Design and Implementation of Scalable Hierarchical Density Based Clustering

Sankari Dhandapani, Gunjan Gupta, Joydeep Ghosh

dhandaps@mail.utexas.edu, gunjan@iname.com, ghosh@ece.utexas.edu

IDEAL-2010-06*

Intelligent Data Exploration & Analysis Laboratory

(Web: http://www.ideal.ece.utexas.edu/)

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712
U.S.A.

June 26, 2010

Abstract

Clustering is a useful technique that divides data points into groups, also known as clusters, such that the data points of the same cluster exhibit similar properties. Typical clustering algorithms assign each data point to at least one cluster. However, in practical datasets like microarray gene dataset, only a subset of the genes are highly correlated and the dataset is often polluted with a huge volume of genes that are irrelevant. In such cases, it is important to ignore the poorly correlated genes and just cluster the highly correlated genes.

Automated Hierarchical Density Shaving (Auto-HDS) is a non-parametric density based technique that partitions only the relevant subset of the dataset into multiple clusters while pruning the rest. Auto-HDS performs a hierarchical clustering that identifies dense clusters of different densities and finds a compact hierarchy of the clusters identified. Some of the key features of Auto-HDS include selection and ranking of clusters using custom stability criterion and a topologically meaningful 2D projection and visualization of the clusters discovered in the higher dimensional original space. However, a key limitation of Auto-HDS is that it requires $O(n^2)$ storage[1], and $O(n^2 logn)$ computational complexity, making it scale up to only a few 10s of thousands of points[2]. In this thesis, two extensions to Auto-HDS are presented for lower dimensional datasets that can generate clustering identical to Auto-HDS but can scale to much larger datasets. We first introduce Partitioned HDS that provides significant reduction in time and space complexity and makes it possible to generate the Auto-HDS cluster hierarchy on much larger datasets with 100s of millions of data points. Then, we describe Parallel Auto-HDS that takes advantage of the inherent parallelism available in Partitioned Auto-HDS to scale to even larger datasets without a corresponding increase in actual run time when a group of processors are available for parallel execution. Partitioned Auto-HDS is implemented on top of GeneDIVER[3], a previously existing Java based streaming implementation of Auto-HDS, and thus it retains all the key features of Auto-HDS including ranking, automatic selection of clusters and 2D visualization of the discovered cluster topology.

[1]Java Based Auto-HDS reduces the space complexity by streaming the distance matrix to the secondary storage nevertheless storage required is $O(n^2)$.

[2]limited by the computation time and not the memory, since the $O(n^2)$ storage is on the hard drive.

[3]Java based Implementation of Auto-HDS `http://www.ideal.ece.utexas.edu/~gunjan/genediver`.

1 Introduction

Clustering is a very useful unsupervised learning technique that partitions data points into a number of groups such that the data points within a group exhibit similar properties. Clustering techniques are extensively used in many areas such as data mining, machine learning, bioinformatics, marketing, astronomy, pattern recognition, image processing, etc. Exhaustive clustering techniques are quite common in which each data point is assigned to at least one cluster. However, such exhaustive clustering techniques are not appropriate for datasets that have a significant fraction of irrelevant data points. As an example, identifying genes that exhibit similar properties on a microarray gene dataset is difficult using an exhaustive clustering method as the dataset often consists of a set of experiments as clustering dimensions that only correspond to correlated activity across a small subset of genes that are involved for that specific set of experimental conditions [9]. For example in the Gasch data set [9], only a few hundred genes involved in stress response cluster well while the remaining need to be pruned to discover the stress related gene clusters. In such cases, useful clusters can be obtained more easily if the clustering is performed on the highly correlated gene subset after discarding the irrelevant genes. Several clustering techniques have been formulated to partition a smaller subset of the data into multiple clusters [14, 17, 15, 5]. Automated Hierarchical Density Shaving (Auto-HDS) [10] is a non-parametric density based hierarchical clustering technique that partitions only the relevant subset of the dataset into multiple disjoint clusters and finds a compact hierarchy of the clusters identified. A key limitation of Auto-HDS is that it requires $O(n^2)$ storage [4] and $O(n^2 logn)$ computational complexity, making it scale up to only a few tens of thousands of points [5]. Many industry-based data mining applications involve very large volumes of data that are constantly being produced by hundreds, or even thousands of servers serving millions of consumers. In order to cluster such datasets, it is important that the clustering techniques employed can scale by taking advantage of a parallel environment to address such industrial-scale clustering problems.

We present Partitioned Automated Hierarchical Density Shaving (Partitioned Auto-HDS) and Parallel Automated Hierarchical Density Shaving (Parallel Auto-HDS), that are extensions to Auto-HDS and that improve the overall performance by exploiting the inherent parallelism available in Auto-HDS. The new extensions find a compact hierarchy of dense clusters in the dataset after ignoring the irrelevant data and can easily scale up to huge datasets using either a single processor or distributed systems. In Partitioned Auto-HDS, by dividing the large dataset into p smaller partitions, the computational and storage complexities associated with Partitioned Auto-HDS is approximately [6] reduced to $O(p \times ((n/p)^2 log(n/p)))$ and $O(p \times (n/p)^2)$ respectively. With Parallel Auto-HDS in a distributed environment, the computational and storage complexities are further reduced by a factor of m, where m is the number of machines. If m is equal to p, then the computational and storage complexities can be approximated as $O(((n/p)^2 log(n/p)))$ and $O((n/p)^2)$ respectively. Even if m is less than p, Parallel Auto-HDS scales up linearly with the number of machines available. There is not much increase in the communication overhead with increase in the number of machines as the amount of

[4] Java Based Auto-HDS reduces the space complexity by streaming the distance matrix to the secondary storage nevertheless storage required is $O(n^2)$.

[5] limited by the computation time and not the memory, since the $O(n^2)$ storage is on the hard drive.

[6] this is an approximation because in the current implementation, the multiple partitions created are of equal size only if the data points are evenly distributed across the feature space.

communication required between machines is small. However, performance improvement is possible only for low dimensional datasets. As the dimensionality of the dataset increases, there is not much improvement in performance due to the curse of dimensionality.

2 Motivation and Problem Setting

Clustering in general and density based clustering in particular has been useful in the field of Astronomy that contain billions of data points, often polluted with large volumes of irrelevant data [12]. Recently, researchers have been particularly interested in identifying the halos [7] and subhalos [8] that can be used to solve the well known N-Body Simulation problem [12]. The halos and subhalos in the Halo dataset can be easily identified from the compact hierarchy of dense clusters determined by Auto-HDS. However the computational ($O(n^2 logn)$) and storage ($O(n^2)$) complexity associated with Auto-HDS makes it unsuitable for solving the astronomy clustering problem. An approximate solution to the problem is possible by subsampling the data and computing the coarse clusters. Using the non-overlapping property of sub-clusters within two different clusters, Auto-HDS can be directly applied on each coarse cluster to obtain more refined sub-clusters within the coarse clusters identified from the sampled subset. The clusters identified would be an approximation of Auto-HDS clusters and at the same time, and hence this shortcut solution is prone to noise. This approximation may be needed for high dimensional datasets like Gasch microarray gene dataset [9] but for lower dimensional datasets such as Halo, it is possible to find the exact Auto-HDS hierarchy, and at the same time reduce the computational and storage complexities associated with Auto-HDS. Although algorithms such as DBSCAN [8] can be used in conjunction with a multi-dimensional database index to find dense clusters, on Astronomy datasets such as Halo, such methods are tied to a single database and are difficult to find partitions and hence parallelize.

