IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 10, October 2017

Survey of Parallel Implementations of Clustering Algorithms

Pranav Kadam[1], Sai Jadhav[2], Anirudh Kulkarni[3], Savita Kulkarni[4]

Student, Department of E&TC Engineering, Maharashtra Institute of Technology, Pune, India [1,2,3]

Professor, Department of E&TC Engineering, Maharashtra Institute of Technology, Pune, India [4]

Abstract: Cluster analysis is one of the primary approaches in Machine Learning. Applications involving clustering often deal with large and high dimensional datasets. Clustering of large datasets is a task having high time complexity. Clustering algorithms iterate several times before converging to the solution. A way to speed up the clustering process is Parallel Processing. Parallel programs make use of Graphical Processing Unit (GPU) and/or multi-core CPUs to reduce the computation time. There is enough scope to parallelize clustering algorithms for obtaining faster results. This paper gives a review of parallel implementations of three clustering algorithms viz. k-means, DBSCAN and Expectation-Maximization. We survey the Shared Memory and Message Passing models of parallel programming and how clustering algorithms have been performed using them. We also highlight few applications involving large data where parallel programming would be helpful.

Keywords: Machine Learning, Clustering, Clustering Algorithms, Parallel Processing, High Dimensional Data.

I. INTRODUCTION

In Machine Learning, Unsupervised learning deals with unlabelled datasets. Its main task is to analyse this data and deduce a function to describe a structure out of it. One of the primary approaches to unsupervised learning is clustering. Clustering refers to grouping the data objects into classes (or clusters) in such a way that members of same cluster share some common characteristics and differ from the members of other clusters.

Clustering is a task involving high time complexity especially when dealing with large and high dimensional data. The clustering algorithms run through several iterations before finally converging to the desired result which is the clustered data. The processing of large datasets demands use of parallel and distributed computing methods for faster results. The developments in processor technology such as multi-core processors, Graphical Processing Unit (GPU) for Parallel Programming and use of Workstations for scientific computing have made it possible to reduce the computation time of clustering algorithms.

This paper gives survey of parallel implementation of various clustering algorithms and how particular steps of the algorithm can be computed using specific parallel programming models. The research paper is organized as follows. In II we survey the Parallel Programming models and software platforms / libraries for running parallel programs. In III we discuss three clustering algorithms and review how researchers have implemented these algorithms using parallel processing and their results. In IV we discuss the applications of clustering and parallel computing on large data sets in brief. In V we summarize and conclude our study.

II. PARALLEL PROGRAMMING MODELS

A parallel programming model is an abstraction of the hardware architecture on which the parallel code is running. There exist several parallel computing models as a result of the various ways in which different processors can be grouped together to make a parallel system [1]. Following is a brief study of two such models.

A. Shared Memory

Shared Memory refers to the Parallel Processing model where different processors access the same block of memory. In shared memory architectures communication is implicitly specified as memory is accessible to all processors [18]. Processors interact by modifying data objects stored in the common memory. If the processor takes same time to access any memory word then it is classified as a Uniform Memory Access (UMA) system. However if time taken to access different memory words differ then the platform is called Non-Uniform Memory Access (NUMA).

1) OpenMP:

Open Multi-Processing (OpenMP) is an API which supports programming of shared memory platforms. OpenMP consists of a set of compiler directives, pragmas and a runtime which provides both management of the thread pool and a set of library routines. These directives instruct the compiler to create threads, perform synchronization operations, and manage shared memory. Therefore, OpenMP does require specialized compiler support to understand and process these directives. One of the disadvantages of using OpenMP is that it cannot scale more than a couple of hundred nodes because of thread management overheads and cache coherence hardware requirement [16].

2) CUDA:

Compute Unified Device Architecture (CUDA) is a parallel computing platform created by NVIDIA in order to use the GPU for general purpose computing. CUDA allows developers to build applications using the large number of processor cores provided by the GPU. The parallel system using CUDA consists of host and device. CPU is the host and the device is usually the GPU where computation is carried out by several threads running in parallel. The GPU architecture for threads consists of two level structures of blocks and grids. A block is a set of tightly coupled threads while a grid is a set of loosely coupled blocks [17].

B. Message Passing

Message Passing is a parallel programming model where communication between processes is done by exchanging messages [Parallel Review]. It supports only explicit parallelization. This is a common model for a distributed memory system, where communication cannot be achieved by sharing variables. In Message Passing communication occurs when part of the address space of one process is copied into the address space of another process [1].

