To The University of Wyoming:
The members of the Committee approve the Plan B project of Dharani Koratala presented on [09/27/2016] .

Dr. Craig Douglas, Committe Chair

Dr. John Hitchcock, Co-Chair

Dr. Thomas Bailey, Professor

Dr. Tim Considine, External member

Serial and Parallel Clustering Algorithms in Data Mining

by

Dharani Koratala

A Plan B project submitted to the University of Wyoming in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Laramie, Wyoming
September 2016

Abstract

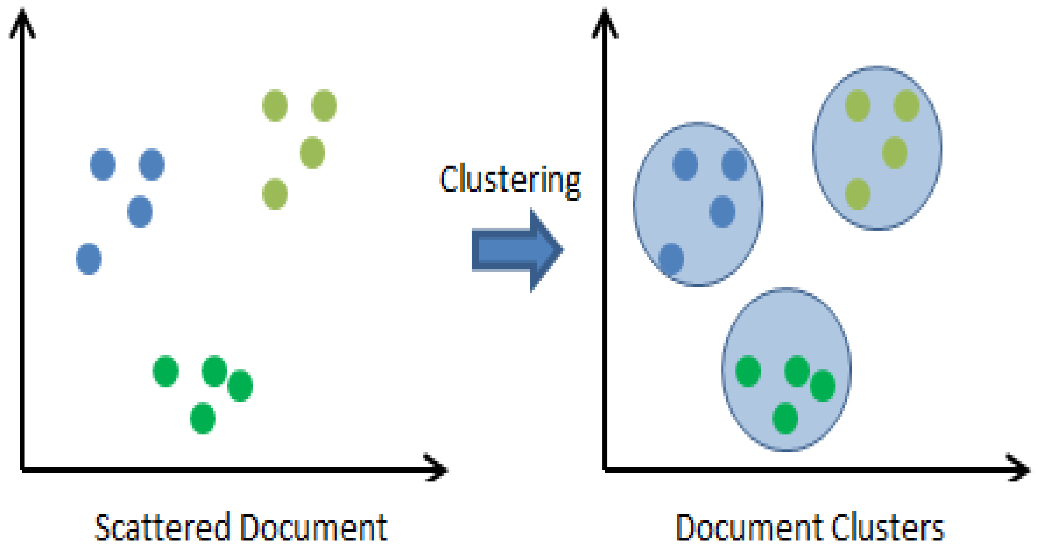
The grouping of input data sets into subsets is called ’clustering’ and within each cluster the elements are similar. In general, clustering is an unsupervised learning task as very little or no prior knowledge of the data is given to people except the input data sets. These tasks are used in many fields resulting in the development of many clustering algorithms. The task of clustering however, is computationally expensive as many of the algorithms require iterative or recursive procedures and most real-life data are high-dimensional. This resulted in the parallelization of clustering algorithms and various parallel clustering algorithms have been implemented and applied to many applications. This paper presents the implementation of the most influential partitioning and density based clustering algorithms: k-means, DBSCAN, and OPTICS. These algorithms were implemented in serial and parallel versions. The parallel k-means algorithm uses the technique of parallelizing data using threads. The parallel DBSCAN and parallel OPTICS algorithms use the techniques of implementing graph algorithmic concepts (i.e., disjoint set data structures). The input to the algorithms are the datasets with variable sizes and the output describes the efficiency of using these parallel clustering algorithms. Furthermore, to calculate the efficiency of the algorithms, testing is performed for both the sequential and parallel versions using several threads. With each algorithm, we provide a description of the algorithm, discuss its impact, and efficiency based on its computational time. Each algorithm deals with different techniques in showing how the data are clustered. We show how to automatically and efficiently extract not only the ‘traditional’ clustering information (e.g., representative points, shaped clusters), but also the arbitrary structures independent of the size of data sets.

Table of Contents

Chapter 1
Introduction

1.1 Clustering and its applications

Clustering is a technique that assigns data to points in some space and then groups the nearby points into clusters based on some distance measure. Hence the definition of clustering could be grouping of data objects into similar groups of data sets called clusters. A cluster is therefore a collection of objects which are "similar" between them and the "dissimilar" objects belong to other clusters. This can be shown with a simple example.



Here we identify 3 clusters that the data can be divided. The similarity criterion is distance - two or more objects belong to the same cluster if they are close according to a given distance. The goal of clustering is to determine the intrinsic grouping in a set of unlabeled data.

Clustering algorithms can be applied in many fields, for instance pattern recognition and anomalies in cyber security, athletic recruiting for data analysis to check the consistency of the players, bioinformatics in finding similar gene types, classifying gene data sets and comparison of gene sequences, transaction analysis in the finance and banking sectors. These are also very useful in the field of petroleum engineering to automatically generate reports on the state of oil fields. And in this project we propose three most useful clustering algorithms: k-means, DBSCAN and OPTICS.

1.2 Types of Parallel Programming

Here we describe MPI [3] and OpenMP [3] for programming shared memory and distributed memory machines.