The main idea behind the Partitioned and Parallel Auto-HDS is that the large dataset is split into multiple smaller partitions based on the feature space. The clusters in each partition are identified using Auto-HDS. The clusters are merged to ensure the correctness of the clusters that are spread across multiple subsets. This divide-and-conquer approach gives very good speed-up on large datasets; however, as the dimensionality increases beyond 10, not much speed-up is possible due to the curse of dimensionality. The highly scalable new extensions can be used to find halos and subhalos (within a halo) from the compact hierarchy of dense clusters identified in the astronomy dataset since that dataset is only 3-dimensional. The Partitioned and Parallel Auto-HDS could also be useful in clustering extremely complex datasets such as market basket data [11, 16] that have a very limited subset of customers who exhibit similar buying behavior and a relatively huge subset of customers who exhibit completely random buying patterns (irrelevant data).

Typically, Partitioned Auto-HDS involves a single machine whereas Parallel Auto-HDS can involve hundreds or thousands of machines. With Partitioned Auto-HDS, there is a noticeable[9] decrease in the execution time and temporary storage as compared to Auto-HDS running on a comparable single machine. Even better, the relative difference in run-time gets larger as the data-sets get larger. This is due to the fact that in terms of time complexity, Partitioned

[7] a group of celestial bodies that are closely packed - example the Milky Way Galaxy.

[8] subhalos refer to smaller halos within a halo.

[9] by several orders of magnitude.

Auto-HDS is up to p×log(p) times faster than the standard Auto-HDS, where p is the number of partitions. As one would expect, the run-time for Parallel Auto-HDS on a distributed environment is less compared to Partitioned Auto-HDS on a single processor.

3 Thesis Outline

The rest of the thesis is laid out as follows: A brief introduction to the Auto-HDS framework and its key limitations are provided in Chapter 4. Then, in Chapter 9, we present an improved version of Auto-HDS, Partitioned Auto-HDS, that addresses the performance issues with clustering on large volume datasets. The detail in Chapter 15. In Chapter 23, a quick introduction to Parallel Auto-HDS, Map-Reduce framework and HADOOP along with the design overview and implementation details of HADOOP based Parallel Auto-HDS are presented. Speed-ups obtained, time and space complexities, memory usage and future work are discussed for both Java based Partitioned Auto-HDS and HADOOP based Parallel Auto-HDS in Chapter 15 and Chapter 23 respectively. Finally, results and conclusions are discussed in Chapter 31.

4 Background

Clustering techniques can be classified in many different ways but the classification that is most relevant to this thesis is parametric and non-parametric clustering. A brief introduction to the existing parametric and non-parametric clustering techniques is presented in this chapter. The non-parametric density based clustering technique, Auto-HDS, is discussed later in the chapter.

5 Parametric Clustering

Parametric clustering techniques generally make an assumption about the dataset and the clusters identified are based on the assumption. The most widely stated example of parametric clustering is k-means clustering that makes an assumption on the total number of clusters in the dataset k. Another set of parametric clustering techniques tend to assume that the dataset comes from a distribution, say mixture of Gaussians. In such cases, inferences about the parameters of the underlying assumed distribution will divide the dataset into multiple clusters. Bregman Clustering [3] is a parametric clustering technique that is highly scalable and can find clusters on large, high dimensional datasets. Bregman Bubble Clustering (BBC) [1] is an extension of Bregman clustering that can cluster a subset of the data into dense clusters.

6 Non-Parametric Clustering

The alternative to parametric clustering techniques is non-parametric clustering that does not need prior information about the number of clusters and usually does not make strong assumptions about the underlying distribution generating the clusters. In many practical applications, it is not easy to pre-determine the number of clusters due to insufficient knowledge of the dataset. The clusters in such datasets can sometimes be better determined using non-parametric clustering techniques. Density based clustering is a specific type of non-parametric

clustering technique that identifies arbitrary shaped clusters using kernel density estimation at each data point.

7 Non-Parametric Density Based Clustering

Density based clustering algorithms can find multiple clusters in the relevant subset of the dataset after ignoring the data points that do not cluster well. DBSCAN [8] is a well known kernel based density based clustering techniques. DBSCAN takes 2 parameters to find the dense clusters - neighborhood size n_ϵ and radius r_ϵ. If a data point has at least n_ϵ data points within a hypersphere of radius r_ϵ centered at the data point, all the data points within the hypersphere including the data point itself belong to the same cluster. A faster implementation of DBSCAN is possible for two to three dimensional datasets where multi-dimensional indexes for range queries are feasible. Three limitations of DBSCAN are: (1) difficulty in selecting input parameters n_ϵ and r_ϵ as they are highly data-dependent. Also, the shape and size of the clusters identified change drastically based on these input parameters, (2) As the database indices are not possible for higher dimensional datasets, DBSCAN cannot scale very well on high dimensional datasets, (3) Clustering is dependent on the order of the data. This happens due to the consideration of non-dense neighbors of points as belonging the cluster of the dense points.

OPTICS [2] is a hierarchical density based clustering algorithm that partially addresses the parameter selection problem by providing a visualization that enables the selection of parameters manually. However, significant human intuition is required to achieve that goal. An interactive exploration of the cluster hierarchy identified is also possible using the visualization framework. OPTICS however still suffers from some of the same limitations as DBSCAN, such as the clustering being dependent on the order in which the data is presented, and database driven limited scalability [10] for 2-D or 3-D datasets.

Hierarchical Mode Analysis (HMA) [19] is a non-parametric density based clustering technique that was introduced by D. Wishart in 1968 and largely unifies both the DBSCAN [8] and Auto-HDS [10] presented in this paper. DBSCAN falls out as a special case of HMA, mentioned as a footnote in Wishart's original paper, and rediscovered later by [8]. Density Shaving, a sub-algorithm of Auto-HDS, corresponds to one of the levels in HMA, and maps to one of the levels in Auto-HDS also. Just like other density based clustering methods, HMA can ignore the irrelevant data and divide only a subset of the dataset into multiple dense clusters. HMA also finds a compact hierarchy of the dense clusters identified. Hierarchical Mode Analysis is not as popular as DBSCAN and OPTICS that were developed after HMA, perhaps because in its original form it was slow and memory intensive, requiring $O(n^3)$ computation and $O(n^2)$ memory. Just like HMA, Automated Hierarchical Density Shaving (Auto-HDS) also identifies clusters of different densities and finds a compact hierarchy of the dense clusters identified. Besides better computational scaling, a significant extension in Auto-HDS over HMA is the compact 2-D projection and visualization of the clusters that still maintain their relative topological positions in the original high-dimensional space. The visualization is very useful for cluster selection and browsing, and is exploited in the Java implementation of Auto-HDS known as GeneDIVER.

[10] Database scaling is limited by the size of the single database host. The Auto-HDS clustering scaling we describe in this thesis can map-reduced to unlimited sizes since it does not use a database index for scaling.