1) MPI:

Message Passing Interface (MPI) has been evolved as a library for Message Passing. It is extensively used for High Performance Computing (HPC) applications on distributed architectures [1]. With MPI it is the job of the programmer to explicitly partition the data and decide which tasks are to be managed by each process. It is found in clustered workstations and non-shared address space multi-computers [18]. MPI finds its use in high- performance scientific computing domain. Cluster computing systems having more than 1,00,000 nodes have successfully used MPI for various applications [16]. The basic communication operations in MPI are summarized in the following table:

TABLE I
MPI COMMUNICATION ROUTINES

Operation	**Function**
MPI_Send	Sends message to a process
MPI_Recv	Receives message from a process
MPI_Bcast	Broadcasts messages to all processes
MPI_Reduce	Accumulates data into a single process
MPI_Scatter	Sends unique message to each process
MPI_Gather	Receives unique messages from all processes

TABLE II
FEATURE COMPARISON OF PARALLEL PROGRAMMING PLATFORMS

Features	**CUDA**	**OpenMP**	**MPI**
Memory model	Shared	Shared	Shared and Distributed
Management and division of data	Explicit	Implicit	Explicit/ Implicit
Communication	Shared address space	Shared address space	Message passing
Extensive Work on	GPU	GPU/CPU	CPU

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 10, October 2017

III. CLUSTERING ALGORITHMS AND THEIR PARALLELIZATION

Most researchers describe a cluster by considering the internal homogeneity and the external separation i.e., patterns in the same cluster should be similar to each other, while patterns in different clusters should not. Both the similarity and the dissimilarity should be examinable in a clear and meaningful way [2]. Although the common goal of any clustering algorithm is to form groups of data objects on the basis of certain rules, researchers classify the algorithms into different types. Clustering algorithms have been divided into nine categories in [3]. However in this paper we concentrate on algorithms based on three of these categories:

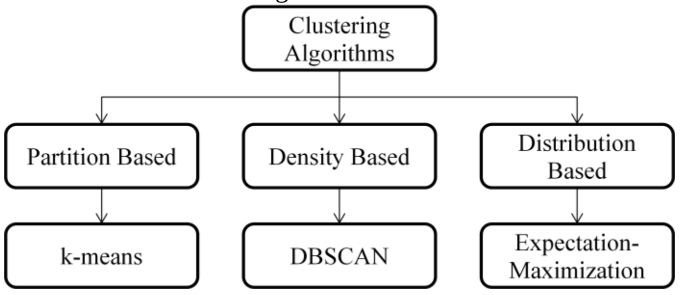

Fig. 1 Classification of Clustering Algorithms

A clustering algorithm based on partition regards the centre of data points as the centre of corresponding cluster. K-means [15] is one of the famous partition based algorithm. In density based algorithms data which is in the region of high density is considered to belong to the same cluster. Typical density based clustering algorithms is DBSCAN [10]. If there are several distributions in data such as Gaussian distribution then data belonging to a particular distribution can be grouped as one cluster. This is the basic idea of Distribution based clustering. The Expectation-Maximization algorithm [9] is a part of this category. An overview of these algorithms is as follows.

A. K-means Algorithm

The k-means algorithm was first presented by James MacQueen in 1967 [15]. The objective of k-means is to group n data points into k clusters such that each cluster has maximum similarity as defined by an objective function. The steps involved are:

1. Choose value of k.
2. Generate k clusters randomly and determine centre of each cluster which is the mean for that cluster.
3. Compute the distance between data point and each of the k means and assign the point to the cluster having least distance with its mean. Repeat for all data points.
4. Re-compute the centre of each new cluster formed.
5. Repeat steps 3 and 4 until the algorithm converges i.e. the assignment in step 3 no longer changes.

The choice of parameter k depends on the application and the distribution of data points in space. A common distance measure in step 3 is Euclidian distance, however different distance measures such as City-block, Chebyshev, Cosine distance etc. as described in [3] can be used depending on which is best suited for the application.

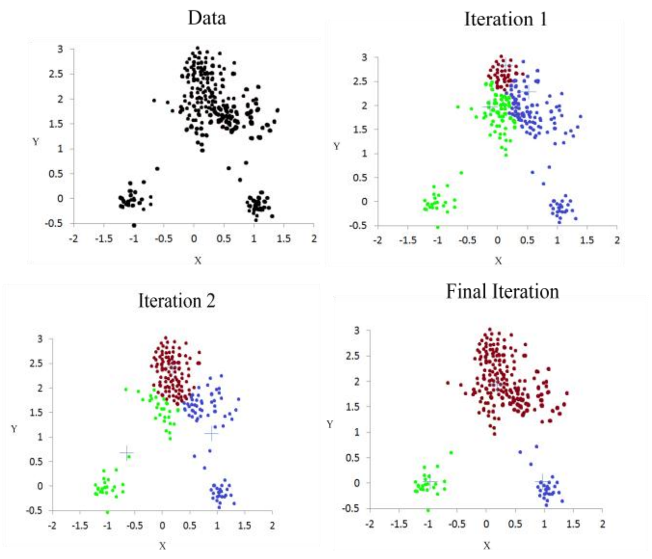

Fig. 2 K-means algorithm

1) Parallel Implementation of k-means clustering:

The above mentioned K-means clustering algorithm leaves a lot of space for improvement in terms of speed and power consumption. A way to parallelize k-means algorithm is presented in [4]. The proposed approach is to implement parallelization in the first phase of each step, where we calculate the nearest centroid to each point. The parallelization is done on CUDA platform. CUDA technology gives computationally intensive applications access to the tremendous processing power of the latest GPUs through a C-like programming interface.

Introducing parallelization basically consist of creating threads which are responsible for carrying out a job simultaneously. Each processor can implement one thread at a time. For uniform working, the number of threads should be a multiple of the number of processing units. As previously mentioned, the time-dominant phase of the algorithm is in the assignment of data points to each cluster in which the Euclidean distance between data point and each of the k means is calculated and tries to reassign each point to the nearest cluster. In CUDA we can create a thread to calculate these distances for a single point. This thread would go through all the centroids, calculating the distance between them and the respective point and finding the minimum of them finally to become a part of that cluster. In a fantastic example where the number of processing units is equal to the thread, this would be finished in one single step. Otherwise, realistically, *p* cores would speed up our process by a factor of *p*.

The next phase of the algorithm that recomputes the centre of each new cluster formed is better left untouched for several reasons. The most important is that the computations in this step require memory sharing and create a lot of congestion. As it is, parallelizing this step is not going accelerate our process by that great a factor. Hence, this step is done serially. The results of implementing k-means clustering algorithm on CUDA are satisfactory as far as time consumption is concerned. The results of [4] are tabulated below.

TABLE III
RESULTS OF K-MEANS CLUSTERING ON DIFFERENT HARDWARE

Platform	**Time (s)**	**Performance Increase**
Intel Pentium D, 3 GHz	9.830	1 X
NVIDIA 8600 GT	0.724	13.57 X
NVIDIA 8800 Ultra GTX	0.144	68 X

B. DBSCAN Algorithm

Density Based Spatial Clustering of Applications with Noise (DBSCAN) [10] groups together points that are closely packed together in space and marks points in low-density region as noise. It has two parameters minPts (minimum number of points) and ε (Eps). DBSCAN defines a core point, border point and a noise point as follows:

1. A point is a core point if it has more than minPts within a radius of ε.
2. A border point has fewer than minPts within ε, but is in the neighbourhood of a core point.
3. A point which is neither core nor border point is called as noise point.

Formation of cluster is as follows:

1. Any two core points which are within a distance ε of one another are put in the same cluster.
2. Any border point which is within a distance ε from a core point is put in the same cluster as the core point.
3. Noise points are discarded and are not a part of the cluster.

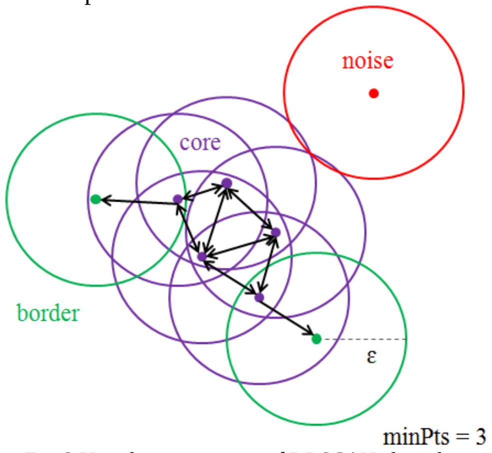

Fig. 3 Visual representation of DBSCAN algorithm

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 10, October 2017

1) Parallel Implementation of DBSCAN algorithm:
Parallelizing DBSCAN algorithm is a difficult task because the basic algorithm is sequential in nature. An existing but flawed parallelization method is the master-slave model in which the slave is responsible for a chunk of data. The slave calculates its local cluster. All the slaves send these clusters to the master which finally merges them all sequentially to get the final result [6].

