Parallel Implementation of Shared Nearest Neighbor Clustering Algorithm

Nikhilesh Meghwal, Suguna M
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore, India
nikhileshmeghwal42@gmail.com, sugu2803@gmail.com

***Abstract*—Shared Nearest Neighbor (SNN) is a density-based algorithm which is efficient in clustering of very large data. But the quadratic time-complexity of the algorithm and the large memory space requirement to accommodate the data on a single machine demands for a parallel implementation of the algorithm. In this project we have implemented two different parallelized approaches of SNN algorithm, one on distributed systems using MPI and other is on GPU using CUDA. We will show that we can achieve significant speedups for MPI and more than 30X of speedup on GPU compared with serial version of SNN. also discuss about the hybrid of CPU and GPU implementation of SNN algorithm for large data set.**

I. Introduction

With the advent of the big data era, it becomes very crucial to identify and group together similar objects to represent information in a compact form. This leads us to cluster the data in some sense, with an aim to group together similar data instances and discover new patterns and hidden relationships among them. In fact, clustering has been found useful in many different applications such as pattern recognition, decision making, image segmentation and document retrieval. As a result, several data clustering algorithms have been proposed in many different contexts, eg; based on proximity to centers (like k-means), connectivity-based(hierarchical clustering), distribution based (Gaussian mixture models) and density based (DBSCAN). Shared Nearest Neighbor(SNN) [1] is a density-based clustering algorithm which identifies the clusters based on the number of densely connected neighbors. For example, the consider the data shown in fig 1 [2]. The t4.8k Chameleon data has dense and complex shaped clusters. The quadratic time-complexity of the algorithm and the large memory space requirement to accommodate the data on a single machine demands for a parallel implementation of the algorithm.This problem motivated us to proceed with this project. In this project we have implemented two different parallelization approaches of SNN algorithm.

Section III-B deals with the parallel implementation on distributed systems using MPI. Section III-C deals with Parallel Implementation on GPU using CUDA.

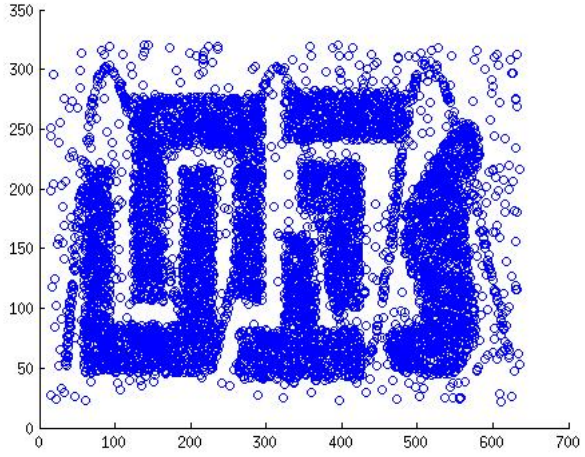

Fig. 1. Chameleon dataset: a challenging example for clustering

II. Related Work

The SNN algorithm has been discussed in detail in [1] and [3]. The notion of similarity in the SNN algorithm is based on the number of neighbors that two points share. That is, the similarity between two points is confirmed by their common (shared) near neighbors. If point A is close to point B and if they are both close to a set of points C then we can say that A and B are close with greater confidence since their similarity is confirmed by the points in set C. This idea of shared nearest neighbor was first introduced by [4]. A similar idea was later presented in [5].Another popular density based algorithm is the G-DBSCAN [6] which is divided into two stages: graph construction and identification of clusters through breadth first search. The parallelization of both stages on GPU achieved excellent speed-ups (82x for graph construction and 21x for the clusters identification).

The first step of the SNN algorithm involves finding of the k-Nearest neighbors. In [7] a brute-force method using GPU programming (through NVIDIA CUDA) has been implemented to find k-NN and compared its performance to several CPU-based implementations. The brute-force method has two steps: the distance computation and the sorting. It is showed that the use of the NVIDIA CUDA API accelerates the kNN search by up to a factor of 400 compared to a brute force CPU-based implementation. In [8] the author proposes CUDA

implementations of pairwise distances. The number of bank conflicts in the shared memory access are reduced by a factor of 16 and dramatically speed up the calculation. This method achieves a 20 to 44 times speedup than a CPU implementation.

III. Methodology

A. SNN algorithm

Let us assume the input data is an $n \times m$ matrix, where n is the number of data points and m is the number of dimensions. Parameters:
k= number of nearest neighbor to be identified for each point.
Eps = the density threshold that establish the minimum number of neighbors two points should share to be considered close to each other.
$MinPts$ = the minimum density that a point should have to be considered a core point. Clusters are defined using the found core points.