8 Automated Hierarchical Density Shaving (Auto-HDS)

Auto-HDS [10] is a non-parametric density based clustering technique that is a faster and more scalable extension to Hierarchical Mode Analysis. Auto-HDS includes several features such as the ability to use multiple distance measures that determine the notion of density at a point, the ability to prune irrelevant data, the ability to simultaneously identify clusters of different densities, the ability to find compact hierarchy of dense clusters identified and the ability to project and browse clusters from the original high-d space into a topologically meaningful 2-D projection. However, Auto-HDS in its present form scales well only on medium-sized clustering problems involving up to 10^5 points[11]. Auto-HDS is very useful in the field of bioinformatics as the high dimensional microarray gene datasets [9, 13, 18] tend to be very noisy with a significant fraction of irrelevant data.

Auto-HDS requires three parameters from the user: (1)n_ϵ, minimum neighborhood size required for a data point to be classified as dense, (2)f_{shave}, the fraction of least dense points that are to be ignored before partitioning the remaining dataset into clusters of different densities, (3)r_{shave}, the fraction of least dense points to be ignored at each level. n_ϵ is perhaps the only significant parameter and acts as a smoothing parameter. Less significant and smaller clusters disappear from clustering results as n_ϵ increases. The clustering results are fairly robust to r_{shave} which is more useful for controlling clustering speed, by trading off for slight degradation of discovered cluster boundaries. f_{shave} is also used for speeding up clustering by simply ignoring the least dense fraction. This property is useful when the user is only looking for the most dense and small clusters. The absence of too many critical parameters for clustering is another important and an often useful (especially in a highly unsupervised setting) feature of Auto-HDS.

8.1 Dense Points

Auto-HDS uses a kernel-based notion of density where the density at a point is measured by the number of points within a pre-defined radius around the point. In Auto-HDS, a data point is considered as *dense* if there are at least n_ϵ data points within a hypersphere of certain radius, say r_ϵ, centered at the data point. Also it can be stated that a data point is considered to be non-dense if that neighborhood size is less than n_ϵ.

8.2 Density Shaving (DS)

The notion of density at a data point is theoretically determined by two parameters: neighborhood size n_ϵ and radius r_ϵ (Section 8.1). However n_ϵ is an input parameter that is held constant over all points, hence the notion of density is technically dependent on just the radius r_ϵ. Then in Density Shaving, which is one of steps in the Auto-HDS clustering algorithm, the task is to find the clusters from the dense points identified using this r_ϵ. Two dense points belong to the same cluster if they lie within the distance of r_ϵ. Note that this results in a chain of dense points, such that if there is at least one other dense point belong to the chain within the distance of r_ϵ from each dense point in the chain, then the entire chain of dense points belong

[11] Beyond that size even Auto-HDS requires too much hard drive space and computation time on a standard 2010 desktop. This is still better than its ancestor HMA, which would only scale to a few 1000 points on a modern desktop.

to the same cluster. Note that all the points that are not dense get pruned or "shaved", and do not end up in any of the dense clusters. This is the key difference between density-based clustering, where we discover dense clusters only, vs. traditional partitional clustering such as K-Means, where we cluster all the points into some cluster. This shaving is important for finding pure dense clusters since it removes less dense regions. Other density based clustering methods such as [1] and [7] use other ways of pruning such less dense points.

8.3 Hierarchical Density Shaving (HDS)

HDS finds a compact hierarchy of clusters of different densities identified by Density Shaving. Conceptually in HDS, the dense clusters of different densities are identified by repeatedly applying Density Shaving by holding n_ϵ constant and the only varying parameter is r_ϵ which is not an input to the algorithm. Let $\mathbf{d}_{n_\epsilon}$ denote the vector of distances of each data point from its n_ϵ^{th} closest point and the distances are sorted in an ascending order. There are two possible ways for determining this r_ϵ in order to find dense clusters.

Linear Shaving:
A straight forward method is to increase r_ϵ gradually in a linear fashion by setting r_ϵ to $\mathbf{d}_{n_\epsilon}(1)$, $\mathbf{d}_{n_\epsilon}(2)$, $\mathbf{d}_{n_\epsilon}(3)$, etc. The compact hierarchy of dense clusters thus identified using linear shaving is identical to the HMA cluster hierarchy.

Exponential Shaving
Another option is to increase r_ϵ exponentially based on a desired shaving fraction of least dense points f_{shave}.

When r_ϵ is set as $\mathbf{d}_{n_\epsilon}(2)$, that is to the closest neighbor, the only point that is classified as dense for this r_ϵ is that closest neighbor with $\mathbf{d}_{n_\epsilon}$ set as r_ϵ. Similarly, if r_ϵ is set as $\mathbf{d}_{n_\epsilon}(3)$, only up to two dense points, besides the point itself, are present within this r_ϵ and the rest are 'pruned' or 'shaved' or 'ignored' as irrelevant. HDS cluster hierarchy is thus a sampled subset of the HMA cluster hierarchy, and is obtained using the Exponential shaving using the shaving parameter r_{shave}. The corresponding list of r_ϵ values and hence the number of HDS iterations n_{iter} can be determined using the input parameter r_{shave}, using the following equation:

$$n_{iter} = \lceil -\frac{\log(n)}{\log(1 - r_{shave})} \rceil \tag{1}$$

The list of r_ϵ values is given by

$$\mathbf{r}_{\epsilon list} = \mathbf{d}_{n_\epsilon}(\mathbf{n}_{nclist}) \tag{2}$$

where $\mathbf{n}_{nclist}$ is given by

$$\mathbf{n}_{nclist} = sortd(unique(\{\lceil n \times (1 - r_{shave})^t \rceil\}_{t=0}^{j_{max}})) \tag{3}$$

where *sortd* represents a sort by decreasing value and $n_{iter} = |\mathbf{n}_{nclist}|$ represents the number of iterations, with the j^{th} entry of $\mathbf{n}_{nclist}$ corresponding to the j^{th} iteration of HDS.

8.4 Pseudo-code

Auto-HDS consists of levels or iterations, given by $\lceil -\frac{\log(n)}{\log(1-r_{shave})} \rceil$ with each iteration finding dense clusters of specific density determined by r_ϵ corresponding to each level (Equation 2). Notice that the iterations are independent of each other and therefore dense clusters of different densities can be identified independently. Density Shaving (Algorithm 1) is used for finding dense clusters corresponding to a specific density (iteration) and the inputs to Density Shaving are n_ϵ, distance matrix $\mathbf{M}_S$ and r_ϵ or n_c. Note that either r_ϵ or n_c that determines r_ϵ can be passed as an input to Density Shaving. In summary, an Auto-HDS algorithm would involve $\lceil -\frac{\log(n)}{\log(1-r_{shave})} \rceil$ calls to Density Shaving to find dense clusters of different densities. Finally, relabeling of clusters, also known as 'compaction' of clusters, is performed to find a compact hierarchy of the dense clusters identified over all the HDS levels.

