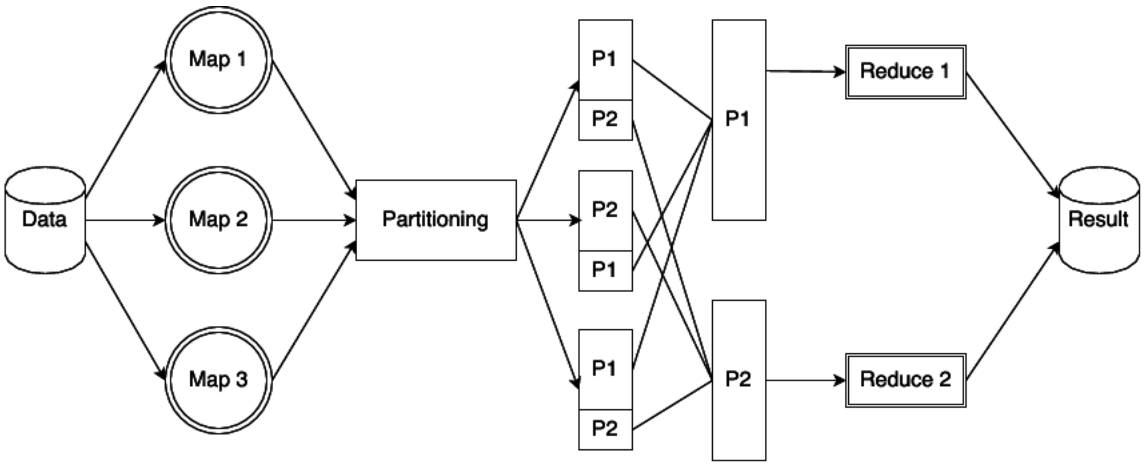

Figure 1. Hadoop MapReduce architecture

2.1. Partitioning Skew

Currently, in MapReduce implementations, data are allocated to the reduce nodes according to the function $Hash(HashCode\ mod\ number\ of\ reducers)$. Usually this function allocates tasks so that the partitions get balanced. When the size of the key-value pairs are not uniformly distributed, the allocation of tasks becomes unbalanced. This type of situation characterizes partitioning skew, in which some reduce nodes receive high loads, increasing the time of response of assigned tasks, and others receive low loads, being idle because they have more resources allocated than necessary. Many authors studied the partitioning skew in mapreduce applications [Kwon et al. 2011, Okcan and Riedewald 2011, Ibrahim et al. 2010, Atta et al. 2011].

2.1.1. Skewed Key Frequencies

For a number of applications, some keys occur more frequently in the intermediate data. For example, in word count some words appear more frequently than others, such as definite articles. In this type of application, the default Hadoop partitioning function will allocate tasks overloading the reduce nodes that receive most keys.

In the Figure 2, the partitioning skew generated by skewed key frequencies is illustrated. Notice that the K1 partition has got more intermediate data allocated than the others. If the key-value pairs (k1, v1) and (k2, v2) were respectively allocated to partitions K2 and K3, the partitions would be balanced.

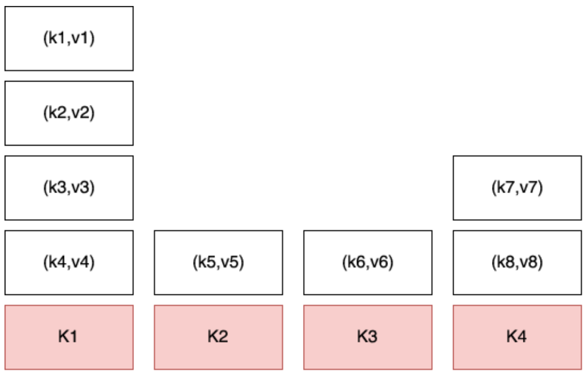

Figure 2. Skewed key frequencies example

2.1.2. Skewed Key Size

In applications where value sizes in key-value pairs vary significantly, the distribution of the tasks to reduce nodes may become unbalanced. Thus, although the number of different values for each key is approximately the same, the execution time may vary between one key and another, resulting in some idle nodes while others are overloaded.

In the Figure 3, the partitioning skew generated by skewed key sizes is shown. Although only one key value data is allocated to partition K3, this data takes longer to execute than the two data allocated to partition K2. In this case, for the nodes to be idle for the shortest possible interval, it is necessary that the data whose value is v4, allocated to partition K1 be allocated to partition K2 or K4.