***MPI*:** The Message Passing Interface was developed to facilitate portable programming for distributed-memory architectures (MPPs), where multiple processes execute independently and communicate data as needed by exchanging messages. MPI is primarily used for parallelization across multiple nodes. Here every thread can access only its own memory. It can be difficult to create a single program version that will run efficiently on many different systems since the relative cost of communicating data and performing computations varies from one system to another. Proper guidance must be taken to avoid certain errors, particularly deadlock where two or more processes each wait in perpetuity for the other to send a message.

OpenMP: OpenMP is a shared memory (or shared address space) programming model. This model assumes, as its name implies, that programs will be executed on one or more processors that share some or all of the available memory. Shared memory programs are typically executed by multiple independent threads. The threads share data, but may also have private data. Shared memory approaches to parallel programming must provide, in addition to a normal range of instructions, a means for starting up threads, assigning work to them, and coordinating their accesses to the shared data, including ensuring that certain operations are performed by only one thread at a time.

MPI + OpenMP: The combination of MPI and OpenMP parallelization is used in cases where OpenMP is used inside a single node, and MPI across different nodes.

Depending on how the data is classified and the number of processors used, we can say that MPI can be used for distributed programming models and OpenMP can be used for shared memory programming models.

Chapter 2

Serial and Parallel Algorithms

In this chapter we deal with several serial and parallel clustering algorithms: k-means, DBSCAN and OPTICS.

2.1 k-means and parallel k-means

2.1.1 k-means algorithm

There are four steps:

1. Initialization
2. Distance calculation
3. Centroid recalculation
4. Convergence.

Given a set of n instances $I = \{m_i \mid 0 < i < n\}$, a set of k initial cluster centroids $C = \{c_j \mid 0 < j < k\}$, and a distance function $D(m_i, c_j)$. Here n is the number of instances in the dataset, d is the number of attributes for each instance, k is the number of clusters, and z is the total number of instances assigned to a thread during parallel execution.

• Initialization: Select k starting points $C = \{c_j \mid 0 < j < k\}$ from the instance space, which are the initial cluster centers.

• Distance Calculation: Compute the distance to the nearest cluster center.

• Centroid Recalculation: For the given cluster centers, a set of clusters with minimum variance is found. However, at this point centroids, have been selected in an arbitrary manner. So they cannot be the exact centers of the clusters. As a result, in this step, the centers of clusters are recalculated. This is done by calculating the mean of each attribute for all instances within the cluster.

• Convergence: The clusters found after the above three steps are actually not optimized as the cluster centroids are recalculated and shifted on the instance space. Convergence occurs when the shifting falls below a given threshold.

Algorithm [4]:

```
 1: $minD \leftarrow 0$
 2: $OldminD \leftarrow \infty$
 3: Let Instance set be INST = $\{i_x\}_{0<x<n}$
 4: Select initial centroids CENT = $\{c_y\}_{0<y<k}$ as the first k instances in I
 5: Let DIST = $\{d_z\}_{0<z<P}$ be the distances between $i_x$ and $c_z$
 6: procedure SEQUENTIAL_K_MEANS
 7:     while minD < OldminD do
 8:         OldminD ← minD
 9:         for $i \leftarrow 1, n$ do
10:             for $j \leftarrow 1, k$ do
11:                 $Dist_j \leftarrow$ distance($INST_i$, $CENT_j$);
12:             $minD \leftarrow$ minimum($DIST_{j,0<j<k}$);
13:         for all instances $I_i \subset INST$ assigned to cluster $i$, $0 < i < k$ do
```

```
14:                 $c_i = \sum I_i$
```

n: The number of instances in the dataset.
d: The number of attributes for each instance.
k: The number of clusters.
z: The total number of instances assigned to a thread during parallel execution.

The algorithm proceeds as follows:

1. First, it randomly selects k objects from the whole objects which represent initial cluster centers.
2. Each remaining object is assigned to the cluster to which it is the most similar to, based on the distance between the object and the cluster center.
3. The new mean for each cluster is then calculated. This process iterates until the criterion function converges. In the k-means algorithm, the most important calculation is the calculation of distances.

In each iteration, it would require a total of nk distance computations where n is the number of objects and k is the number of clusters being created. It is obviously that the distance computations between one object with the centers is irrelevant to the distance computations between other objects with the corresponding centers. Therefore, distance computations between different objects with centers can be executed in parallel.

4. In each iteration, the new centers, which are used in the next iteration, should be updated. Hence, the iterative procedures must be executed serially.

2.1.2 Parallel k-means algorithm

There are four steps:

1. Initialization
2. Distance calculation
3. Centroid recalculation
4. Convergence.

Algorithm [5]:

```
 1: minD ← 0
 2: OldminD ← ∞
 3: Let $minD_p$ be the partial minD computed by the processor p
 4: Let Instance set be INST = $\{i_x\}_{0<x<n}$
 5: Select initial centroids CENT = $\{c_y\}_{0<y<k}$ as the first k instances in I
 6: Let DIST = $\{d_z\}_{0<z<P}$ be the distances between $i_x$ and $c_z$
 7: procedure PARALLEL_K_MEANS
 8:     while minD < OldminD do
 9:         OldminD ← minD
10:         for i ← 1, P do
11:             Assign n/P instances to processor i
12:             Assign k/P centroids to processor i
```

```
13:     for each Processor $p \in P$ do
14:         for $i \leftarrow (i - 1) * (n/P), i * (n/P) - 1$ do
15:             for $j \leftarrow 1, k$ do
16:                 $Dist_{p,j} \leftarrow$ distance($INST_i$, $CENT_j$);
17:             $minD_P \leftarrow$ minimum($DIST_{j,0<j<k}$);
18:     for each Processor $p \in P$ do
19:         for all instances $I_i \subset INST$ assigned to cluster $i$, $0 < i < k$ do
20:             $c_i = \sum I_i$ / number of instances assigned to cluster $i$
21:     WAIT ALL PROCESSORS and LOCK MEMORY minD
22:     for each Processor $p \in P$ do
23:         minD += $minD_P$
24:     WAIT ALL PROCESSORS and UNLOCK MEMORY minD
```

n: The number of instances in the dataset.
d: The number of attributes for each instance.
k: The number of clusters.
P: The total number of processors.
z: The total number of instances assigned to a thread during parallel execution.

1. The first step of parallel k-means [5] is the initialization of centroids. Each of the *P* processor can be assigned the task of selecting *k*/*P* centroids. Hence a set of centroids will be decided among processors.
2. The second step of the parallel k-means algorithm is each processor computes a distance for all instances assigned to itself and all cluster centroids. This step can be parallelized by assigning *n*/*P* instances to each processor, where each processors compute the closest centroid of each instance in a parallel fashion.

After concluding step 2, each instance is assigned to a cluster according to their closest distance to each cluster centroid and membership information of all instances for each cluster is established.
3. In this step the parallel k-means algorithm computes new cluster centroids by calculating the minimum distance *mind* and the task of calculating the new cluster centroids must be done in parallel. Thus after each iteration of centroid recalculation, the processors should be synchronized.

In this work, in order to maximize the parallelization, the data is divided into computational batches called threads, where for each batch a minimum distance *minD* is calculated in parallel. After all threads finish computation, parallel tasks are synchronized in order to compute new cluster centroids. Cluster centroids are assigned to one of the existing threads and computations are carried out in parallel.
4. Convergence is done sequentially.

2.2 DBSCAN and parallel DBSCAN

2.2.1 DBSCAN algorithm

Density Based Spatial Clustering of Applications with noise is the algorithm that deals with the spatial data clusters with noise. In DBSCAN [6], the density associated with a point is obtained by counting the number of points in a region of specified radius *eps* around the point. DBSCAN algorithm has the ability to discover clusters with arbitrary shape such as linear, concave, oval, etc. Furthermore, in contrast to some clustering algorithms, it does not require the predetermination of the number of clusters. DBSCAN has been proven in its ability of processing very large databases. There are five steps:

1. Select an arbitrary point *q*
2. Retrieve all points density reachable from *q* w.r.t. *eps* and *minpts*.
3. If *q* is a core point, a cluster is formed.
4. If *q* is a border point, no points are density reachable from *q* and DBSCAN visits the next point of the database.
5. Continue the process until all the points have been processed.

Algorithm [6]: The DBSCAN algorithm. Input: A set of points *X*, distance threshold *eps*, and the minimum number of points required to form a cluster, *minpts*. Output: A set of clusters.

```
 1: procedure DBSCAN($X$, $eps$, $minpts$)
 2:     for each unvisited point $x \in X$ do
 3:         mark $x$ as visited
 4:         $N \leftarrow$ GETNEIGHBORS($x$, $eps$)
 5:         if $|N| < minpts$ then
 6:             mark $x$ as noise
 7:         else
 8:             $C \leftarrow \{ x \}$
 9:             for each point $x^* \in N$ do
10:                 $N \leftarrow N \setminus x^*$
11:                 if $x^*$ is not visited then
12:                     mark $x^*$ as visited
13:                     $N^* \leftarrow$ GETNEIGHBORS($x^*$, $eps$)
14:                     if $|N^*| \geq minpts$ then
15:                         $N \leftarrow N \cup N^*$
16:                 if $x^*$ is not yet member of any cluster then
17:                     $C \leftarrow C \cup \{ x^* \}$
```

eps: distance threshold
minpts: minimum number of points required to form a cluster
X: a set of points
N: eps-neighborhood
C: new cluster

2.2.2 Parallel DBSCAN algorithm

Although DBSCAN is popular for its efficiency in discovering arbitrary shaped clusters and eliminating noise data, parallelization of DBSCAN is challenging as it exhibits sequentiality in data access. Therefore, a new parallel DBSCAN algorithm (PDSDBSCAN) [7] using graph algorithmic concepts is more efficient in dealing with arbitrary shaped clusters. More specifically, we use the disjoint set data structure to break the sequentiality of DBSCAN. We use a tree based, bottom up approach to construct the clusters.