Algorithm 1 DS

Input: Distance matrix $\mathbf{M}_S$, n_ϵ, n_c or r_ϵ
Output: Cluster labels $\{lab_i\}_{i=1}^n$ corresponding to the n data points.
Initialize: $\{lab_i\}_{i=1}^n = 0$
// Sort each row of the distance matrix
$[\mathbf{M}_{rad}^{nbr}, \mathbf{M}_{idx}^{nbr}] = sortrows(\mathbf{M}_S)$
//Sort $n_\epsilon{}^{th}$ column of matrix $\mathbf{M}_{rad}^{nbr}$
$[\mathbf{radx}^{n_\epsilon}, \mathbf{idx}^{n_\epsilon}] = sort(\mathbf{M}_{rad}^{nbr}(\cdot, n_\epsilon))$
// Recover the r_ϵ threshold
if (exists(n_c) $r_\epsilon = radx^{n_\epsilon}(n_c)$
// Recover the n_c densest points
$\mathcal{G} = \{\mathbf{x}(idx^{n_\epsilon}(i))\}_{i=1}^{n_c}$
/* Lines 17-33: For each point in $\mathcal{G}$, find other dense points within r_ϵ distance of it and make sure they have the same labels, if not, relabel */
for $i = 1$ to n_c **do**
  /* Find the position of the last point within distance r_ϵ of dense point $\mathbf{x}(idx^{n_\epsilon}(i))$. */
  $idxb = binSearch(\{\mathbf{M}_{rad}^{nbr}(idx^{n_\epsilon}(i), j)\}_{j=n_\epsilon}^{n})$
  /* Neighbors of $\mathbf{x}(idx^{n_\epsilon}(i))$ are the $idxb$ closest points, all within r_ϵ distance. */
  $\mathcal{X}_{nbrs} = \mathbf{M}_{idx}^{nbr}(idx^{n_\epsilon}(i), l)_{l=1}^{idxb}$
  // save the neighbors
  // Identify neighbors that are dense points
  $\mathcal{X}_{dnbrs} = \mathcal{X}_{nbrs} \cap \mathcal{G}$
  // Recover their labels that are not 0
  $L_{dnbrs} = unique(lab(\mathcal{X}_{dnbrs}))/\{0\}$
  // Relabel all points that share this label to label i
  $\forall y \in \mathbf{lab}$ if $\exists y \in L_{dnbrs} : y = i$
  $\mathbf{lab}(indexOf(\mathcal{X}_{dnbrs})) = i$
end for
Count clusters: $k = |unique(\mathbf{lab})|/\{0\}$
Remap the non-zero labels in **lab** to the range 1 to k.

9 Partitioned Automated Hierarchical Density Shaving (Partitioned Auto-HDS)

10 Introduction

Auto-HDS is a non-parametric density based clustering algorithm that finds a compact hierarchy of dense clusters of different densities. Some of the key features of Auto-HDS include an interactive 2D visualization framework, ability to address large clustering problems, ability to filter out irrelevant data, ability to identify clusters of different densities and the ability to select and rank clusters using a custom stability criterion. All these key features make it a good fit for addressing several problems with the bio-informatics datasets. Although the GeneDIVER implementation of Auto-HDS is suitable for solving the clustering problems in several domains, it takes relatively long time and more memory when it comes to large volume datasets. For example, it takes approximately 1.5 days to run GeneDIVER on a 3-D astronomy dataset of 2 million data points on a 8 Core AMD machine. In this chapter, we present 'Partitioned Auto-HDS' that is an improved version of the Auto-HDS algorithm. The statistics presented in this chapter were collected using the MATLAB based Partitioned Auto-HDS on a modest dual core AMD desktop machine. A more detailed explanation of Java based Partitioned Auto-HDS, which runs even faster, is presented in Chapter 15.

An overview of Partitioned Auto-HDS is as follows: divide the dataset into multiple partitions followed by stitching the clusters obtained from each partition. The theory behind Partitioned Auto-HDS, an overview of the components in Partitioned Auto-HDS, the correctness of the new framework and the speed up achieved have been discussed in the following sections. The issues that have been addressed to ensure the correctness of Partitioned Auto-HDS are also explained briefly in this chapter.

11 Partitioned Auto-HDS

As mentioned earlier, Auto-HDS identifies clusters of different densities and finds a compact hierarchy of dense clusters. The dense clusters in the hierarchy are recovered from an independent set of iterations where each iteration finds clusters of a specific density. A simple Auto-HDS algorithm involves repeated calls to the density shaving algorithm with different but automatically computed r_ϵ. Partitioned Auto-HDS consists of three steps: partitioning the feature space into overlapping partitions, repeated calls to Modified Density Shaving on each partition followed by Stitching. In Auto-HDS, clusters of a specific density would involve a call to density shaving algorithm. In case of Partitioned Auto-HDS, this would involve partitioning ($Partitioner$), Auto-HDS on each partition ($SlaveDIVER$), followed by stitching ($Stitcher$). The proof of correctness[12] of Partitioned Auto-HDS can be narrowed down to verifying the correctness of clusters identified in a specific iteration (i.e clusters of a specific density).

Lemma 11.1. *Given a dataset, the number of clusters is always either less than or equal to the number of dense points in the dataset.*

[12]correctness is defined here as results of Partitioned Auto-HDS being identical to those of Auto-HDS.

Assume the number of clusters is greater than the dense points. In Auto-HDS, the dense points are identified and the rest are classified as non-dense points. In identifying the clusters, only dense points are considered and hence a non-dense point can never belong to a cluster. In density based clustering, each (dense) data point can belong to just one cluster (unlike in some other clustering techniques where a data point can belong to more than one cluster). Since only dense points can be clustered, even in the worst case of each dense point belonging to a different cluster, the number of clusters is equal to the dense point count. Hence, the number of clusters is always less than or equal to the number of dense points in the dataset.

11.1 Partitioner

In this stage of Partitioned Auto-HDS, the dataset is divided into p partitions of approximately equal extent along each dimension of the feature space. In very simple terms, the feature space is diced up into segments of equal length along each dimension, resulting in each partition occupying a contiguous region. Furthermore, the partitions are created so that they partially overlap along each dimension. Furthermore, we create partitions that are overlapping along each dimension by exactly $3 \times r_\epsilon$. We now show why that is required in the Partitioner for Partitioned Auto-HDS to work correctly.

Lemma 11.2. *An overlap of at least r_ϵ between adjacent partitions along each feature dimension is required to guarantee that each point in the dataset is correctly clustered in at least one of the p partitions.*

From the definition of the Auto-HDS algorithm, a data point is labeled as a dense point if at least n_ϵ data points (including itself) are enclosed within the radius $\mathbf{r}_\epsilon$. Therefore, for all the data points in a partition to be correctly classified as dense or non-dense point, the partition should include data points from the adjacent partitions that are within a certain distance from the partition border. An extra band of width of r_ϵ guarantees the accurate classification of the data point that lies exactly on the partition border. Since a border point (that lies exactly on the border of the partition) is an upper bound case, it is guaranteed that all the other partition (border) data points will be accurately classified as a dense/non-dense point. It is important to note that an extra band of width of r_ϵ into the adjacent partition results in a total overlap of $2 \times r_\epsilon$ between any two adjacent partitions.

Note that the resulting partitions get populated based on the distribution of the data points in the feature space. Therefore, the number of points in each partition may not be the same. Hence the computing load may not be equally distributed across all partitions[13].

