Journal of Theoretical and Applied Information Technology
30th November 2020. Vol.98. No 22
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

PARTITIONED GLOBAL ADDRESS SPACE APPROACH FOR THE MAPREDUCE IMPLEMENTATION OF THE PARALLEL KMEANS ALGORITHM

[1]MANSUROVA MADINA, [2]SHOMANOV ADAY

[1]Al-Farabi Kazakh National University, Department of Computer Science, Kazakhstan

[2]Nazarbayev University, Department of Computer Science, Kazakhtan

E-mail: [1]madina.mansurova@gmail.com, [2]adai.shomanov@nu.edu.kz

ABSTRACT

In recent years there is a growing challenge in processing large amounts of data as the size of the data gets exponentially increasing. Mapreduce became an advanced tool to tackle these problems with processing of large arrays of data. As a result, many current Mapreduce frameworks such as Apache Hadoop, Apache Spark rely on Mapreduce as a backbone technology to solve their large-scale problems. Though, such approaches have their benefits, performance wise they cannot always guarantee a linear speed-up and hence a new parallel methods and frameworks needs a thorough study in order to understand scalability and performance benefits in these cases. In this work we present Kmeans parallel (||) clustering algorithm implemented in a partitioned address space Mapreduce system. This work includes a comparison and performance analysis of the presented implementation. In the paper we propose a novel approach that was not considered in literature before. In particular, it was found that our Mapreduce implementation of Kmeans parallel algorithm achieves a strongly linear speed-up that makes this approach an excellent candidate to solve high-dimensional and large-scale clustering problems.

Keywords: *Mapreduce, PGAS, UPC, K-Means, Clustering*

1. INTRODUCTION

In the recent years, we are witnessing a tremendous increase in the volume of the data, generated from various heterogeneous sources of information. In the current age, the field of data processing needs to tackle the problem of dealing with ever larger datasets. The issue lies in the very limited power of a single desktop machine to handle the vast amount of data due to limitation of disk space, memory capacity and CPU processing power. As a solution, a variety of approaches have been proposed to split the work into manageable pieces and hence make large tasks feasible to be solved. In that way, we can make each sub-task to be processed independently from another across a large number of processing units such as threads or even separate cluster machines. Among these approaches, we can put an emphasis on several most popular and efficient tools that have been widely used in different domains.

Mapreduce is a popular parallel programming model for processing and generating large datasets [1]. It has been adopted for solving a variety of large-scale problems, such as genomic data processing [2-3], large-scale graph analysis [4], natural language processing [5], clustering [6]. The popularity of Mapreduce is explained by a simple computational model, that encapsulates details of the parallelization and offers fault-tolerance. Many frameworks that rely on the Mapreduce model have been introduced. The most prominent frameworks such as Apache Hadoop [7], Apache Spark [8] widely used today by many leading companies to store and process large volumes of information.

Clustering can be seen as a non-supervised learning approach to find similarity groups inside a dataset. There exist 4 different categories of algorithms for solving a clustering problem: connectivity, centroid, distribution and density based approaches. Centroid-based clustering works by smoothly moving cluster centers from some initial position to a position where closeness of points within a single cluster is minimized. Formally, the centroid-based clustering problem can be defined as minimization problem (see Eq. 1), where the goal lies in finding a particular assignment of points to cluster centers such that the given assignment minimizes within-cluster squared differences among points.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

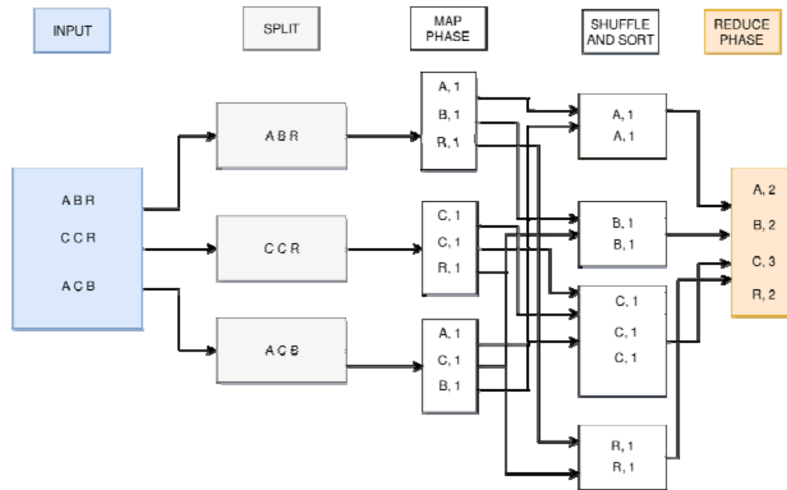

Figure 1: The scheme of the PGAS Mapreduce architecture

$$cost = argmin_{c} \sum_{i=1}^{K} \sum_{j=1}^{count} |x_j - c_i| \quad (1)$$

Centroid-based clustering is particularly suitable for parallel implementation due to the inherently independent update procedure of cluster centers in each phase of the algorithm. Therefore, each cluster can be assigned a single process or a thread that will independently perform a commonly defined function of re-clustering of data points at each iteration step.