Algorithm [7]: The parallel DBSCAN algorithm on a shared memory computer (PDSDBSCAN). Input: A set of points X, distance *eps*, and the minimum number of points required to form a cluster, *minpts*. Let X be divided into P equal disjoint partitions $\{ X_1, X_2, \dots, X_P\}$. For each thread t, Y_t denotes a set of pairs of points (x, x^*) such that $x \in X_t$ and $x^* \notin X_t$. Output: A set of clusters.

```
 1: procedure PDSDBSCAN( $X$, $eps$, $minpts$ )
 2:    for $t$ = 1 to $P$ in parallel do .              # Stage: Local comp. (Line 2-18)
 3:        for each point $x \in X_t$ do
 4:            $p(x) \leftarrow x$
 5:        $Y_t \leftarrow \emptyset$;
 6:        for each point $x \in X_t$ do
 7:            $N \leftarrow$ GETNEIGHBORS($x$, $eps$)
 8:            if $|N| \geq minpts$ then
 9:                mark $x$ as a core point
10:                for each point $x^* \in N$ do
11:                    if $x^* \in X_t$ then
12:                        if $x^*$ is a core point then
13:                            UNION($x$, $x^*$)
14:                        else if $x^* \notin$ any cluster then
15:                            mark $x^*$ as member of a cluster
16:                            UNION($x$, $x^*$)
17:                    else
18:                        $Y_t \leftarrow Y_t \cup \{ x, x^* \}$
19:    for $t$ = 1 to $P$ in parallel do .              # Stage: Merging (Line 19-25)
20:        for each $(x, x^*) \in Y_t$ do
21:            if $x^*$ is a core point then
22:                UNIONUSINGLOCK($x$, $x^*$)
23:            else if $x^* \notin$ any cluster then        # Line 23-24 are atomic
24:                mark $x^*$ as member of a cluster
25:                UNIONUSINGLOCK($x$, $x^*$)
```

eps: distance threshold
minpts: minimum number of points required to form a cluster
X: a set of points
N: eps-neighborhood
C: new cluster
Y_t: set of pair of points (x, x^*)

The use of the disjoint set data structure in density based clustering works as a primary tool in breaking the access order of points while computing the clusters. Here we present the disjoint set based parallel DBSCAN algorithm [7]. The key idea of the algorithm is that each process core first runs a sequential DBSCAN algorithm on its local data points and computes local clusters in parallel without requiring any communication. After this we merge the local clusters to obtain the final clusters. This is also performed in parallel and is quite opposite to the master-slave approach. Moreover, the merging of two trees only requires changing the parent pointer of one root to the other one. Since the entire computation is performed in parallel, substantial scalability and speedup have been obtained.

Algorithm [7]: Merging the trees containing x and x^* with UNION using lock.

```
 1: procedure UNIONUSINGLOCK($x$, $x^*$)
 2:     while $q(x) \neq q(x^*)$ do
 3:         if $q(x) < q(y)$ then
 4:             if $x = q(x)$ then
 5:                 LOCK($q(x)$)
 6:                 if $x = q(x)$ then
 7:                     $q(x) \leftarrow q(x^*)$
 8:                 UNLOCK($q(x)$)
 9:             $x = q(x)$
10:         else
11:             if $x^* = q(x^*)$ then
12:                 LOCK($q(x^*)$)
13:                 if $x^* = q(x^*)$ then
14:                     $q(x^*) \leftarrow q(x)$
15:                 UNLOCK($q(x^*)$)
16:             $x^* = q(x^*)$
```

X: a set of points
N: eps-neighborhood
Y_t: set of pair of points (x, x^*)
q: parent pointer

2.3 OPTICS and parallel OPTICS

2.3.1 OPTICS algorithm

OPTICS is a hierarchical density based data clustering algorithm that discovers arbitrary shaped clusters and eliminates noise using a reachability distance option. Ordering points to identify the clustering structure (OPTICS) [8] is an algorithm for finding density based clusters in spatial data. It is similar to DBSCAN, but solves one of DBSCAN's major weaknesses: the problem of detecting meaningful clusters in data of varying density. In order to do so, the points of the database are (linearly) ordered such that points which are spatially closest become neighbors in the ordering. Additionally, a core distance and reachability distance are computed efficiently. A priority queue is used instead of maintaining a set.

Algorithm [8]: The OPTICS algorithm. Input: A set of points *X* and the input parameters, generating distance *ε*, and the minimum number of points required to form a cluster *minpts*. Output: An order of points *O*, the core distances, and the reachability distances.

```
 1: procedure OPTICS( $X$, $\varepsilon$, $minpts$, $O$ )
 2:     $pos \leftarrow 0$
 3:     for each unprocessed point $x \in X$ do
 4:         mark $x$ as processed
 5:         $N \leftarrow$ GETNEIGHBORS($x$, $\varepsilon$)
 6:         SETCOREDISTANCE( $x$, $N$, $\varepsilon$, $minpts$ )
 7:         O[$pos$] $\leftarrow x$; $pos \leftarrow pos + 1$
 8:         RD[$x$] $\leftarrow$ NULL
 9:         if CD[$x$] $\neq$ NULL then
10:             UPDATE($x$, $N$, $Q$)
11:             while $Q \neq$ empty do
12:                 $y \leftarrow$ EXTRACTMIN($Q$)
13:                 mark $y$ as processed
14:                 $N^* \leftarrow$ GETNEIGHBORS( $y$, $\varepsilon$ )
15:                 SETCOREDISTANCE( $y$, $N^*$, $\varepsilon$, $minpts$)
16:                 O[$pos$] $\leftarrow y$; $pos \leftarrow pos + 1$
17:                 if CD[$y$] $\neq$ NULL then
18:                     UPDATE( $y$, $N^*$, $Q$ )
```

eps: distance threshold
minpts: minimum number of points required to form a cluster
X: a set of points
N: eps-neighborhood
Q: priority queue
CD: core distance
RD: reachability distance
O: order of points