From Property 11.2, an overlapping width of at least r_ϵ guarantees correctness of the clusters that are confined in the non-overlapping region of a partition. To verify the correctness of clusters across all partitions, we need to ensure the following:

1. Clusters should have unique labels across all partitions.

2. A cluster that is split across multiple partitions should be assigned a unique label.

[13]We do not address this issue in this thesis; for future work, a more advanced partitioning strategy that results in approximately equal load in each partition could be developed using estimates derived from a random sample of the original distribution.

The first case can be handled by relabeling clusters such that no two clusters from different partitions have the same label. But this solution does not solve the second problem of clusters spread across multiple partitions. Since each partition has unique cluster labels, a cluster that is spread across multiple partitions will be assigned different labels.

A solution to handle both cases would therefore be to relabel the clusters that are confined to a single partition. However clusters, that are spread across multiple partitions, should be handled separately such that a cluster gets the same label irrespective of the partition. The first step towards solving this special case would be to identify these clusters and we claim that this is possible with an overlap of at least $3 \times r_\epsilon$ between adjacent partitions. (This claim will be proved eventually in Property 13.1)

In an overlap of $3 \times r_\epsilon$ between adjacent partitions, $1.5 \times r_\epsilon$ will come from each partition. Hence each partition will include an extra band of width of at least $1.5 \times r_\epsilon$ from the adjacent partitions along each dimension.

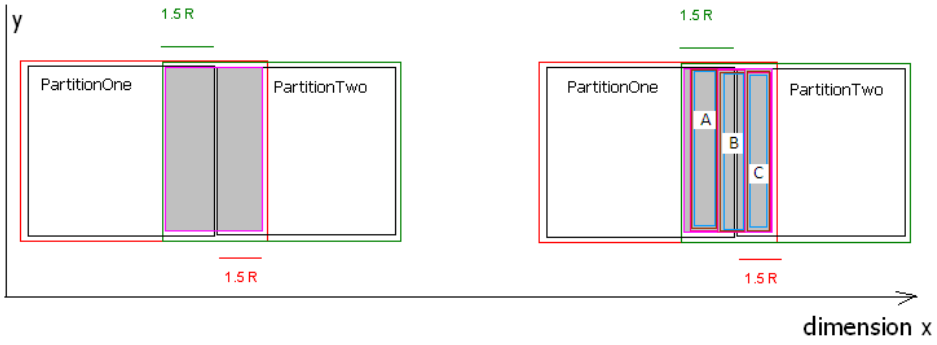

Figure 1: Two Adjacent Partitions *PartitionOne* and *PartitionTwo* of Partitioned Auto-HDS overlapping along feature dimension x.

This overlapping section of $3 \times r_\epsilon$ can be divided into three regions of width r_ϵ each. Let these sections be A, B and C shown in Figure 1. Note that if the overlap between partitions increases beyond r_ϵ, the width of Section A and C remain the same but Section B linearly increases with increase in the overlap. The data points of a partition can be categorized into three categories:

1. points from this partition that are correctly classified (Category 0).
2. points from the adjacent partition that are incorrectly classified (Category 1).
3. points from the adjacent partition that are correctly classified (Category 2).

A pictorial representation of the three data point categories is in Figure 2.

From Property 11.2, an extra band of width of at least r_ϵ is required for identifying the dense points. Hence the dense points in Section A and Section B of PartitionOne will be identified correctly because of the extra band of width r_ϵ formed by Section C. Likewise, the dense points in Sections B and C are identified correctly in PartitionTwo with the extra band of width r_ϵ formed by Section A. Note that the dense points in Section A and C are wrongly identified in PartitionTwo and PartitionOne respectively.

To summarize

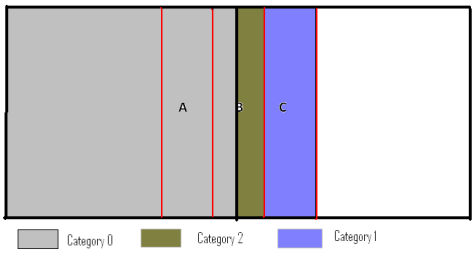

Figure 2: Three Categories of Data Points in a Partition of Partitioned Auto-HDS

- The dense points in Section B are clustered correctly in both PartitionOne and PartitionTwo.
- The dense points in Section A are correctly clustered in PartitionOne.
- The dense points in Section C are correctly clustered in PartitionTwo.
- The dense points in Section C cannot be clustered correctly in PartitionOne.
- The dense points in Section A cannot be clustered correctly in PartitionTwo.
- In each partition, data points are either correctly clustered or incorrectly not clustered.
- The data points that are incorrectly not clustered come from the adjacent partitions.

12 SlaveDIVER

Once the partitions are created, Auto-HDS on each partition is then used to find clusters for different Auto-HDS levels, each of which correspond to different densities as given by Equation 2. With the dense clusters obtained from this module, the Stitcher merges the dense clusters across all partitions.

13 Stitcher

The job of the stitcher is to relabel clusters found across multiple partitions thus giving us the final clustering that is identical to the non-partition Auto-HDS. Stitching is the second main component, next to partitioner, that is responsible for the correctness of the algorithm. Since the dataset can have multiple partitions and multiple dimensions, stitching can be thought of as having three stages. A typical stitching component may or may not include all the three stages depending on the number of partitions and dimensionality. The three stages of Stitching - Stitching Between Two Partitions, Stitching Along One Dimension and Stitching Along Multiple Dimensions - are discussed in more detail in the following sections.

13.1 Stitching Two Partitions

In the overlapping region of $3 \times r_\epsilon$ between adjacent partitions, dense clusters in Section A&B of first partition and Section B&C of second partition are identified. Section B is present in both the partitions and stitching of the cluster is done based on the dense points in Section B. There are two main cases that should be considered for stitching:

No Dense points in Section B
In the case where there are no dense points found in Section B, it can be concluded that the clusters in PartitionOne are independent of the clusters in PartitionTwo. Hence Stitching need not be performed.

Dense points in Section B
In this case, there is at least one dense point found in Section B. The cluster labels from PartitionOne and PartitionTwo of the dense points in Section B will not match since labels from different partitions are guaranteed to be unique. However the dense points in Section B will be identified in both PartitionOne and PartitionTwo. Taking advantage of this fact, stitching is performed by iterating through each dense point in Section B and finding the labels from both the partitions, say LabelOne and LabelTwo. Once the labels are identified, the dense points with LabelOne from PartitionOne and dense points with LabelTwo from PartitionTwo are relabeled to a new cluster label, say LabelNew. The complexity associated with Stitching is therefore dependent on the number of dense points in Section B.

Hence this optimized stitching logic offers significant performance improvement in practice as the number of dense points to the number of clusters ratio can be very high for many problems.

13.2 Stitching Along One Dimension

13.2.1 Linear Traversal

In this approach, if there are p partitions along a dimension, then the partitions are stitched in a linear fashion, say (1, 2), followed by (2, 3) and this goes on till ($p-1$, p). It can be seen that there are totally $(p-1)$ stitches involved along a dimension that has p partitions. The time complexity associated with Linear Traversal can be approximated as $O(p \times \mathbf{clus}_{avg})$, where $\mathbf{clus}_{avg}$ is the average number of clusters in Section B of the $p-1$ stitches performed.