1) Create a distance matrix whose $(i, j)^{th}$ element is the distance between nodes i and j.
2) Find the first k minimum distances along each row of this distance matrix to get the k-Nearest neighbor to each data point.
3) Establish the SNN density of each point. The SNN density is given by the number of nearest neighbors that share Eps or more neighbors.
4) Identify the core points of the data set. Each point that has a SNN density greater or equal to $MinPts$ is considered a core point.
5) Build clusters from core points. Two core points are allocated to the same cluster if they share Eps or more neighbors with each other.
6) Assign the remaining points to clusters. All non-core and non-noise points are assigned to the nearest cluster.

We can see that the time complexity of the above algorithm is $\mathcal{O}(n^2)$ but the steps are highly parallelizable.

B. MPI implementation

The data ($n \times m$, n points and m dimensions) is divided among p processors i.e each processors will get $\frac{n}{p}$ number of data points. Each processors has to communicate its data with other processors to compute its part of distance matrix ($\frac{n}{p} \times n$). KNN matrix ($\frac{n}{p} \times k$) is computed from the distance matrix to get k-nearest neighbors. This requires sorting of the distance matrix up to first k elements. We use both insertion and selection sort but they give same performance, $\mathcal{O}(n \times k)$ for the first k-nearest neighbors. KNN has to be gathered by all the processors to compute the SNN matrix $\frac{n}{p} \times n$ i.e. similarity between for each two points. Core points are recognized by calculating SNN densities of the points. Root processor will cluster the core points and this information is broadcasted. The final step of assigning the non-core and non-noise points are done by each processor for its respective data points. In this implementation even though large data has to be communicated among the processors, the computation time is greater than communication time and hence we get good speedups.

C. CUDA implementation

More than 80% of the time in the serial code is taken in Computing KNN and SNN matrices. These steps are highly parallelized in CUDA if we have entire data on the GPU. This is possible for smaller data set. The KNN algorithm in [8] works for datasets having dimensions in multiples of 16. This CUDA code uses one thread for each entry for KNN matrix. Thus there are n^2 threads. The threads are organized into 16×16 two-dimensional blocks, and the blocks are organized into an $\frac{n}{16} \times \frac{n}{16}$ two-dimensional grid. The 256 threads in each block first load the two 16×16 sub-matrices of Data matrix into shared memory Xs and Ys. After the threads are synchronized, each of them calculates and accumulates its own partial Euclidean distance in the variable sum. Then the threads need to be synchronized again before proceeding to the next pair of 16 by 16 sub-matrices. Hence to compute one element of the KNN matrix, each block has to load the sub-matrices $\frac{m}{16}$ times. A subtle yet very important point in the code is that the array Xs in the shared memory is transpose of its image in the matrix KNN. Through the transpose of Xs, they reduce the number of bank conflicts in the shared memory access by a factor of 16 and dramatically speed up the calculation.
We have modified this algorithm for any dimension. First, multiple of 16 out of m dimensions are loaded as mentioned above and in the last iteration, $m - \left\lfloor \frac{m}{16} \right\rfloor$ values of the KNN row are loaded. Hence in the last iterations only $\left(m - \left\lfloor \frac{m}{16} \right\rfloor\right) \times \left(m - \left\lfloor \frac{m}{16} \right\rfloor\right)$ number of threads will execute the remaining partial Euclidean distance.
Sorting in CUDA: Each thread sorts each row of distance matrix and return K-nearest neighbors indices. Hence we require n threads for n data points. Insertion and Selection sort gives the same performance having complexity $\mathcal{O}(n \times k)$.
SNN matrix computation in CUDA: Each thread computes one row of SNN matrix i.e each thread is assigned to one point which computes the number of shared nearest neighbors with all other points.
Remaining steps of the SNN algorithm is executed in the CPU.

IV. Experiments and Results

The experiment done on the Chameleon Data sets [2] Experiments are done by setting different values for hyperparameters k, Eps, $MinPts$. The results are shown in Table IV. The codes for MPI and CUDA implementation are run on the datasets. Below are the results of the experiment on the chameleon data set t4.8k having the size (n = 8000, m = 2).

Fig 2 and 3 are the plots of execution time and speedup vs number of processors for the MPI implementation. It is observed that speedup is significantly good.
The execution time taken by the CUDA imlpementation on the same dataset is 8.9 seconds, thus CUDA implementation has speedup of more than 30x as compared to serial code.