In terms of the Mapreduce model, the problem can be decomposed into two or more stages [1]. In the first map stage data is divided into several chunks and distributed among participating threads. The result of the map phase is represented by a set of key-value pairs, where each key corresponds to some aggregate feature that can be extracted from the given input and the value denotes associated with that key property. For instance, in the word count problem the paragraph of the text can be decomposed into single word tokens, where each unique token is a key. In turn, each word token receives an assigned value of how many times that word occurred in that particular paragraph of text. Secondly, the reduce stage is used to perform a group or aggregate operation for each key separately. In order to obtain an assignment between distributed parallel threads and keys, it is necessary to perform an intermediate shuffle procedure. Operations of moving and copying the key-value pairs from a memory of one thread to the memory of another thread, comes with a set of problems for the runtime environment. For instance, in the shuffle phase, it is required to do a lot of movement operations, which incur a lot of network traffic or thread-to-thread memory operations. To overcome that some research work points to developing optimization routines that can efficiently distribute the workload among the threads and improve traffic issues.

Mapreduce consists of three main stages:

1. map stage.
2. sort and shuffle.
3. reduce stage.

The problem of handling large datasets using distributed memory or shared memory approaches lies in limitations naturally inherent in message passing models and shared memory parallel programming models.

Sending and receiving large messages results in huge network latency and underutilized CPU clock cycles. Essentially, these approaches are limited to either using a high-performance distributed file system or developing other workarounds that could potentially address the aforementioned scalability issues. Distributed file systems alone do not provide a local solution and tend to degrade performance when performing large-scale data processing tasks.

The local processing property is a very important part of Mapreduce's performance compared to other parallel programming models in parallel and intensive applications. The shared memory model is even more robust against large-scale data processing tasks. The reason is that, by default, such a system is very expensive to build and maintain. In addition, the shared memory model has poor scalability due to the relatively low bandwidth of the shared bus interconnect relative to the

© 2005 – ongoing JATIT & LLS
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

number of memory cells. the shuffle phase, it is required to do a lot of movement operations, which incur a lot of network traffic or thread-to-thread memory operations. To overcome that some research work points to developing optimization routines that can efficiently distribute the workload among the threads and improve traffic issues. In our work we propose a solution to kmeans clustering problem based on Mapreduce parallel programming model. The implementation was specified for a clustering task on an open source dataset that consists of many-dimensional points. The results was evaluated based on scalability metrics, in which we run the proposed clustering approach for different number of threads and clustering points.

A. Partitioned global address space model

PGAS is one kind of parallel programming paradigm, in which the address space is divided into two types: shared and private. Compared to other parallel programming paradigms PGAS combines features of both distributed and shared memory models, i.e. every thread can access both objects located in local address space and remote data in a transparent way using the same functions, provided by the particular implementation of the model. In the partitioned global address space model as illustrated in Figure 2, the global memory section contains distributed objects. These objects can be accessed by all running threads either directly or using special bulk copy operations. Since the PGAS model has a clear view of memory-address space, it is always easy to verify which thread is owning a particular memory location. In that way, objects can be placed following a certain pattern, in which they are distributed across threads such that objects logically having the same thread assignment are processed together. This situation illustrates the main benefits of relying on PGAS-based approach, since approaches based on shared memory lack that affinity property and hence treat all objects as addressed in the same memory space without a clear pattern of how to distinguish the ownership of each object in the shared memory.

UPC is a parallel programming language that operates in SPMD (single process, multiple data) mode. UPC follows the PGAS (Shared Global Address Space) programming model. The first version of the UPC was released in 1999. The specified number of threads operate independently, and each of them has private and shared memory areas [23]. A private area of memory is allocated for variables that are local to the executing thread. Shared memory (Figure 2) has an additional affinity property. The affinity is defined as a property of a shared memory location that resides in the local memory of the executing thread. Therefore, if a thread tries to access an area of memory outside of the thread's allocated shared memory, the access scheme will be different from the access scheme for local memory. This difference in access schemes allows you to control the placement of data in a distributed environment and improve data placement for the specific needs of the algorithm.

Shared qualifier in UPC is used for specifying variable`s memory allocation scheme of shared. Shared variables can be manipulated in a similar fashion to ordinary variables.