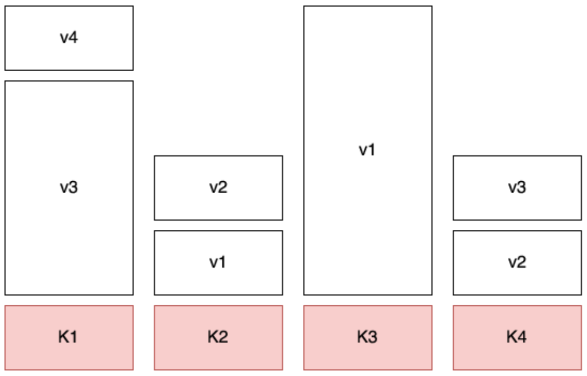

Figure 3. Skewed key size example

2.1.3. Skewed Execution Times

Some tasks may require more computing power than others. An example where this can occur is in the page classification algorithm, that classifies a page according to the links to other pages, which we call neighboring pages. The greater the number of neighbors pointing to the same page, the longer it will take to process it. In Figure 4, 128 machines processes a set of pages. During execution a machine takes about five times more than the average of all others.

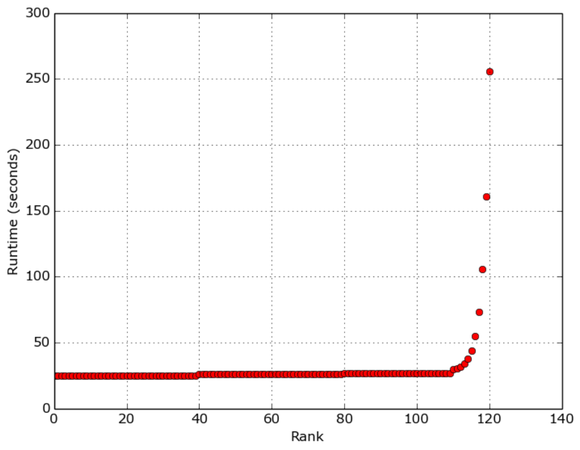

Figure 4. Skewed execution times example

3. Related Work

MALiBU focuses on the partitioning skew caused by skewed key frequency and skewed key size, since these types of skew are the major factors impacting job execution time. Recent works [Gufler et al. 2012, Ramakrishnan et al. 2012] and [Zaheilas and Kalogeraki 2014] show that solving the partitioning skew is not a trivial task. Many other works proposed solutions for the partitioning skew [Rosen and Zhao 2012], we detail the most important ones in this section.

3.1. OPTIMA: On-Line Partitioning Skew Mitigation for MapReduce with Resource Adjustment

OPTIMA [Zhihong Liu 2016] is a resource allocation framework based on the detection of overloaded reduce tasks due to partitioning skew. The approach sends information about the allocation of the intermediate data to the controller machine, which uses it to predict the size of tasks and identify possible overloaded tasks. With this information, OPTIMA allocates resources for overloaded machines up to the maximum machine load limit. OPTIMA implements five main components. During the execution of the map tasks, the Partition Size Monitor sends the size of the intermediate data produced to the master machine through a modification in the reporter responsible for updating the master. With the information gathered from the map tasks, Partition Size Predictor uses a linear regression to predict the size of the partitions in an online manner, so that Overloaded Task Detector be able to identify the reduce jobs that are overloaded and tells the resource allocator that these tasks will need more performance. Lastly, the Fine-grained Container Scheduler increases the amount of memory and CPU used by machines that need higher performance to the maximum amount available on the machine based on information about overloaded tasks. Despite the decrease of a job execution time, OPTIMA is not able to prevent a machine from draining its resources and become overloaded because it continues to use the standard partitioning function.

3.2. SkewTune: Mitigating Skew in MapReduce Applications

Skewtune [Kwon 2012] monitors jobs in order to identify which ones are responsible for overloading reduce nodes. With this information, it balances the utilization of the cluster nodes by repartitioning these jobs, allocating them to other reducers with less load. The process consists in verifying if a job that is causing the overhead is at a point of its execution in which it is advantageous to divide its data with a reducer that is underloaded. This division will only take place if the time needed to divide the data plus the estimated time remaining for the split job is less than the remaining time of the complete job. This process is repeated until all tasks are completed or there is no task that can be divided. At the end of the execution, the data of the same task that has been divided are grouped in the same machine, in order to keep the result consistency of MapReduce. Although it works well for complex tasks, overhead caused by the analysis and division of a small task generates an additional cost that ends up negatively impacting the execution time.