13.3 Stitching Along Multiple Dimensions

The number of partitions in Partitioned Auto-HDS increases exponentially with increase in dimensionality. Stitching performed in an organized manner will avoid unnecessary computation and will therefore improve performance. The idea is to perform Stitching along one dimension at a time and then move on to the second dimension.

The fact that partitions once merged can be considered as a single unit for further processing is extensively used in stitching along multiple dimensions. The dimensions already stitched are considered as a single unit if there are multiple partitions involved along this dimension. In case of a 2D dataset with 8 partitions along the first dimension, the entire row stitched earlier

can be considered as a single unit. Based on the same argument, the first two partitions along the second dimension can be represented as ((1, 2, 3, 4, 5, 6, 7, 8), (9, 10, 11, 12, 13, 14, 15, 16)). Stitching along the second dimension is again performed in a linear fashion. So if there are 4 partitions along the second dimension, totally three stitches are performed in two (log 4 = 2) rounds. On a 2 dimensional dataset with m × n partitions, if linear stitching is performed, the stitching time complexity can be approximated as O(m × n × $\mathbf{clus}_{avg}$).

Lemma 13.1. *A minimum overlap of at least 3 × r_ϵ is required for the correctness of the clusters across all partitions after stitching.*

From Property 11.2, an overlap of 3 × r_ϵ will ensure that the dense points in Section B of width r_ϵ between adjacent partitions are correctly identified. The dense points separated by a distance of less than or equal to r_ϵ should belong to the same cluster irrespective of the partitions. If the width of Section B is less than r_ϵ, the two border dense points separated by a distance of r_ϵ (upper bound case) will belong to different clusters as the dense points do not lie in Section B. Since merging of clusters is based on the dense points in Section B, the width of Section B should at least be r_ϵ (upper bound case). As mentioned earlier, Sections A and C have a fixed width of r_ϵ each. Hence, it has been proved by contradiction that an overlap of at least 3 × r_ϵ is required for the correctness of the Partitioned Auto-HDS algorithm.

14 Results

The scale-up factor can be defined as ratio of the time taken by the Auto-HDS to the time taken by Partitioned Auto-HDS to solve a problem. The scale-up achieved with Partitioned Auto-HDS comprises of two components: scale-up based on the dataset size and scale-up based on the number of partitions. The dataset scale-up increases with increase in the dataset size, whereas the partition scale-up increases with increase in the number of partitions until a threshold, after which there is no noticeable scale-up. The increase in the dataset scale-up is due to the fact that the computational and storage complexity associated with Auto-HDS increases drastically with increase in the dataset size, whereas the complexity associated with Partitioned Auto-HDS is relatively low because of multiple smaller subsets. The increase in partition scale-up is because the kernel density estimate needs to be done only for a smaller neighborhood of data points within a partition; however, beyond a certain point, the overhead associated with the partitioning and stitching dominates the overall complexity and hence there is no increase in the partition scale-up.

The results in this section are from the MATLAB implementation of Partitioned Auto-HDS. The speed-up achieved using Partitioned Auto-HDS was tested on an artificial dataset Sim-2 [10]. The Sim-2 dataset was generated containing five 2-D Gaussian distributions of different variances, where each distribution corresponds to a cluster. Since the ground truth is known, this simulated dataset is very useful in verifying the correctness of the clustering algorithms. Figure 3 captures the behavior of Partitioned Auto-HDS based on the dataset size. The time complexity increases with increase in the dataset size as expected. The speed-up achieved with respect to Auto-HDS also improves gradually as the dataset size increases. In these experiments, both the dimension(2) and partitions(4) were held constant and the dataset size was varied.

Figure 4 and 5 capture the behavior of Partitioned Auto-HDS based on the number of partitions created in the dataset of size 9000. The time complexity decreases gradually as

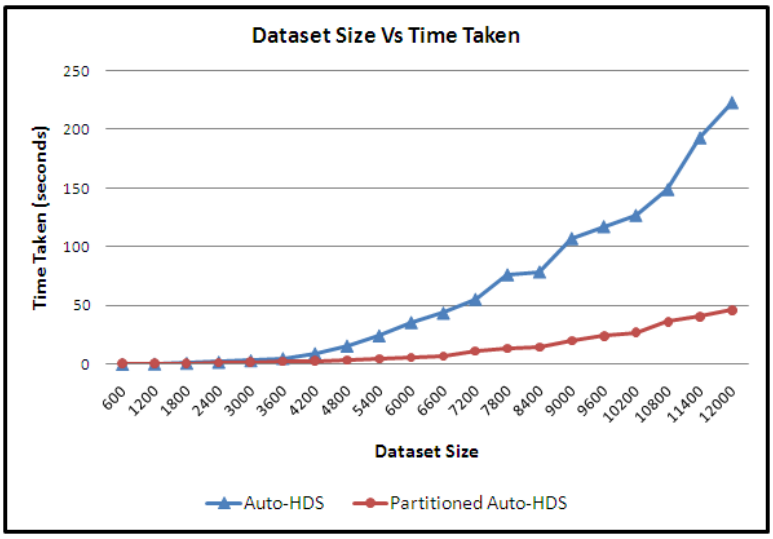

Figure 3: Execution time for varying size of the 2-D Sim-2 Dataset (4 Partitions).

the number of partitions increases and after a certain point, the curve gets almost flattened and there is not much decrease in the computational time complexity. This behavior could be attributed to two reasons: (1) the decrease in computational time complexity is small beyond a certain number of partitions such that the curve looks flattened relative to other sections in the graph (2) the time complexity of stitching is directly dependent on the number of partitions and hence the computational time complexity is dominated by the stitching operation as the number of partitions increases.

In the experiments shown in Figure 6, the dataset size (3000) and the number of partitions (4) are held constant and the dimensionality of the dataset is varied. It is evident from the plot that not much speed-up is achieved with increase in the dimensionality of the dataset, due to the curse of dimensionality. As dimensionality increases, the fraction of data points that lie in the overlapping region outweighs the fraction of data points that lie in the non-overlapping region. The amount of unnecessary computation, that is performed to ensure the correctness of the algorithm, increases as the volume of data points in the overlapping region increases. Hence it can be concluded that Partitioned Auto-HDS outperforms Auto-HDS on lower dimensional datasets (approximately up to 10D), whereas Auto-HDS outperforms Partitioned Auto-HDS on higher dimensional datasets. Unlike Partitioned Auto-HDS, Auto-HDS depends only on the dataset size and hence the time complexity is constant irrespective of the dimensionality of the dataset.