For number of threads THREADS pre-defined

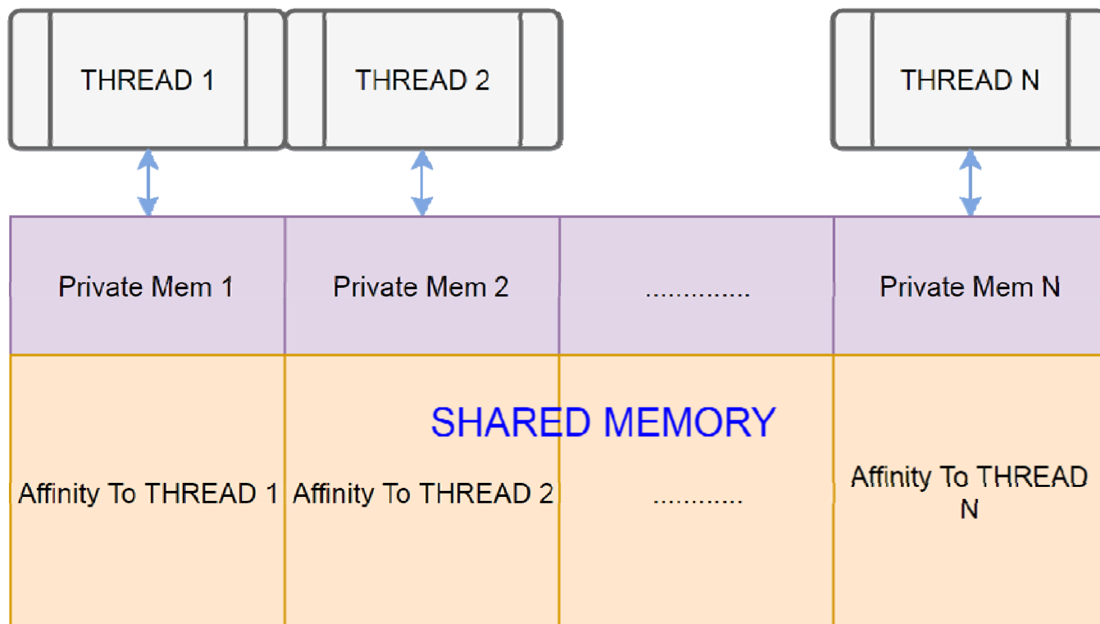

Figure 2: Memory model in PGAS

Journal of Theoretical and Applied Information Technology
30th November 2020. Vol.98. No 22
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

constant is used. The value of THREADS variable is the same among all threads.
Unique thread index is contained in MYTHREAD pre-defined constant.
Pointers to shared in UPC have the same purpose as usual pointers in C language. The difference is that pointers to shared are specifically designed to work on shared memory.

B. Mapreduce system

In this work we present a parallel centroid-based clustering algorithm using a Mapreduce system developed based on a partitioned global address space memory model. The given Mapreduce system [2] was designed to support arbitrary data types of keys and values. Hashmap data structure was used to allow a number of operations that are essential in Mapreduce (see Figure 5). First, hashmaps supports fast lookup/read/write operations for the keys, i.e. operations on hashmap are performed in asymptotic complexity of O(1). In our approach, the intermediate key/value pairs are stored in an affine hashmap located in a shared portion of thread`s memory (see Fig. 6). Reduce threads are assigned key-value pairs after shuffle collectively gathers and groups keys across hashmap structures.
The operations on a shared hashmap can be performed transparently by any thread, however, only a single, so called affine thread, is assigned to store in its local memory underlying key-value pairs associated with the data partition assigned to that thread. The cost of local operation by orders of magnitude faster than remote accesses, therefore, it is important to consider optimal thread-to-data mappings. In PGAS – Mapreduce system we proposed a scheduling approach that assigns threads to data according to an optimality criterion that consists of workload and network latency. The given optimization problem is solved by means of a genetic algorithm that tries to iteratively improve the scheduling till the process converges to a particular solution.

```
void * map (string filename)
{
    char * file_data;
    file_data = read_file_contents (filename);
    Vector tokens;
    vector_init(&tokens);
    Tokenize (file_data,&tokens);
    for (int i = 0;i<tokens.size;i++)
    {
        collect (vector_get (&tokens,i),1);
    }
    free(file_data);
}
```

Figure 3: Sample code for the map function in PGAS Mapreduce system

```
void reduce (string key,shared [] vector_sh *values)
{
    int i;
    int cnt = 0;
    for (i = 0;i<values->size;i++)
    {
        int v = vector_get_shared_copy (values,i);
        cnt+=v;
    }
    reduce_collect (key,cnt);
}
```

Figure 4: Sample code for the reduce function in PGAS Mapreduce system for Wordcount problem

In the PGAS Mapreduce system map and reduce function should be priorly implemented and provided as a function pointers to the input of init_mapreduce function, that will launch the parallel processing routines, leading to the execution of the specified Mapreduce task. In Figures 3 and 4 we provide an example of a map and reduce functions that implement the wordcount Mapreduce task. This is a standard task that computes a frequency of each word in a document corpus. To handle the task using Mapreduce we