3.3. Online Load Balancing for MapReduce with Skewed Data Input

The work described in [Yanfang Le 2014] presents two key allocating algorithms for the partitioning phase. The first is a greedy online algorithm that acts on the partitioner distributing the keys according to the size of the existing partitions giving priority to the smaller ones. In the second algorithm, it performs the distribution of the keys based on a preprocessing over the statistics of a priori data. Both algorithms ignore that the key distribution can vary significantly during a mapping phase, causing an imbalance in the allocation of the processed keys, failing to avoid partitioning skew in a significant way.

The approaches described as related work show different ways of dealing with partitioning skew. In the next section we present MALiBU, a meta-heuristic approach to balance the load allocated to the reduce nodes, avoiding the overload of these nodes without generating a large overhead in the reduce phase.

4. The MALiBU Method

MALiBU balances the load received by the reduce nodes following a combinatorial optimization problem solution. We assume that the input of this problem is the generated intermediate data set and the output is a partitioning of the keys of this set into a number of reducers. Because it is an online solution involving a large amount of data, we apply metaheuristic approaches to find a sufficiently good solution. In combinatorial optimization, by searching over a set of feasible solutions, metaheuristics can often find good solutions with less computational effort than optimization algorithms, iterative methods, or simple heuristics [Blum and Roli 2003]. Metaheuristics use a sample of the solution set, which is too large to be completely covered. MALiBU takes advantage of the Simulated Annealing metaheuristic to partition the keys to reduce tasks. In a metaheuristic, we look for a solution from small variations of a candidate solution, which can be generated in a random way. This variation is compared with the candidate solution for evaluation according to some criterion of acceptance. The best evaluated solution is maintained and the process is repeated until a stopping criterion is satisfied. In our approach, a candidate solution is a variation of a partitioning which we will call its neighboring partition. A partition A is a neighbor of a partition B if they differ by the allocation of exactly one key. To facilitate the understanding of the metaheuristics, we define below the formal concept of neighboring partitions and detail later the operation of Simulated Annealing in the partitioning phase of MapReduce.

Definition 1 (Neighboring partitions)**.** Let $K = \{k_1, k_2, \ldots, k_n\}$ be a set of keys, $R = \{r_1, r_2, \ldots, r_m\}$ a set of reducers and $P_1 = \{p_1, p_2, \ldots, p_m\}$, $P_2 = \{p_1, p_2, \ldots, p_m\}$ two partitions from K to R. We say that P_1 is a neighbour of P_2 if and only if just one of the following conditions is satisfied:

- $k_i \in p_x \in P_1$, $k_j \in p_y \in P_1$, $k_j \in p_x \in P_2$ and $k_i \in p_y \in P_2$, with $1 \leq i < j \leq n$ and $1 \leq x < y \leq m$ for only one pair of keys k_i and k_j. The partitioning is the same for the remaining keys in P_1 and P_2
- $k_i \in p_x \in P_1$ and $k_i \in p_y \in P_2$, with $1 \leq i \leq n$ and $1 \leq x < y \leq m$ for only one key k_i. The partitioning is the same for the remaining keys in P_1 and P_2

4.1. Simulated Annealing

In metallurgy, annealing is a process for hardening metals or glass by heating and cooling them gradually, allowing the material to reach a low-energy crystalline state [Russell et al. 2003]. The Simulated Annealing (SA) can be seen as the process of making a ping-pong ball reach the bottom of an irregular surface. If we just let the ball roll, it can get stuck at a point other than the bottom, because the surface is uneven. So that the ball can leave the current point and continue rolling, it is necessary for someone to shake the surface.

In our problem, we want to attenuate the imbalance of the intermediate data allocated to the reducers. The solution based on SA is done as follows. An initial temperature and a cooling factor are defined. According to the intermediate data given as input, a key partitioning is created using the standard Hadoop partitioning function. We call this partitioning *current partitioning*. From this, we compare the current with a randomly generated neighbor partitioning. If the neighboring partitioning better balances the intermediate data allocated to each reducer, it becomes the current partitioning. Otherwise, the neighboring partitioning will become the current partitioning with a probability less than 1. This probability decreases proportionally to the current temperature. Finally, the current temperature is updated according to the cooling factor. This process is repeated until the temperature reaches 0. To facilitate understanding, we will show the pseudocode of this procedure.