Algorithm [8]: The UPDATE function. Input: A point *x*, its neighbors, *N*, and a priority queue, *Q*. Each element in *Q* stores two values, a point x^* and its best reachability distance so far.

```
1: procedure UPDATE( $x$, $N$, $Q$ )
2:     for each unprocessed point $x^* \in N$ do
3:         $new$D = MAX(CD[$x$], DISTANCE($x^*$, $x$))
4:         if RD[$x^*$] = NULL then
5:             RD[$x^*$] $\leftarrow new$D
6:             INSERT($Q$, $x^*$, $new$D)
7:         else if $new$D < RD[$x^*$] then
8:             RD[$x^*$] $\leftarrow new$D
9:             DECREASE($Q$, $x^*$, $new$D)
```

X: a set of points
N: eps-neighborhood
Q: priority queue
CD: core distance
RD: reachability distance
O: order of points

2.3.2 Parallel OPTICS algorithm

Parallelizing OPTICS is considered challenging as the algorithm which exhibits sequentiality in data access. We present a scalable parallel OPTICS algorithm (POPTICS) [9] designed using graph algorithmic concepts. To break the data access sequentiality, POPTICS uses the similarities between the OPTICS algorithm and Prim's Minimum Spanning Tree [10] algorithm. Additionally, we use the disjoint set data structure to achieve a high parallelism for distributed cluster extraction.

Algorithm [9]: The parallel OPTICS algorithm on a shared memory computer (POPTICSS).
Input: A set of points X and the input parameters ε, and *minpts*. Let X be divided into P equal disjoint partitions X_1, X_2, X_3, . . . , X_P. Output: The Minimum Spanning Trees T.

```
 1: procedure POPTICSS( $X, \varepsilon, minpts, T$ )
 2:     for $t$ = 1 to $p$ in parallel do        # Stage: Local computation
 3:         for each unprocessed point $x \in X_t$ do
 4:             mark $x$ as processed
 5:             $N \leftarrow$ GETNEIGHBORS($x, \varepsilon$)
 6:             SETCOREDISTANCE( $x, N, \varepsilon, minpts$ )
 7:             if CD[$x$] $\neq$ NULL then
 8:                 MODUPDATE( $x, N, P_t$ )
 9:                 while Pt $\neq$ empty do
10:                     ($u, v, w$) $\leftarrow$ EXTRACTMIN( $P_t$ )
11:                     $Q \leftarrow$ INSERT($u, v, w$) in critical
12:                     if $u \in X_t$ then
13:                         mark $u$ as processed
14:                         $N^* \leftarrow$ GETNEIGHBORS( $u, \varepsilon$ )
15:                         SETCOREDISTANCE( $u, N^*, \varepsilon, minpts$ )
16:                         if CD[$u$] $\neq$ NULL then
17:                             MODUPDATE( $u, N, P_t$ )
18:                             for each point $x \in X$ in parallel do  #Stage: Merging
19:                                 PAR($x$) $\leftarrow x$
20:                                 while $Q \neq$ empty do
21:                                     ( $u, v, w$) $\leftarrow$ EXTRACTMIN( $Q$ )
22:                                     if UNION( $u, v$ ) = TRUE then
23:                                         $T \leftarrow T \cup (u, v, w)$
```

eps: distance threshold
minpts: minimum number of points required to form a cluster
X: a set of points
N: eps-neighborhood
Q: priority queue
CD: core distance
RD: reachability distance
O: order of points
T: minimum spanning tree

Hence, this chapter dealt with the k-means, DBSCAN, and OPTICS algorithms.

Chapter 3

Numerical Experiments

The serial algorithms are implemented in MATLAB using internal MATLAB functions. The parallel k-means algorithm was adapted from the reference [5] using MPI with C++. The parallel DBSCAN and parallel OPTICS algorithms were adapted from the references [7] [9] using Open MP with C++. The comparisons for various data sets are done with respect to several threads for parallel k-means, parallel DBSCAN, and parallel OPTICS algorithms.

For this project, a sample of random datasets were generated. The data sets could be either plain text files or binary files. Data is stored in a serial fashion and the clustering of the data objects is performed with various clustering algorithms and the computational time is calculated for serial and parallel algorithms. This is done for several threads to observe the time used. The point in choosing large data objects is that the numerical experiments are accurate when performed with huge data.