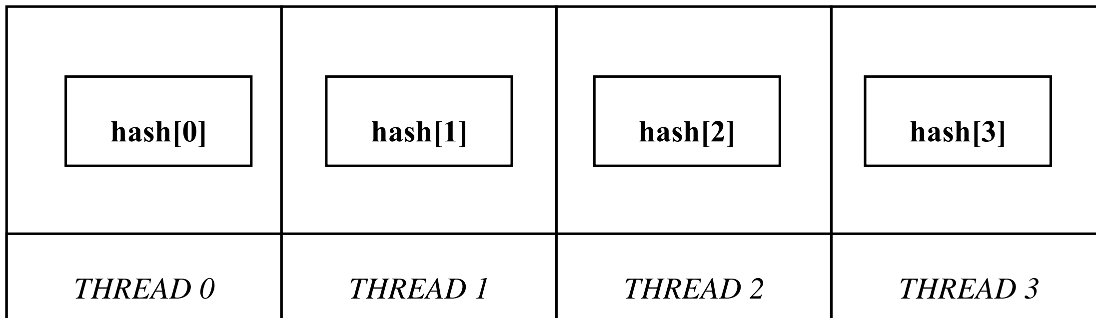

Figure 5: Shared array representation of hashmap

© 2005 – ongoing JATIT & LLS

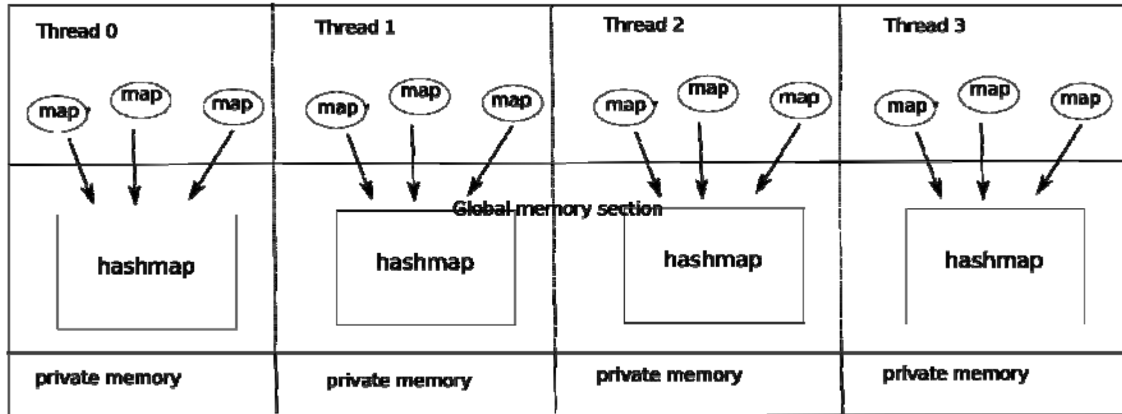

Figure 6: PGAS Mapreduce system: map processes, created in each thread write results to a shared hashmap

propose to use map task that tokenizes and emits key/value pairs of 'keyword/1' that are grouped and sended for execution to reduce threads that finally computes frequency values by taking the 'keyword/1' pairs corresponding to the same keyword and adding up all the entries received from the all map threads.

In article [24] the authors describe the Mapreduce framework implemented in the UPC language. The approach described in this article uses collective functions to exchange data in an arbitrary phase. The mapping and pruning functions in this approach work with the local storage of each node, and for this reason, the authors were forced to change the implementation of the collective UPC functions to work with the local memory space of each thread. In our implementation for the exchange of keys and values, we used a different approach based on the common hashmap data structure. Hashmap instances are in a common address space, and each instance resembles a single thread. Accordingly, every thread has access to a hashmap instance of every other thread.

In another paper [25], the authors presented a similar approach in which they used the X-10 library implementation of the hashmap data library to store intermediate key / value pairs locally in each stream, and then concatenated all values into a single stream. The X-10-enabled Mapreduce merge does not scale well because all data is moved to one location, and therefore has inherent limitations in the processing and storage capabilities of a single node.

In our approach, we store one instance of a shared hashmap per thread, so that each thread operates on the local part of its shared hashmap, and other threads can perform remote operations on that thread-local instance of the shared hashmap when needed. Therefore, the processing is not limited to the resources of a single node and requires efficient communication only after the completion of the map phase. In addition, in this way we can control the locality of operations for each hashmap instance and, as a result, later optimize the distribution of keys between threads to reduce the stage. Shared hashmap allows you to efficiently retrieve and write key / value pairs on average O (1) time complexity. Therefore, based on the features of the hashmap data structure, we tried to reduce the overhead associated with finding and extracting keys [25].

2. PROBLEM STATEMENT

A. Kmeans in Mapreduce

In standard Mapreduce-based clustering algorithms, the clustering process is divided into 2 parts: map and reduce. It is based on dividing the workload in which map processes are responsible for assigning data points to specific cluster centers [2]. The assignment is formed by making points that are closer to a corresponding cluster center. Data points that were assigned to the same cluster center are grouped together in a shuffle phase. Reduce phase is responsible for modifying cluster centers according to a mean sum of all data points in a cluster. The mean is computed according to Equation 2.

$$c = \frac{1}{count} \sum_{i=1}^{count} |pts_i| \qquad (2)$$

There are many other variants of the K-means algorithm, implemented using the Mapreduce approach [13-21]. In these approaches there are attempts to improve running time and performance by additional procedures such as elimination of outliers, better initialization of initial cluster centers.

© 2005 – ongoing JATIT & LLS
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
Algorithm: Map
input  : shared void * clusterCenters, string key, void * inputSplit
output: pair(int clusterCenterId,void * pointVal)
void * points = Tokenize (inputSplit);
foreach point ∈ points do
  | minDist ← maxDistanceValue;
  | for clusterID ← 1 to K do
  |   | dist = findDistance (point,clusterID);
  |   | if dist < minDist then
  |   |   | minDist ← dist;
  |   |   | clusterMinIndex ← clusterID;
  |   | Emit (clusterMinIndex,point);
  | end
end
```

Figure 7: Map for Mapreduce centroid-based Kmeans clustering