In the Algorithm 1, we aim to reduce the imbalance, which is the load of the reducer with the highest volume of data allocated. That is, in order to balance the load, we must partition the keys between the reducers so that the allocated load of the machine with the highest allocation is minimal.

Our implementation consists of integrating Simulated Annealing to Hadoop, using it as the partitioning function. The partitioning phase works as follows: as the dataset is processed and the intermediate data is generated, Hadoop accumulates a portion of that data and Simulated Annealing is executed to find an allocation that minimizes the unbalance of the reducers. This process will repeat itself until the end of the map phase.

5. Experimental Results

In this section we compare the performance of MALiBU against the standard partitioning function of Hadoop YARN 2.7.2. For the implementation of the described algorithms, we modified the Hadoop YARN 2.7.2 by adding the partitioning functions and bypassing the native partitioning function.

Algorithm 1 Simulated Annealing

```
1: procedure PARTITIONER($D$,$T$,$C$,$R$)
2:     $current \leftarrow$ partitioning of the keys of $D$ using $Hash(HashCode(D)\ mod\ |R|)$.
3:     $t \leftarrow T$
4:     $c \leftarrow C$
5:     while $T \neq 0$ do
6:         $neighbor \leftarrow generateNeighbor(current)$
7:         $b_1 \leftarrow unbalance(current)$
8:         $b_2 \leftarrow unbalance(neighbor)$
9:         if $b_2 < b_1$ then
10:            $current \leftarrow neighbor$
11:        else if $random(0,1) < e^{\frac{b_1 - b_2}{t}}$ then
12:            $current \leftarrow neighbor$
13:        $t = t - c$
14: return $current$
```

Our experiments were done using Hadoop YARN 2.7.2 with a master node and nine worker nodes with 8GB of RAM, 4 virtualized CPUs and 80GB of hard drive. All virtual machines are running in the OpenStack cloud environment hosted at the LSBD/UFC. We executed a WordCount application over a real dataset made up of book reviews from the Amazon store [McAuley et al. 2015b, McAuley et al. 2015a]. This dataset has an approximate size of 500MB.

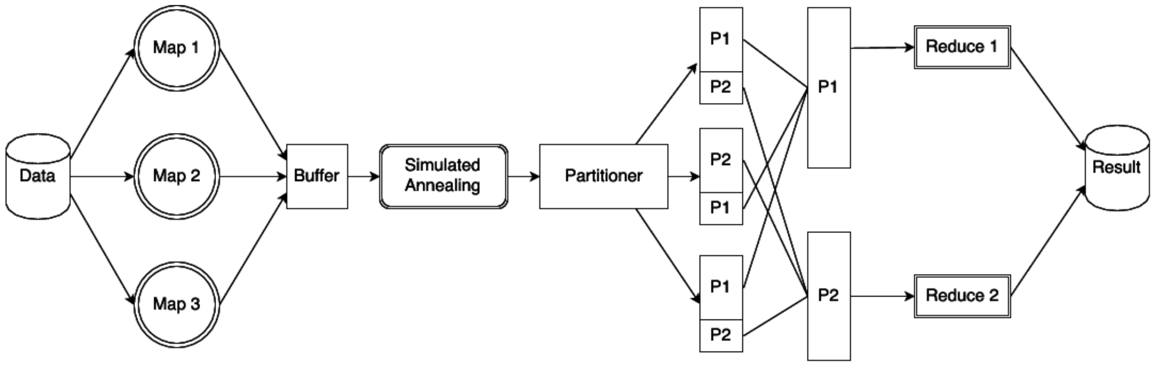

Figure 5. MALiBU architecture

The MALiBU execution occurs as follows: the first $X\%$ of the intermediate data generated in the map phase are accumulated in a buffer. Once the processing is done at $X\%$ percent, this data is partitioned according to Simulated Annealing (Figure 5), which is set to an initial temperature of Y and a cooling factor of 1. After that, each arriving key is either allocated according to the Simulated Annealing result or it is regularly allocated if it's a new key. This process repeats until all the intermediate data is generated and allocated to its due partitions, which in turn, will be executed by the reduce nodes of the system. For our experiments, we consider $X \in \{5, 10, 15, 20\}$ and $Y \in \{5000, 10000, 20000, 30000\}$. The complete MALiBU architecture can be seen in Figure 5.