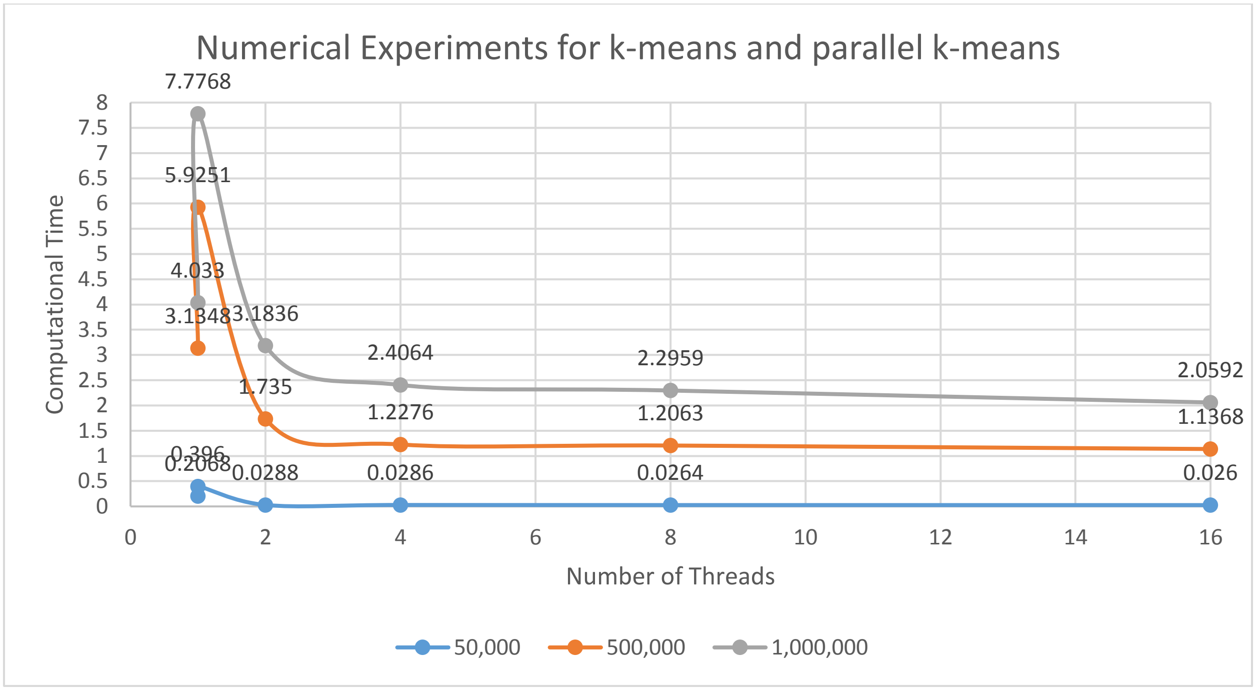
Testing is performed for 50,000, 500,000, and 1,000,000 data objects as well for all of the clustering algorithms with several threads. For a sample with 1 thread, 2 threads, 4 threads, 8 threads, and 16 threads, the parallelization would be done in all the cases. I have also tested for 5,000,000 data objects, but it did not make any change compared to 1,000,000 data objects. The time taken to perform all of the operations was reduced when the number of threads increase as the threads are distributed in a parallel manner. The time taken to execute the whole process with respect to several threads and data sets are present in this chapter.

Numerical Experiments for k-means and parallel k-means algorithms:

Numerical Experiments for various data sets

Number of threads	50,000 data objects	500,000 data objects	1,000,000 data objects
Sequential	0.2068 sec	3.1348 sec	4.0330 sec
1	0.3960 sec	5.9251 sec	7.7768 sec
2	0.0288 sec	1.735 sec	3.1836 sec
4	0.0286 sec	1.2276 sec	2.4064 sec
8	0.0264 sec	1.2063 sec	2.2959 sec
16	0.0260 sec	1.1368 sec	2.0592 sec

Table 1: Numerical Experiments for k-means and parallel k-means



Graph 1: Numerical Experiments for k-means and parallel k-means

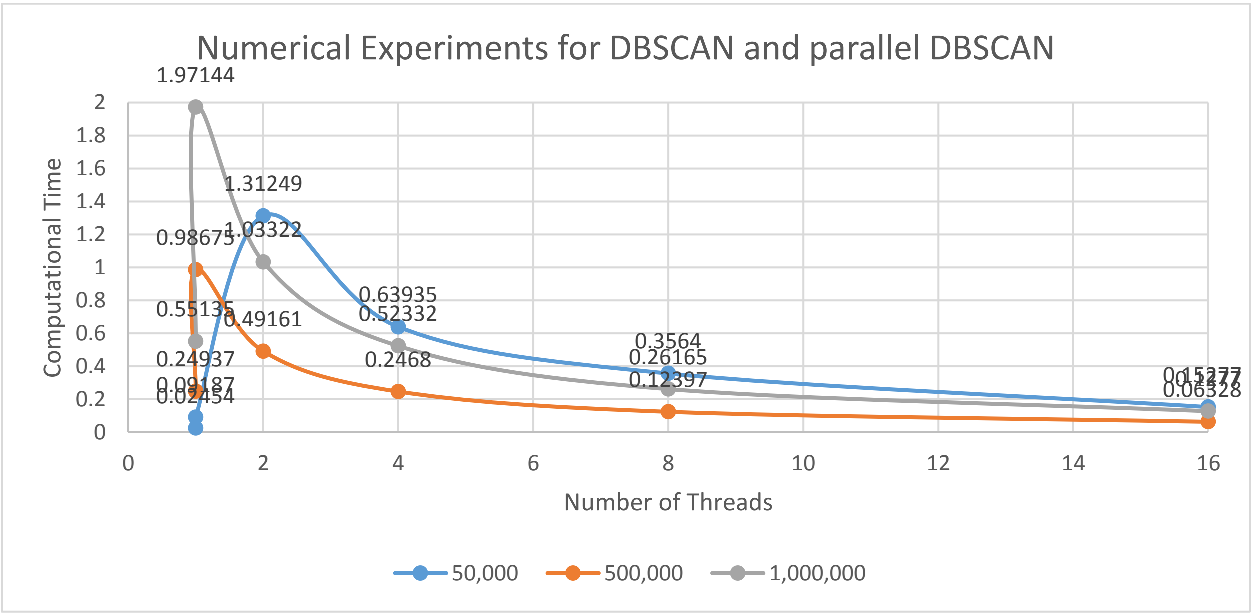
Table 1 here shows the computational time in running the serial and the parallel algorithms with respect to several threads (i.e., for 1 thread, 2 threads, 4 threads, 8 threads and 16 threads). Graph 1 shows the comparisons of time taken for computing three different sizes of data sets. Parallel k-means did not scale well as the number of threads increased either for small data sets or large data sets.