```
Algorithm: Reduce
input  : shared void * clusterCenters, int clusterID, void * values
output: pair(int clusterCenterId,void * newClusterCentroid)
foreach point ∈ values do
  | average ← average + findMagnitude(point);
end
clusterCenterId = clusterID;
newClusterCentroid ← getAverage(average);
Emit (clusterCenterId,newClusterCentroid);
```

Figure 8: Reduce for Mapreduce centroid-based Kmeans clustering

In Figures 7 and 8 algorithms for map and reduce functions in PGAS-based Mapreduce system.

Some works present design and implementation of Mapreduce systems that adopt partitioned global address space model. In their works authors specified strengths and weaknesses of their proposed systems [3, 4]. In our work, however, we present an implementation based on the Mapreduce system developed previously in one of our preceding works, that was mostly convenient for us to work with and more appropriate to follow based on previously obtained results [1, 5].

In our work we devised a Mapreduce solution to implement a kmeans-parallel algorithm on a PGAS-based Mapreduce framework.

Novel parallel algorithms and approaches need a thorough study in order to understand scalability and performance benefits in using these new parallel algorithms and frameworks. Apache Hadoop, for example, have limitations in terms of poor performance in solving iterative tasks and Apache Spark, although provides a framework for efficient in-memory iterative data processing, still do not resolve the problem of locality that PGAS Mapreduce approach naturally could address.

PGAS Mapreduce framework relies on efficient tools to handle basic Mapreduce operations like map, shuffle and reduce. From a user standpoint, it is only required to implement map and reduce functions, the other workload is performed automatically by the system that handles all operations to perform parallel Mapreduce execution pipeline.

The map and reduce functions mostly describe the algorithmic aspect of the work, the rest, including steps to perform complex operations of key exchange or creating optimized data distribution plans are handled by the Mapreduce system in the background.

Map processes send key/value pairs for further processing using an *Emit* function that saves pairs into a local part of a shared hashmap structure. After map processes finish their execution a shuffle procedure collectively merges all the generated keys for distributing key/value pairs to their scheduled threads for the *reduce* task execution. Reduce processes fetch the data from a shared hashmap according to the collective exchange algorithm described above.

In the worst case k-means algorithm could lead to an exponential time complexity. Nevertheless, in practice the algorithm shows good results in terms of quality and speed. However, the main limitations of kmeans algorithms lie in slow convergence, local optimum problem, in which the algorithm tends to converge to a local minima, therefore, making the algorithm sometimes impractical to use due to long fine-tuning procedures of an appropriate randomized sample of initial points.

Due to these issues, researchers tried to find a good initialization routine that could provide some guarantees to the estimated optimal clustering solution. In the kmeans-parallel algorithm [13] it was shown that efficient clustering algorithm k means-parallel avoids the inherently sequential nature of kmeans++ algorithm, described in [14]. In their work authors derive a theoretical bounds on the cost of optimal solution compared to the solutions, obtained using kmeans-parallel algorithm. Their results prove a constant factor approximation to the optimal solution, and hence provides a decent solution to the problem of locality of the naive kmeans approach. On a series of experiments on the commonly used benchmarks, authors were able to demonstrate the practical benefits of the algorithm compared to the kmeans++ and naive kmeans.

Their work reduces the number of passes required to get a nearly optimal clustering result from *k to log (k)*, effectively making the algorithm an excellent candidate to handle high-dimensional feature space of clustering objects. Surprisingly, in practice authors showed that as many as five iterations of an algorithm is enough to converge to an optimal set of clusters. The general idea of the algorithm is to oversample at each step a number of clustering centers. As shown in Figure 4, the algorithm starts by finding an initial cluster center

© 2005 – ongoing JATIT & LLS
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