15 Java Based Partitioned Auto-HDS

This chapter discusses the Java implementation of Partitioned Auto-HDS in more detail. It has been shown from the MATLAB implementation of Partitioned Auto-HDS that given a complex dataset, clusters that are identical to Auto-HDS are obtained in a more efficient manner. A few limitations with the MATLAB implementation are (1) memory is taxed heavily due to the lack of features like streaming and therefore cannot scale up to large datasets, (2) Implementation

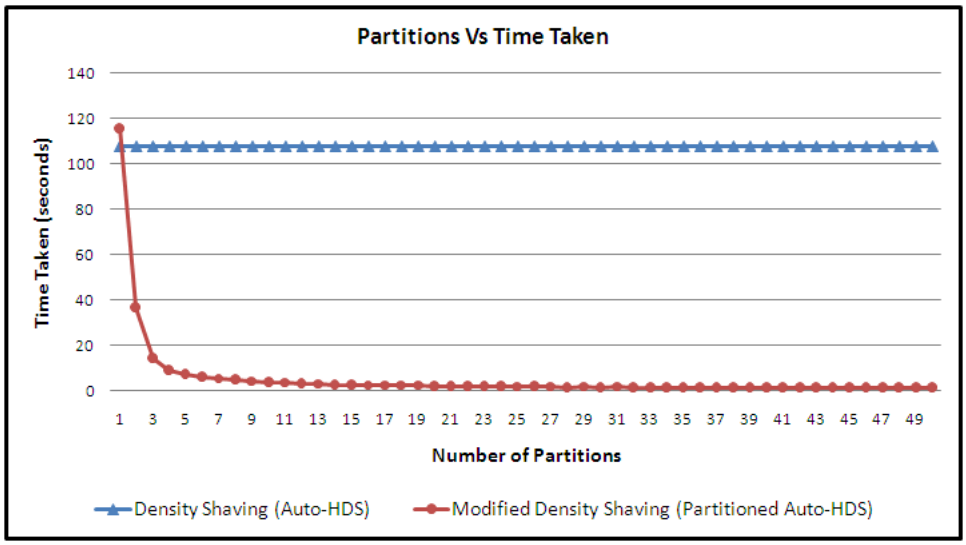

Figure 4: Execution time of Auto-HDS and Partitioned Auto-HDS for varying number of partitions on the 2-D Sim-2 Dataset of size 9k.

requires the commercial platform 'MATLAB' for finding clusters in the dataset, (3) MATLAB interpreter runs much slower than Java. The advantages of the Java based implementation include efficient use of memory using streaming and ability to use the implementation to solve clustering problems on most of the platforms. Also, a Java based implementation does not require proprietary MATLAB license to run.

The existing Java heap-based implementation of Auto-HDS, GeneDIVER, can be used to handle reasonably large clustering problems. GeneDIVER includes an interactive SWING based interface that enables the user to visually analyze the hierarchy of clusters obtained from Auto-HDS. The improvement in time and space complexities is achieved by various optimizations including a custom heap sort that reuses the partially sorted heaps present in secondary storage and does not load the entire distance matrix into memory. By building Java based Partitioned Auto-HDS as an extension of GeneDIVER, several useful features of GeneDIVER are retained. An introduction to Java based Partitioned Auto-HDS followed by a detailed explanation of the modules in Partitioned Auto-HDS is presented in this chapter.

16 Introduction

Recall from Property 13.1, an overlap of $3 \times r_\epsilon$ is required between adjacent partitions to ensure correctness of the algorithm. Partitioning of the dataset is based on r_ϵ, as this determines the degree of overlap required between adjacent partitions. Therefore the first major challenge associated with the Java Implementation is to obtain the list of r_ϵ for hierarchical clustering $\mathbf{r}_{\epsilon list}$ to find clusters of different densities. In GeneDIVER, once the distance matrix is sorted, $\mathbf{r}_{\epsilon list}$ is calculated from the distance matrix based on the parameter r_{shave}. Sorting the distance matrix and calculating $\mathbf{r}_{\epsilon list}$ is a cumbersome task with large volume datasets as the $O(n^2)$ distance matrix may not fit into memory of a single machine. GeneDIVER handles this problem with such large datasets by not loading the $O(n^2)$ distance matrix into memory at any point of

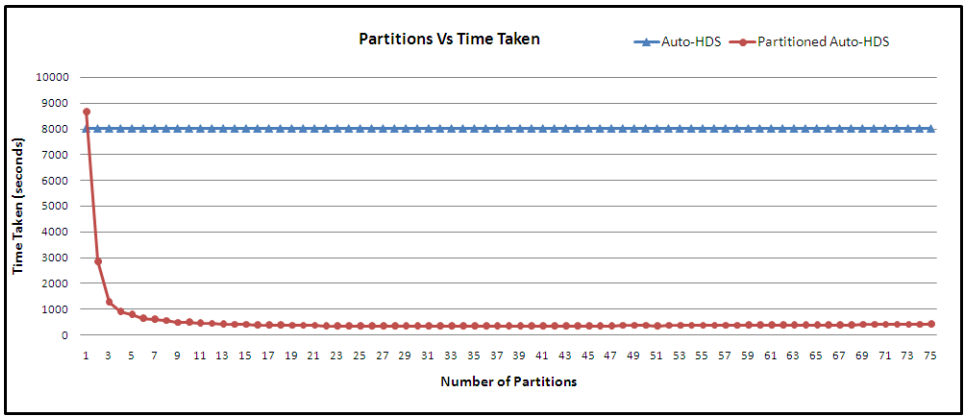

Figure 5: Execution time of Density Shaving and Modified Density Shaving (Partitioned Auto-HDS) for varying number of partitions on the 2-D Sim-2 Dataset of size 9k.

time and by sorting distances from one data point at a time. GeneDIVER estimates r_ϵ for each of the Auto-HDS levels based on the desired shaving fraction r_{shave} given by Equation 2 and 3. However, for the Partitioned based extension, the correct r_ϵ cannot be computed from the partitions as the number and distributions of data points in each partition can be very different. This problem is addressed by finding an approximation for $\mathbf{r}_{\epsilon list}$ using a sampling of the whole data, which takes considerably much less time. Note that this only leads to approximations in the overall shaving rates and not the correctness of the Auto-HDS clusters. We still get a subset of the HMA cluster hierarchy that follows an approximately exponential shaving rate. Hence Java based Partitioned Auto-HDS includes 'Parameter-Estimator' phase in addition to the 'Partitioner', 'SlaveDIVER' and 'Stitcher' phases discussed in Chapter 9.

Java based Partitioned Auto-HDS includes five modules that are listed below:

1. Parameter-Estimator
2. Partitioner
3. SlaveDIVER
4. Stitcher
5. Compact-HDS

The first four modules form the major components of Partitioned Auto-HDS and the improvement in performance over Auto-HDS can be attributed to these modules. A block diagram that indicates control flow and data flow between different components of Partitioned Auto-HDS is displayed in Figure 7.

17 Parameter-Estimator

In this module, a sampled subset of the complete dataset is used for estimating $\mathbf{r}_{\epsilon list}$. A random sampling of the dataset is performed with the subset size determined by the configurable parameter - 'random rate'. GeneDIVER returns a list of r_ϵ on the sampled subset.

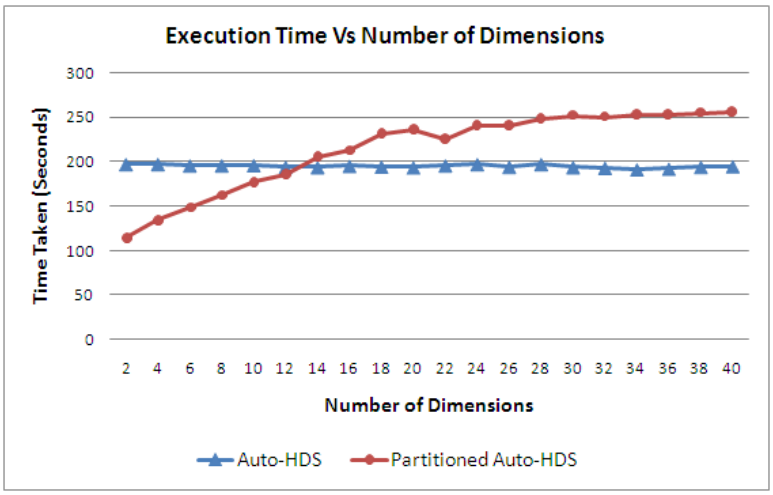

Figure 6: Execution time of Auto-HDS and Partitioned Auto-HDS for varying number of dimensions of the Sim-2 Dataset of Size 3k (4 Partitions).