Numerical Experiments for DBSCAN and parallel DBSCAN algorithms:

Numerical Experiments for various data sets

Number of threads	50,000 data objects	500,000 data objects	1,000,000 data objects
Sequential	0.02454 sec	0.249375 sec	0.55135 sec
1	0.09187 sec	0.986754 sec	1.97144 sec
2	1.31249 sec	0.491614 sec	1.03322 sec
4	0.639358 sec	0.246805 sec	0.52332 sec
8	0.356402 sec	0.123975 sec	0.26165 sec
16	0.152778 sec	0.06328 sec	0.12770 sec

Table 2: Numerical Experiments for DBSCAN and parallel DBSCAN



Graph 2: Numerical Experiments for DBSCAN and parallel DBSCAN

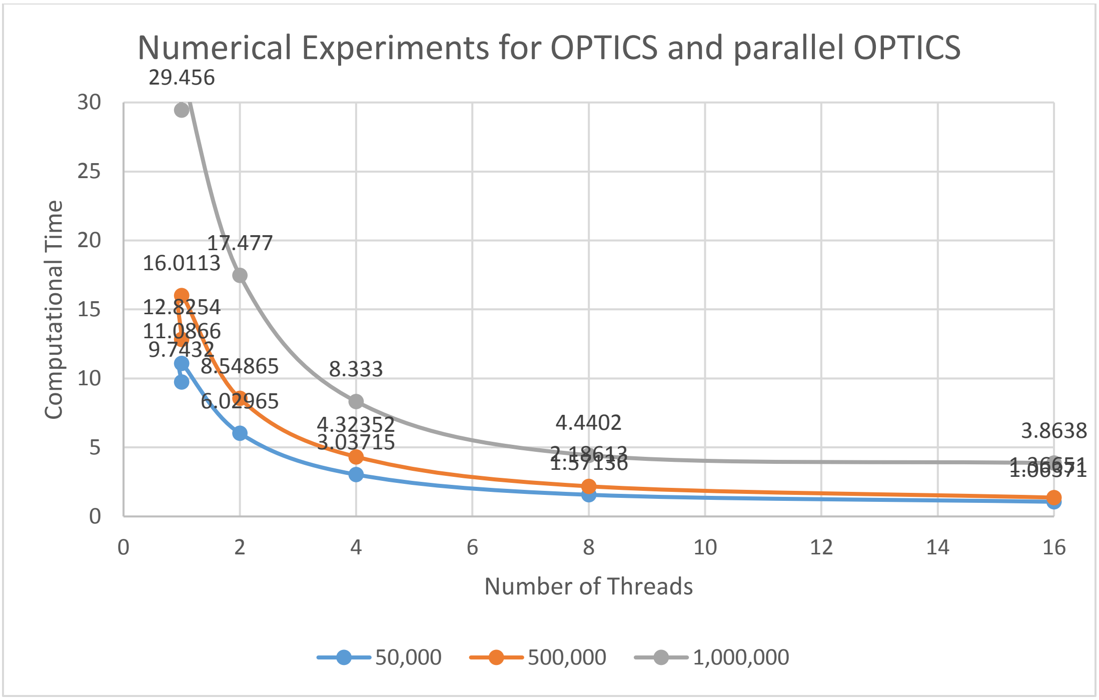
Table 2 here shows the computational time in running the serial and the parallel algorithms with respect to several threads (i.e., for 1 thread, 2 threads, 4 threads, 8 threads and 16 threads). Graph 2 shows the comparisons of time taken for computing three different sizes of data sets. Parallel DBSCAN scaled well for small data sets and it is exceptionally good for large data sets. Parallel DBSCAN scaled much better than the parallel k-means and parallel OPTICS.