C. Following step involves computing an initial cost of clustering with only a single cluster center chosen. Obviously, the initial solution will make a huge error in clustering cost. The next phase of the algorithm samples *l* new clustering centers each step from the specific distribution that weights the points located nearby to some of the clusters lower compared to points located further from the clusters. The distance between a point and clustering center is taken as minimum of the norm over differences between the point *x* and a point taken from the set *C* of current clustering center choices. While the cost is taken as a squared sum of distances between a subset of points *Y* and current set of clusters *C*. In a parallel implementation of the k-means-parallel algorithm subsets can designate a portion of points taken from the dataset by means of some data decomposition routine. Final step of the clustering involves a weighted reclustering routine that does one more kmeans run over the cluster centers obtained during the main loop of the algorithm. This step accounts for oversampling of cluster centers when the number of clusters exceeds the number of clusters that is required to obtain *K*. In that algorithm compared to kmeans++, there is an additional factor of *l* in the numerator of the probability computation in the sampling routine, due to necessity to account for oversampling during the sampling process.

The next step of the algorithm involves an iterative procedure, in which we incrementally, at each iteration, obtain a new set of cluster centers, which consists of *l* new sampled points over the distribution that we employed. The distribution is computed according to the specified approach presented in Equations 3-4. Inside the iterative part we perform two different mapreduce task executions. The first task is designed to update the distribution function, the second to sample new points. The distribution function is designed to construct an array of weights with the size equal to the number of points that was assigned to the current map task. The index of the point that should be chosen according to that distribution is computed by taking the randomly generated number R from uniform distribution in range [0, 1], and searching through the array to find a place in which the normalized cumulative sum up to that index exceeds the randomly generated number R.

In the first map stage, we need to fetch the points for each map thread and then update the locally cached table of probabilities.

$$d(x, C) = min_{y \in C} \|x - y\| \quad (3)$$

$$cost_Y(C) = \sum_{y \in Y} d^2(y, C) \quad (4)$$

Algorithm 1 Kmeans++

C = sample a point uniformly from X
while $|C| < k$ **do**
  sample X with probability $\frac{d^2(x,C)}{cost_X(C)}$
end while

Figure 9: K-means++ algorithm.

Algorithm 2 Kmeans||

C = sample a point uniformly from X
$c = cost_X(C)$
while *iteration* $< \log k$ **do**
  $C\prime$ = sample each point $x \in X$ with probability $p_x = \frac{l * d^2(x,C)}{cost_X(C)}$
  $C = C \cup C\prime$
end while
Recluster the weighted points in C into k clusters
For $x \in C$, set w_x to be the number of points in X closer to x than any other point in C

Figure 10: K-means parallel algorithm.

3. RESULTS

A. Kmeans in Mapreduce

In the K-means parallel algorithm a set of k clustering points are chosen, where each point sampled according to a probability distribution , in which the cluster centers are sampled with the probability proportional to a distance between a point and an already chosen set of clusters C.

In our study the algorithm was divided into several mapreduce jobs. In the first step of computing the initial cluster center, it is required to evaluate the cost function over the whole dataset, in which we choose one randomly chosen cluster point c. For the task of computing the cost value $cost_X(C)$ the map task is formed, which runs over the local part of the dataset that belongs to executing map thread. The output of that function is fetched and distributed across reduce threads that sum the local values of the cost function, obtained over points belonging to each thread. Hence, we obtain the cost over the whole set of points (see Figure 11).

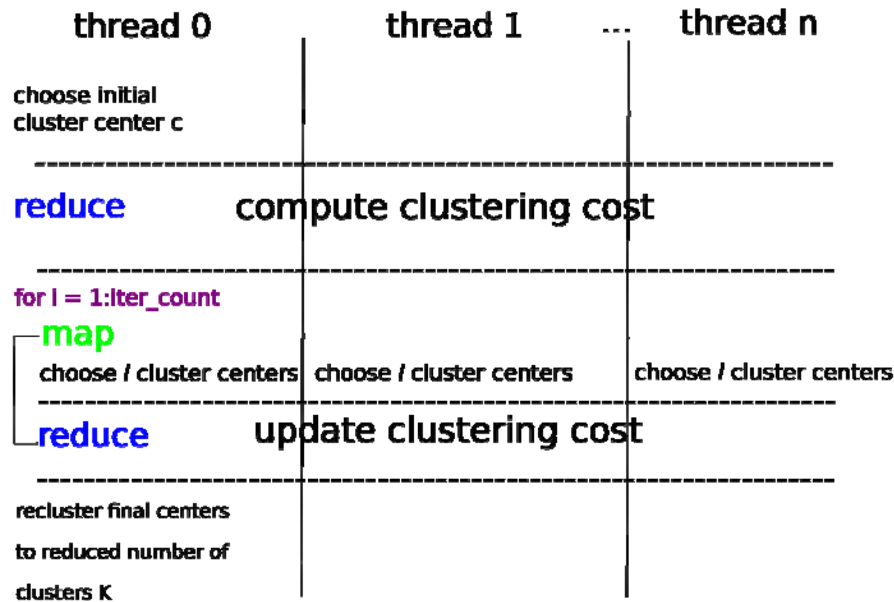

Figure 11: The scheme of Mapreduce algorithm

B. Experimental results

Experimental hardware used in testing our proposed solution was a virtual machine, hosted on a DigitalOcean cloud infrastructure. The given machine consisted of 32 cores, 64GB memory and 400GB SSD drive. As a software we used the Berkeley UPC runtime and compiler, version 2020.4.0. All the included libraries that were used in our implementation were written using UPC Berkeley runtime and compiler.