Next, we check the performance of the native Hadoop partitioning functions and

MALiBU for different scenarios. To measure the performance, we calculate the size of the partition that has the largest amount of data allocated (Y axis) taking into account the progress of the data map phase (X axis).

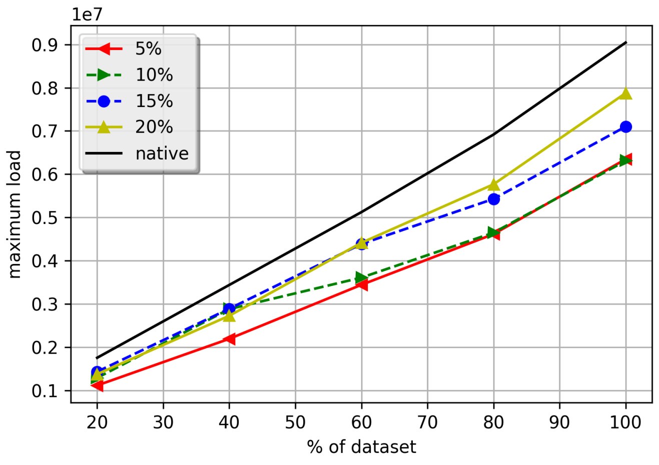

Figure 6. Comparison of several scenarios of data accumulation of the proposed method using 5000 of temperature with the native Hadoop partitioning function.

In our first scenario, as seen in Figure 6, we set a temperature of 5,000 and we varied the amount of data accumulated. Our results showed an improvement in the data balancing in relation to the native Hadoop solution, which is represented by the black line in Figure 6. As expected, the solutions generated by approaches that accumulate less data achieved better results, since the temperature of 5,000 is small in relation to the amount of accumulated data.

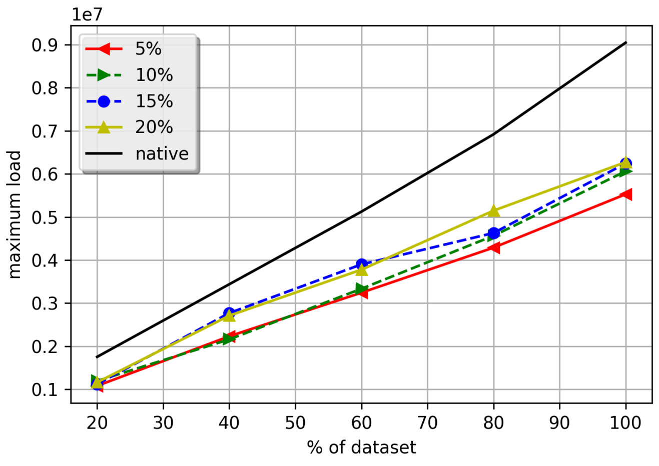

Figure 7. Comparison of several scenarios of data accumulation of the proposed method using 10,000 of temperature with the native Hadoop partitioning function.

In the next scenario, as seen in Figure 7, we set a temperature of 10,000 and we varied the amount of data accumulated. Considering the previous scenario, we have

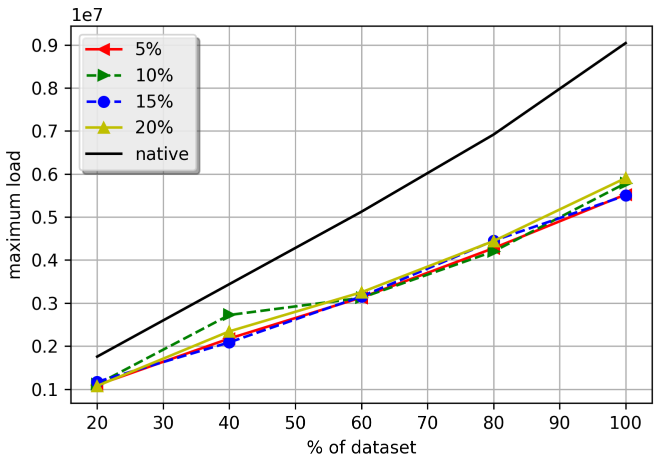

Figure 8. Comparison of several scenarios of data accumulation of the proposed method using 20000 of temperature with the native Hadoop partitioning function.

achieved an improvement in all data accumulation approaches, showing that the increase in temperature reflects in fact a better allocation of the data.