A better alternative is to use disjoint-set data structure [5]. The algorithm is very simple. It creates a single node tree for each point in the set. Each thread/process is responsible for one node tree. This step is done in parallel without any communication. By recognizing which points belong in a cluster, their trees are merged together with the help of disjoint-set data structure. This is done until all the clusters are gone over. In the end we get a single tree for each cluster. This enables substantial speed up and scalability.

Parallel DBSCAN on Shared Memory Computers:
In shared memory there are threads which run in parallel. The dataset is divided in p groups $\{X1, X2, . . , Xp\}$ such that each thread is responsible for each group [5]. Each thread finds a cluster using the points in its group. This is called local computing. The second step is merging. Both these steps run in parallel. The merging operation uses a lock mechanism. In order to be able to merge two nodes a thread must acquire its lock first. Depending on the status of its lock the thread understands whether the node is already a part of some cluster or if it is still a root. When two threads try to acquire the same lock, one of them has to wait for the other to finish.

Parallel DBSCAN on Distributed Memory Computer:
Parallel DBSCAN on distributed memory is very similar to that on shared memory. Instead of threads the data is divided such that each processor gets a chunk of data [5]. In order to find the neighbors of a local point the distance between them is found out. If it falls below a certain threshold value then that point is termed as its local neighbor. Similar to shared memory there are two steps: local computing and merging. The merging on distributed memory takes place by message passing between processors.

Results of parallel DBSCAN on shared and distributed memory architectures are summarized below.

TABLE IV
RESULTS OF PARALLEL DBSCAN

Memory type	**Cores used**	**Speedup factor**
Shared memory	40	25.97
Distributed memory	8,192	5,765

C. Expectation-Maximization Algorithm
The Expectations-Maximization (EM) algorithm was presented by Arthur Dempster, Nan Laird, and Donald Rubin in 1977. EM was summarized in [9]. Following the conventions in [9], EM is an algorithm which estimates unknown parameters Θ from data U marginalizing over hidden variables J. EM alternates between estimating the unknowns Θ and hidden variables J. After computing Θ at every iteration EM computes a distribution over space J.
The E-step constructs a local lower-bound to the posterior distribution, whereas the M-step optimizes the bound, thereby improving the estimate for the unknowns.
At each iteration, the EM algorithm first finds an optimal lower bound $B(\Theta;\Theta^t)$ at the current guess Θ^t and then maximizes this bound to obtain an improved estimateΘ^{t+1}. The two steps in EM algorithm can thus be conveniently summarized as:

1. E-step: Calculate $f^t(J) = P(J|U,\Theta^t)$.
2. M-step: Maximize the expression $[Q^t(\Theta) + \log P(\Theta)]$ for Θ and assign value to Θ^{t+1}.

$Q^t(\Theta)$ is calculated in the E-step by evaluating $f^t(J)$ using the current guess Θ^t, whereas in the M-step we are optimizing $Q^t(\Theta)$ with respect to the free variable Θ to obtain the new estimate Θ^{t+1}.

1) Parallel Expectation-Maximization:
All steps of the algorithm are potentially parallelizable once they iterate over the entire data set. In [7], a parallel implementation of EM for training Gaussian Mixture Model (GMM) using CUDA is proposed. For parallelization, the

main loop of the algorithm is implemented sequentially and different CUDA kernels are in charge of running different steps of the algorithm. Pseudo code for each CUDA kernel implementation is properly provided.
Experiments performed over a UCI database and varying number of Gaussians have shown a speedup of 7 as compared to serial implementation. Modifications in two of the CUDA kernels have also been provided in order to allow more coalesced access to global memory.

Parallel implementation of EM on GraphLab in [8] shows scaling for up to 8 machines. Good performance is, to some degree due to the lack of latent variable structure, which makes the computation highly data parallel.

IV. APPLICATIONS

Applications of clustering have been found in character recognition, web analysis, classification of documents, classification of astronomical data and classification of objects found in an archaeological study [14]. Many applications work on huge data sets. After maximizing the performance of each core the next logical step was going with a multi-core approach. With each generation of new multi-core and many-core processors parallel processing applications benefit hugely. Parallel processing has given way to better and faster implementations of many applications. Parallel processing is used in data mining which uses large data sets [13]. Data analysis creates complex problems which are solved by parallel and distributed computing-based systems and technologies. Another application which has to deal with a large data set is astronomy. Parallel processing takes complete advantage of the multi core facilities available in an astronomical application. Thus High Performance Computers are used efficiently and to their maximum capacity. Power system computation can also be done using parallel mechanisms on shared and distributed memory [12]. Parallel computation also has applications in medical imaging and image processing fields which involve image reconstruction, image de-noising, motion estimation and deformable registration [11].