In order to test our proposed implementation, we used a synthetic dataset from [22], which consists of 105000 64-dimensional vector entries. Points from the dataset priorly represent 25 clusters and our task was to use our Kmeans parallel implementation to see the speed-up depending on the number of threads.

Our approach was to test the algorithm with two different experiments:

1) Fixed number of oversampling factor l
2) Fixed number of iterations

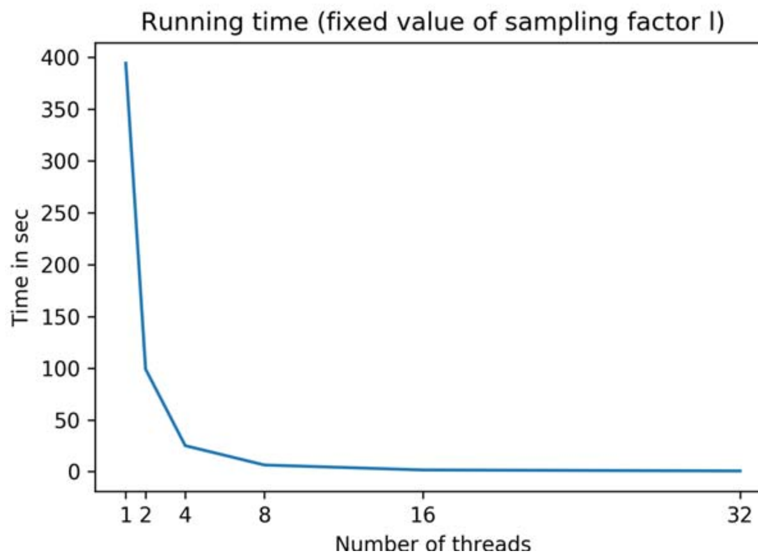

Figure 12: Scalability of the Kmeans Mapreduce algorithm with fixed value of oversampling factor l

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

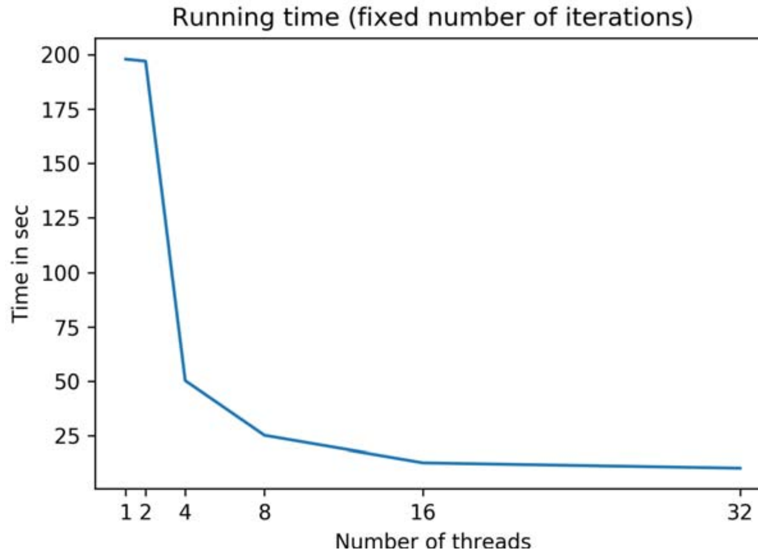

Figure 13: Scalability of the K-means Mapreduce algorithm with fixed number of iterations

As can be seen in Figure 12 our approach allows us to achieve a linear speed-up in terms of running time in case of the fixed oversampling factor l experiment. The experiment with fixed oversampling factor implies that by fixing the total number of points M that are required to be chosen as cluster centers, we derive the value of the fixed oversampling factor *l* to be equal to the total number of points *M* divided by the product of number of threads *T* and oversampling factor *l*.

$$l = \frac{M}{T * I} \quad (5)$$

Similarly, when we state that the number of iterations is fixed, we have to change the formula for number of fixed iterations *I* to be equal to the ratio of number of points *M* over the product of number of threads *T* and the value of oversampling factor *l*.

$$I = \frac{M}{l * T} \quad (6)$$

In the linear case of a single thread, we used just a standard K-means++ approach, which linearly chooses each cluster center, giving no options to do parallel job assignments. Similarly, fixing the number of iterations provides a linear speed-up, which can be seen in Figure 13. As we can see our solution provides an almost theoretically best speed-up that is possible to achieve.

Table 1: running time for the kmeans parallel with fixed value of l

# threads	value of *l*	# iterations	time (sec)
1	3	64	394.50
2	3	32	99.00
4	3	16	25.08
8	3	8	6.36
16	3	4	1.58
32	3	2	0.64

Table 2: running time for the kmeans parallel with fixed value of iterations

# threads	value of *l*	# iterations	time (sec)
1	64	10	197.96
2	32	10	197.03
4	16	10	50.36
8	8	10	25.25
16	4	10	12.82
32	2	10	10.39

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

4. CONCLUSION

In our work we presented a partitioned global address space based Mapreduce implementation of Kmeans parallel algorithm. Performance wise many Mapreduce systems cannot guarantee a linear speed-up and hence a new parallel methods and frameworks needs a thorough study in order to understand scalability and performance benefits in using new parallel algorithms and frameworks. Apache Hadoop, for example, have limitations in terms of poor performance in solving iterative tasks and Apache Spark, although provides a framework for efficient in-memory iterative data processing, still do not resolve the problem of locality that PGAS Mapreduce approach naturally could address. It was shown that our parallel implementation provides a strongly linear speedup. Thus, by increasing number of threads the amount of input that algorithm could process roughly doubles in size, therefore making our solution suitable for solving clustering problems in high-dimensional and large-scale datasets.

ACKNOWLEDGEMENTS

This work was supported in part under grant of Foundation of Ministry of Education and Science of the Republic of Kazakhstan “Development of a system for knowledge extraction from heterogeneous data sources to improve the quality of decision-making” under project ID no. AP05132933.

REFERENCES:

[1] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107–113. DOI:https://doi.org/10.1145/1327452.1327492

[2] McKenna A., Hanna M., Banks E., Sivachenko A., Cibulskis K., Kernytsky A., DePristo M. A. The genome analysis toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data //Genome Research.-2010.-Vol. 20, № 9.-P.1297-1303.

[3] Nguyen T., Shi W., Ruden, D. CloudAligner: A fast and full-featured MapReduce based tool for sequence mapping //BMC Research Notes.-2011.- Vol.4.

[4] Kang U., Tsourakakis C. E., Faloutsos C. PEGASUS: A peta-scale graph mining system - implementation and observations // Proceedings - IEEE International Conference on Data Mining, ICDM.-Miami, Florida, USA, 2009.-P.229-238.