In the next scenario, described in Figure 8, we set a temperature of 20,000 and we varied the amount of data accumulated. Our results showed a slight drop in the maximum load allocated to a reduce node in relation to the temperature set at 10,000. In addition, we notice that as data accumulation approaches show a small difference between them, we can say that the temperature is generating a result close to optimal in relation to the accumulated data quantities.

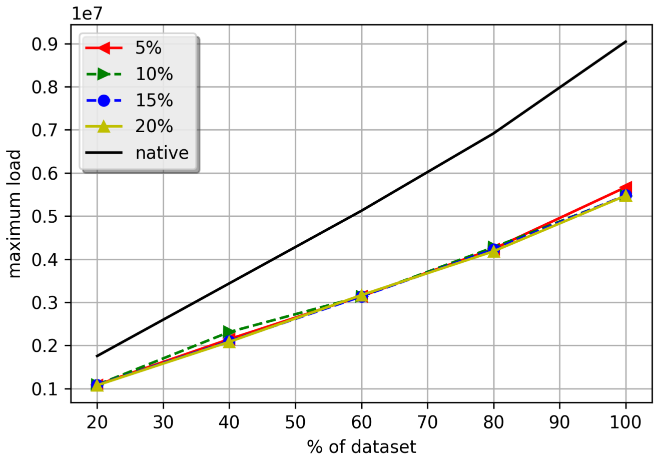

Figure 9. Comparison of several scenarios of data accumulation of the proposed method using 30000 of temperature with the native Hadoop partitioning function.

In our last scenario (Figure 9), we set the temperature of 30,000 and we varied the amount of data accumulated. The results showed a slight improvement in relation to the results presented in the previous scenario, where we set the temperature at 20,000. This result indicates a convergence, so that more temperature or more accumulated data will not be able to generate a better distribution, due to the fact that the data allocation is already very close to the optimal allocation for the dataset used.

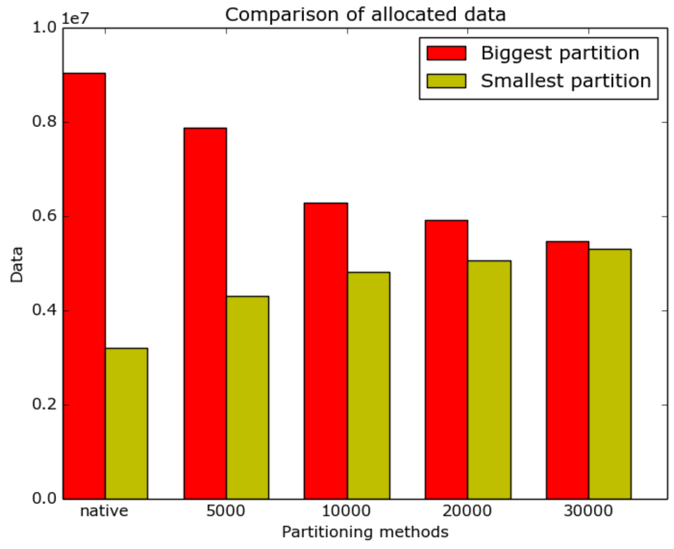

Figure 10. Comparison of the load allocated to smaller and larger partition in different scenarios.

In our experiments, as seen in Figure 10, each time MALiBU accumulates 20% of data, it returns partitions more balanced than all the other scenarios studied. It is worth mentioning that the comparative approaches have been used against data that results in partitioning skew, which is the type of dataset that this work proposes to deal with.

6. Conclusion

In this paper, we investigated the performance of the MapReduce model in a scenario where partitioning skew occurs. This scenario motivated us to mature a balancing approach of the data allocated to the reducers. By verifying the points in which the standard MapReduce could be modified so that partitioning skew was attenuated, it was noticed that altering the partitioning function would be beneficial for the allocation of the data to the reducers, improving data balancing among nodes.