K	Epsilon	MinPoint	# Core Points	#clusters
15	7	12	7679	450
20	10	15	7687	362
20	15	17	253	32
20	15	19	23	7

TABLE I
RESULTS OF SNN ALGORITHM ON CHAMELEON DATASET T4.8K

Processors	Time	Speedup
1	283.25	1
2	115.37	2.45
4	66.77	4.44
8	42.9	6.6
10	33.16	8.54
12	30.15	9.39
16	26.69	10.6
20	23.06	12.29
32	21.56	13.13

TABLE II
EXECUTION TIME AND SPEEDUPS OF MPI IMPLEMENTATION

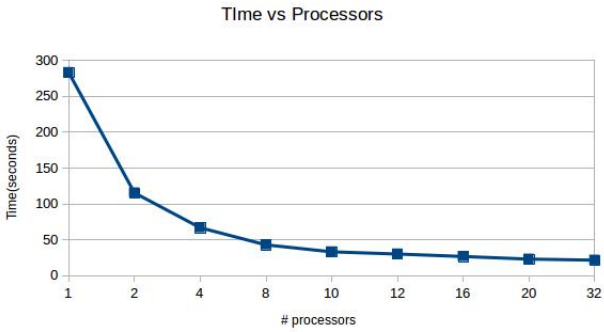

Fig. 2. Computation time of MPI implementation with respect to different number of processors used

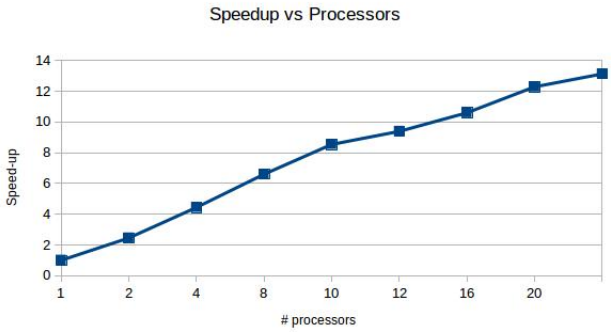

Fig. 3. Speedup of MPI implementation with respect to different number of processors used

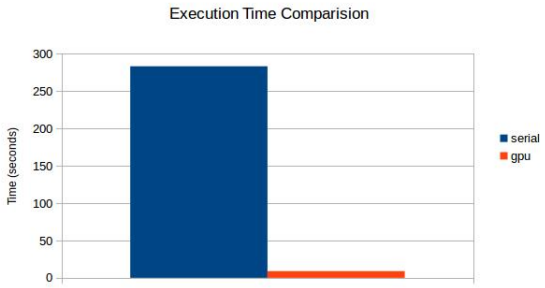

Fig. 4. Comparison of execution times between MPI and CUDA implementation

V. CONCLUSIONS AND FUTURE WORK

A. Conclusion

The parallel version of SNN algorithm is implemented in MPI. For small data set MPI gives very good Speedup. Various steps of SNN algorithm implementation in CUDA is discussed. CUDA code performs more than 30x speedup compared to the serial code.Thus we have shown that SNN algorithm can be highly parallelized with very good speedups. But if the data size is too large then it is not possible to compute KNN or SNN on single GPU since it has memory constraints.

B. Future Work

The Application of SNN is in Spatial data which is very huge, we need to use multiple GPU's. For large data set we would suggest Hybrid of CPU and GPU implementation for SNN algorithm.
Also note that both KNN and SNN matrices are symmetric and thus we can exploit this property to reduce the number of computations as well as the memory usage of GPU. However, since now we have to deal with triangular matrices, there could be load imbalance problem.
The future work of this project can be extended to find better implementation for the above two problems.

ACKNOWLEDGMENT

We would like to thank Prof. Sathish Vadhiyar for giving us this opportunity to explore and learn more about parallel programming concepts and apply them to a practical problem. We are also grateful to the SERC department for providing the system facilities.

REFERENCES

[1] L. Ertoz, M. Steinbach, and V. Kumar, "A new shared nearest neighbor clustering algorithm and its applications," in *Workshop on Clustering High Dimensional Data and its Applications at 2nd SIAM International Conference on Data Mining*, 2002, pp. 105–115.
[2] G. Karypis, E.-H. Han, and V. Kumar, "Chameleon: Hierarchical clustering using dynamic modeling," *Computer*, vol. 32, no. 8, pp. 68–75, 1999.
[3] F. Mendes, M. Santos, and J. Moura-Pires, "Dynamic analytics for spatial data with an incremental clustering approach," in *Data Mining Workshops (ICDMW), 2013 IEEE 13th International Conference on*, Dec 2013, pp. 552–559.
[4] R. A. Jarvis and E. A. Patrick, "Clustering using a similarity measure based on shared near neighbors," *Computers, IEEE Transactions on*, vol. 100, no. 11, pp. 1025–1034, 1973.

[5] S. Guha, R. Rastogi, and K. Shim, "Rock: A robust clustering algorithm for categorical attributes," in *Data Engineering, 1999. Proceedings., 15th International Conference on*. IEEE, 1999, pp. 512–521.

[6] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha, "G-dbscan: A gpu accelerated algorithm for density-based clustering," *Procedia Computer Science*, vol. 18, pp. 369–378, 2013.

[7] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*. IEEE, 2008, pp. 1–6.

[8] D. L. M. O. Darjen Chang, Nathaniel A. Jones, "Compute pairwise euclidean distances of data points with gpus," in *Proceedings of the IASTED International Symposium Computational Biology and Bioinformatics*, 2008, pp. 278–283.