[5] Pantel P., Crestan E., Borkovsky A., Popescu A., Vyas, V. Web-scale distributional similarity and entity set expansion// Proceedings of the EMNLP 2009 - Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: A Meeting of SIGDAT, a Special Interest Group of ACL, Held in Conjunction with ACL-IJCNLP 2009.-Suntec, Singapore, 2009.-P.938-947.

[6] Zhao W., Ma H., He Q. Parallel K-means clustering based on MapReduce // Proceedings of IEEE International Conference on Cloud Computing.-Beijing, China, 2009.-P.674-679.

[7] Chuck Lam. 2010. *Hadoop in Action* (1st. ed.). Manning Publications Co., USA.

[8] Zaharia M., Chowdhury M., Franklin M.J., Shenker S., Stoica I. Spark: Cluster computing with working sets //Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, ACM.-Boston, MA, USA, 2010.-P.10-10.

[9] Aday S., Darkhan AZ., Madina M. (2017) PGAS Approach to Implement Mapreduce Framework Based on UPC Language. In: Malyshkin V. (eds) Parallel Computing Technologies. PaCT 2017. Lecture Notes in Computer Science, vol 10421. Springer, Cham

[10] Teijeiro, C., Taboada, G.L., Tourino, J., Doallo, R.: Design and implementation of Mapreduce using the PGAS programming model with UPC. In: 17th International Conference on Parallel and Distributed Systems (ICPADS 2011), pp. 196–203. IEEE Computer Society, Washington (2011).

[11] Dong, H., Zhou, S., Grove, D.: X10-enabled MapReduce. In: 4th Conference on Partitioned Global Address Space Programming Model (PGAS 2010), pp. 1–6. ACM, New York (2010). doi: 10.1145/2020373.2020382

[12] Shomanov, A. S., Mansurova, M. E., & Nugumanova, A. B. (2018). Design of K-means clustering algorithm in PGAS based mapreduce framework. Paper presented at the IEEE 12th International Conference on Application of Information and Communication Technologies, AICT 2018 - Proceedings, doi:10.1109/ICAICT.2018.8747118

[13] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. 2012. Scalable k-means++. Proc. VLDB Endow. 5, 7 (March 2012), 622–633.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

DOI:https://doi.org/10.14778/2180912.2180915

[14] David Arthur and Sergei Vassilvitskii. 2007. K-means++: the advantages of careful seeding. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '07). Society for Industrial and Applied Mathematics, USA, 1027–1035.

[15] Ramdani, A & Firmansyah, H. (2019). Pillar K-Means Clustering Algorithm Using MapReduce Framework. IOP Conference Series: Earth and Environmental Science. 258. 012031. 10.1088/1755-1315/258/1/012031.

[16] Sreedhar, C., Kasiviswanath, N. & Chenna Reddy, P. Clustering large datasets using K-means modified inter and intra clustering (KM-I2C) in Hadoop. J Big Data 4, 27 (2017). https://doi.org/10.1186/s40537-017-0087-2

[17] Wang, Hui & Zhou, Chengdong & Li, Leixiao. (2019). Design and Application of a Text Clustering Algorithm Based on Parallelized K-Means Clustering. Revue d'Intelligence Artificielle. 33. 453-460. 10.18280/ria.330608.

[18] Cui, X., Zhu, P., Yang, X. et al. Optimized big data K-means clustering using MapReduce. J Supercomput 70, 1249–1259 (2014). https://doi.org/10.1007/s11227-014-1225-7

[19] A. Boukhdhir, O. Lachiheb and M. S. Gouider, "An improved mapReduce design of kmeans for clustering very large datasets," *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*, Marrakech, 2015, pp. 1-6, doi: 10.1109/AICCSA.2015.7507226.

[20] Van Hieu, Duong & Meesad, Phayung. (2014). Fast K-Means Clustering for Very Large Datasets Based onMapReduce Combined with a New Cutting Method; http://link.springer.com/chapter/10.1007%2F978-3-319-11680-8_23. Advances in Intelligent Systems and Computing. 326. 10.1007/978-3-319-11680-8_23.

[21] Ma, Li & Gu, Lei & Li, Bo & Ma, Yue & Wang, Jin. (2015). An Improved K-means Algorithm based on Mapreduce and Grid. International Journal of Grid and Distributed Computing. 8. 189-200. 10.14257/ijgdc.2015.8.1.18.

[22] S. Sieranoja and P. Fränti, "Fast and general density peaks clustering", Pattern Recognition Letters, 128, 551-558, December 2019.

[23] Carlson W.W., Draper J.M., Culler D.E., Yelick K., Brooks E., Warren K. Introduction to UPC and language specification //Technical Report CCS-TR-99-157, IDA Center for Computing Sciences.-1999.-p.17.

[24] Teijeiro C., Taboada G.L., Tourino J., Doallo R. Design and Implementation of MapReduce Using the PGAS Programming Model with UPC //Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS '11).-Tainan, Taiwan, 2011.-P.196-203.

[25] Dong H., Zhou S., Grove D. X10-enabled MapReduce. //Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS '10). ACM.-New York, USA, 2010.-p.6.