We developed MALiBU, a partitioning method that uses the Simulated Annealing metaheuristics to minimize the imbalance of data on the machines involved in MapReduce. The use of a metaheuristic is interesting because it prevents the space of solutions to be completely covered, focusing on the most promising solutions. We proposed our approach using Simulated Annealing because it is a metaheuristic that is able to converge to a global optimal as long as the chosen temperature matches the complexity of the function to be optimized. Hill-Climbing-based metaheuristics may cling to local optima and may return unsatisfactory results. In addition, Simulated Annealing has a relatively low computational cost, favoring its use in an online algorithm. In our experiments using a 10-node cluster running an actual workload, we were able to improve data balancing performance reducing by 40% the size of the partition that has the largest amount of data allocated relative to the native Hadoop. When compared to other partitioning methods in Hadoop, MALiBU is able to efficiently allocate the data to the reduce nodes. The counterpoint is that it becomes necessary to have sufficient computational resource available to use the temperature and data accumulation parameters for the dataset which the MapReduce application is intended.

Other authors have developed extensions to Hadoop. Among them, we can highlight [Yanfang Le 2014], which presented greedy partitioning method, which allocates the data to the partition with the least load. Although this solution finds good results with balanced data, it is not able to predict the evolution of the keys and, in this way, it can allocate two heavy keys in the same partition, compromising the load balancing in the reduction nodes. MALiBU circumvents this problem with a data accumulation strategy, similar to the sample-based approach described in [Yanfang Le 2014], running Simulated Annealing in order to analyze the evolution of each key size in a more efficient manner.

As future works, we intend to improve the key size prediction using machine learning methods, in order to reduce the amount of data accumulated and the temperature. With this, we can decrease the execution time of the method, making it closer to the execution time of the native Hadoop partitioning function. Another possibility of contribution is to use a heuristic that is able to find the ideal temperature for the execution of Simulated Annealing according to the amount of accumulated data, once the temperatures used in this work were chosen by experimentation. In this way, the system could avoid the absence or the wastage of resources when partitioning the accumulated data. In addition, we want to test the behavior of other metaheuristics, such as Local Beam Search and Stochastic Beam Search, in order to verify the difference in data allocation, memory consumption and runtime. Finally, we intend to study the partitioning skew generated by different execution times of the data. This type of skew comes from a different source and also requires a different kind of solution.

Acknowledgements

This research was partially supported by Lenovo/LIC, Capes/Brazil and LSBD/UFC.

References

Atta, F., Viglas, S. D., and Niazi, S. (2011). Sand join—a skew handling join algorithm for google's mapreduce framework. In *Multitopic Conference (INMIC), 2011 IEEE 14th International*, pages 170–175. IEEE.

Blum, C. and Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Gufler, B., Augsten, N., Reiser, A., and Kemper, A. (2011). Handing data skew in mapreduce. In *Proceedings of the 1st International Conference on Cloud Computing and Services Science*, volume 146, pages 574–583.

Gufler, B., Augsten, N., Reiser, A., and Kemper, A. (2012). Load balancing in mapreduce based on scalable cardinality estimates. In *2012 IEEE 28th International Conference on Data Engineering*, pages 522–533. IEEE.

Hadoop (2006). [Online; accessed in 12-6-2016].

He, B., Fang, W., Luo, Q., Govindaraju, N. K., and Wang, T. (2008). Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM.

Ibrahim, S., Jin, H., Lu, L., Wu, S., He, B., and Qi, L. (2010). Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 17–24. IEEE.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007). Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM.

Kwon, Y., B. M. H. B. R. J. (2012). Skewtune: mitigating skew in mapreduce applications. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36.

Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. (2011). A study of skew in mapreduce applications. *Open Cirrus Summit*.

McAuley, J., Pandey, R., and Leskovec, J. (2015a). Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM.

McAuley, J., Targett, C., Shi, Q., and van den Hengel, A. (2015b). Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 43–52. ACM.

Okcan, A. and Riedewald, M. (2011). Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 949–960. ACM.

Ramakrishnan, S. R., Swart, G., and Urmanov, A. (2012). Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 16. ACM.

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. (2007). Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Ieee.

Rosen, J. and Zhao, B. (2012). Fine-grained micro-tasks for mapreduce skew-handling. *White Paper, University of Berkeley*.

Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M., and Edwards, D. D. (2003). *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River.

Yanfang Le, Jiangchuan Liu, F. E. D. W. (2014). Online load balancing for mapreduce with skewed data input. *IEEE Conference on Computer Communications*.

Zaheilas, N. and Kalogeraki, V. (2014). Real-time scheduling of skewed mapreduce jobs in heterogeneous environments. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 189–200.

Zhihong Liu, Qi Zhang, R. B. Y. L. B. W. (2016). Optima: On-line partitioning skew mitigation for mapreduce with resource adjustment. *Journal of Network and Systems Management*.