V. CONCLUSION

This paper reviews the current parallel programming landscape and its applicability to cluster analysis. It provides guidelines which help us in choosing the appropriate parallel programming model and platform according to our application and hardware availability. In case of a CPU with average performance, we can rely on multi-core GPUs for parallelization. Such applications can be performed on CUDA or OpenMP. If we have a CPU rich in terms of its multi-core architecture, distributed memory models like MPI should be used. For best results, a perfect combination of both shared memory and message passing models is recommended. The paper also talks about the parallel implementations of three clustering algorithms, namely k-means, DBSCAN and Expectation-Maximization. It is evident from the study that the degree of speedups provided by parallelizing these algorithms is considerably high.

REFERENCES

[1] Javier Diaz-Montes, Camelia Muñoz-Caro and Alfonso Niño, “A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era,” IEEE Transactions on Parallel and Distributed Systems · June 2012 DOI: 10.1109/TPDS.2011.308
[2] Rui Xu and Donald Wunsch II, “Survey of Clustering Algorithms,” IEEE Transactions on Neural Networks, Vol. 16, No. 3, May 2005.
[3] Dongkuan Xu and Yingjie Tian, “A Comprehensive Survey of Clustering Algorithms,” Annals of Data Science 2:2, 165-193. Online Publication date: 1-June-2015.
[4] Reza Farivar, Daniel Rebolledo, Ellick Chan and Roy Campbell, “A Parallel Implementation of K-Means Clustering on GPUs,” in Proceedings of the International Conference on Parallel and Processing Techniques and Applications, PDPTA 2008, Las Vegas, Nevada, USA, July 14-17, 2008, 2 Volumes.
[5] Md. Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne and Alok Choudhary, “A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-Set Data Structure,” 2012 24th International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2012 - Salt Lake City, UT, United States.
[6] Domenica Arlia and Massimo Coppola, “Experiments in Parallel Clustering with DBSCAN,” Euro-Par 2001: Parallel Processing: 7th International Euro-Par Conference Manchester, UK August 28–31, 2001, Proceedings. Springer Berlin.
[7] Araújo, G.F., H.T. Macedo, M.T. Chella, C.A.E. Montesco and M.V.O. Medeiros, “Parallel Implementation of Expectation-Maximisation Algorithm for the Training of Gaussian Mixture Models,” Journal of Computer Science 10 (10): 2124-2134, 2014 ISSN: 1549-3636 © 2014 Science Publications.
[8] Henggang Cui, JinliangWei and Wei Dai, “Parallel Implementation of Expectation-Maximization for Fast Convergence”.
[9] Dellaert Frank, “The Expectation Maximization Algorithm,” College of Computing, Georgia Institute of Technology, Technical Report number GIT-GVU-02-20, February 2002.
[10] M. Ester, H. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in Proceedings of the 2nd International Conference on Knowledge Discovery and Data mining, vol. 1996. AAAI Press, 1996, pp. 226–231.
[11] YasserM. Kadah, Khaled Z. Abd-Elmoniem and Aly A. Farag, “Parallel Computation in Medical Imaging Applications,” Hindawi Publishing Corporation International Journal of Biomedical Imaging Volume 2011, Article ID 840181, 2 pages doi:10.1155/2011/840181
[12] C. Lemaitre and B. Thomas, “Two Applications of Parallel Processing in Power System Computation,” in IEEE Transactions on Power Systems. Vol. 11, No. 1, February 1996.

[13] Olga Kurasova, Virginijus Marcinkevicius, Viktor Medvedev, Aurimas Rapecka, and Pavel Stefanovic, "Strategies for Big Data Clustering," in 2014 IEEE 26th International Conference on Tools with Artificial Intelligence, Limassol, Cyprus.
[14] Sabhia Firdaus and Md. Ashraf Uddin, "A Survey on Clustering Algorithms and Complexity Analysis," IJCSI International Journal of Computer Science Issues, Volume 12, Issue 2, March 2015.
[15] J. B. MacQueen, "Some Methods for classification and Analysis of Multivariate Observations," in Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability. Berkeley, University of California Press, 1:281-297.
[16] D. Kirk and W. Hwu, "Programming Massively Parallel Processors: A Hands-on Approach," Morgan Kaufmann, 2010.
[17] Henry Kasim, Verdi March, Rita Zhang and Simon See, "Survey on Parallel Programming Model," in IFIP International Federation for Information Processing 2008.
[18] Ananth Grama, Anshul Gupta, George Karypis and Vipin Kumar, "Introduction to Parallel Computing," Addison Wesley, 2003.