Numerical Experiments for OPTICS and parallel OPTICS algorithms:

Numerical Experiments for various data sets

Number of threads	50,000 data objects	500,000 data objects	1,000,000 data objects
Sequential	9.7432 sec	12.8254 sec	29.456 sec
1	11.0866 sec	16.0113 sec	32.324 sec
2	6.02965 sec	8.54865 sec	17.477 sec
4	3.03715 sec	4.32352 sec	8.333 sec
8	1.57136 sec	2.18613 sec	4.4402 sec
16	1.06371 sec	1.36651 sec	3.8638 sec

Table 3: Numerical Experiments for OPTICS and parallel OPTICS



Graph 3: Numerical Experiments for OPTICS and parallel OPTICS

Table 3 here shows the computational time in running the serial and the parallel algorithms with respect to several threads (i.e., for 1 thread, 2 threads, 4 threads, 8 threads and 16 threads). Graph 3 shows the comparisons of time taken for computing three different sizes of data sets. Parallel OPTICS scaled well for both small and large data sets as the number of threads increased and it scaled much better than the parallel k-means. OPTICS is far slower (for both the serial and parallel cases) than either k-means or DBSCAN and is uncompetitive.

Rank	**Serial algorithms**	**Parallel algorithms**
1	*DBSCAN*	*parallel DBSCAN*
2	*K-means*	*parallel K-means*
3	*OPTICS*	*parallel OPTICS*

Table 4: Rankings of serial and parallel algorithms

Algorithms	Serial algorithms	Parallel algorithms
k-means	O(*nkd*) [5]	O(*nkd*/*P* + *n*/*P*) [5]
DBSCAN	O(*nlogn*) [7]	O(*nlogn* / *P*)
OPTICS	O(*nlogn*) [9]	O(*nlogn* / *P*)

Table 5: Computational complexity of algorithms

The computational complexity of the parallel algorithms in Table 5 that do not have citations are based on the run times in Tables 2 and 3 and the O(nlogn) complexity of the serial cases. The calculation of the run times of parallel DBSCAN and parallel OPTICS behave in such a way that the run times divided by the number of processors models the run time of the algorithm with respect to their number of threads.

Table 4 contains the order of the fastest to slowest of the clustering algorithms. The complexity results in Table 5 do not adequately predict that the constants involved for the two OPTICS cases are clearly an order of magnitude larger than the ones for DBSCAN for the data in the experiments.

Chapter 4

Conclusions

This project deals with various serial and parallel clustering algorithms. It tells us how the data is being clustered into groups using the clustering algorithms with respect to different sizes of data sets. The run times in the serial and the parallel algorithms are calculated with respect to several threads. These run times let us rank the clustering algorithms. From the comparison of run times, we conclude that the DBSCAN algorithm is the most efficient among the serial algorithms compared to k-means and OPTICS. Similarly, parallel DBSCAN is the most efficient among the parallel algorithms compared to parallel k-means and parallel OPTICS algorithms. We also conclude that parallel algorithms take less run time compared to serial algorithms. These conclusions are specific to the class of data we experimented with.

References

[1.] R. Dubes and A. Jain, ‘‘Algorithms for Clustering Data,’’ Prentice Hall, Englewood Cliffs, New Jersey, pages 89−101, 1988.

[2.] Wooyoung Kim, “Parallel Clustering Algorithms: Survey,” pages 13−19, 2009. http://grid.cs.gsu.edu/~wkim/index_files/SurveyParallelClustering.html, last visited: October, 2016.

[3.] S.G. Akl. Parallel Computation: Models and Methods. Prentice Hall, ISBN:0-13-147034-5, pages 7−23, 1997.

[4.] Wu, X., Kumar, V., Ross Quinlan, J., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G.J., Ng, A., Liu, B., Yu, P.S., Zhou, Z.-H., Steinbach, M., Hand, D.J. and Steinberg, D. “Top 10 Algorithms in Data Mining. *Knowl. Inf. Syst*.,” pages 6−10, vol. 14, 2007.

[5.] Tayfun Kucukyilmaz, “Parallel K-Means Algorithm for Shared Memory Multiprocessors,” Journal of Computer and Communications, pages 15–23, 2014. http://www.scirp.org/journal/jcc http://dx.doi.org/10.4236/jcc.2014.211002, last visited: October, 2016.

[6.] Izabela Anna Wowczko, “Density–based clustering with DBSCAN and OPTICS,” Business Intelligence and Data Mining, pages 2−7, 2013. https://www.academia.edu/8142139/Density_Based_Clustering_with_DBSCAN_and_OPTICS_-_Literature_Review, last visited: October, 2016.

[7.] Md. Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, Alok Choudhary, “A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-Set Data Structure,” SC ’12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Article No. 62, pages 2−5, 2012.

[8.] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, Jörg Sander, “OPTICS: Ordering Points To Identify the Clustering Structure,” Proc. ACM SIGMOD’99 Int. Conf. on Management of Data, Philadelphia PA, pages 51−54, 1999.

[9.] Md. Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, Alok Choudhary, “Scalable Parallel OPTICS Data Clustering Using Graph Algorithmic Techniques,” SC ’13 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Article No. 49, pages 3−6, 2013.

[10.] R. C. Prim, ”Shortest connection networks and some generalizations”, Bell System Technology Journal, pages 1389−1401, 1957. https://archive.org/details/bstj36-6-1389, last visited: October, 2016.