$Parameter-Estimator$ phase gives an estimate of r_ϵ for the large dataset from its sampled subset. The $\mathbf{r}_{\epsilon list}$ value estimated is used in the succeeding stages to find dense clusters. Since Partitioned Auto-HDS takes much less time relative to Auto-HDS, the overhead associated with adjusting $\mathbf{r}_{\epsilon list}$ and re-running the algorithm is very less. This stage is optional if the user has an idea of $\mathbf{r}_{\epsilon list}$ to be used for the dataset.

The overhead associated with this phase is dependent on the configurable parameter *randomrate* and the overhead increases as *randomrate* increases. This is because as *randomrate* increases, the subset size increases and hence the time complexity of GeneDIVER on this sampled subset increases. Note that an estimate of $\mathbf{r}_{\epsilon list}$ is calculated using the first few steps of Auto-HDS (Algorithm 2). Hence, estimating the $\mathbf{r}_{\epsilon list}$ is not as time consuming and complex as Auto-HDS since the major overhead is associated with finding dense clusters in the dataset.

Algorithm 2 Parameter-Estimator

Input: Sampled Distance matrix $\mathbf{M}_S$, n_ϵ, r_{shave}
Output: $n \times n_{iter}$ Cluster hierarchy matrix $\mathbf{L}$.
Initialize all values in $\mathbf{L}$ to 0.
$[\mathbf{M}^{nbr}_{rad}, \mathbf{M}^{nbr}_{idx}] = sortrows(\mathbf{M}_S)$
$[\mathbf{idx}^{n_\epsilon}, \mathbf{radx}^{n_\epsilon}] = sort(\mathbf{M}^{nbr}_{rad}(\cdot, n_\epsilon))$
Compute $\mathbf{n}_{nclist}$ using Equation 3.
$\mathbf{r}_{\epsilon list} = \mathbf{radx}^{n_\epsilon}(\mathbf{n}_{nclist})$

17.1 Discussion and Future Work

In this implementation, the O(n^2) distance matrix is never loaded into memory and is persisted in the secondary storage. By sorting one data point at a time, memory is not taxed heavily, thereby making it possible to address large clustering problems. $\mathbf{r}_{\epsilon list}$ depends only on the n^{th}_ϵ closest neighbor of each data point. As only the first n_ϵ neighbors of a data point are sorted, the memory usage of Parameter-Estimator can be approximated as O(n $\times$ n_ϵ).

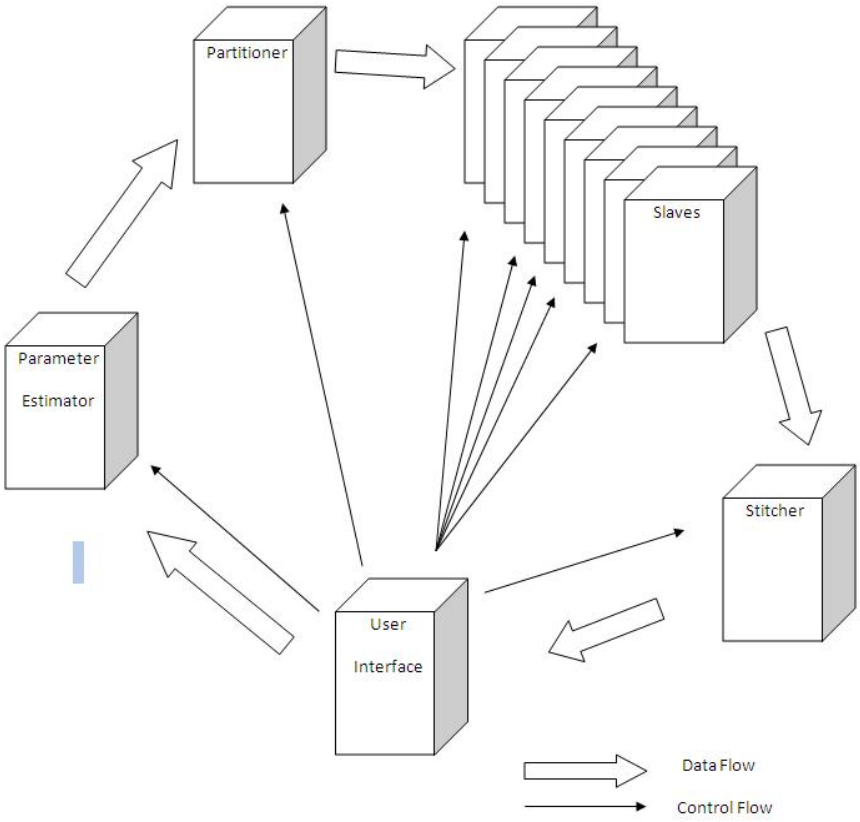

Figure 7: Overview of the Java Based Partitioned Auto-HDS Implementation.

Sorting of each data point is independent of any other data point and this fact leaves room for further optimization using multiple machines. The speed up achieved is directly dependent on the number of machines $n_{machine}$. The memory usage using $n_{machine}$ machines is given by $O((n \times n_\epsilon)/n_{machine})$.

18 Partitioner

The dataset is partitioned into overlapping subsets and Auto-HDS is run on each subset. Recollect the fact that partitioning is based on the feature space. Hence, two data points of the same cluster can belong to different subsets due to their overlapping nature. For example, in a 3D dataset, a data point can belong to up to 8 partitions. An upper bound on the total number of partitions possible for a data point depends on the number of dimensions and is given by 2^d, where d is the dimensionality of the data. A major step in this phase is therefore to identify all possible partitions that a data point belongs to as it is read from the file.

In Chapter 9, we discussed three sets of data points in a single partition.

1. data point from this partition that is correctly classified (0).

2. data point from the adjacent partition that is incorrectly classified (1).

3. data point from the adjacent partition that is correctly classified (2).

In each partition, along with the data point, an additional flag is maintained. A flag value of 0 refers to data points from the same partition, whereas a flag value of '1' and '2' refer to data points from adjacent partitions. Only the data points with flag value of 0 and 2 will be correctly clustered. This information is used in the Stitcher module that stitches the clusters from all the partitions. The pseudo-code of Partitioner is shown in Algorithm 3.

18.1 Discussion

Some of the key features and limitations of Java based Partitioner are as follows:

1. As the data point is read from the input file, all the partitions that the point might belong to are calculated in O(d) time, where d is the dimensionality of the data. Therefore the time complexity of Partitioner is approximately O($n \times d$), where n is the dataset size. Note that the time complexity is independent of the total number of partitions. For example, the time taken for a dataset with 25 partitions and a dataset with 100 partitions are comparable.

2. The partition(s), found in O($n \times d$) time, are updated with the data point for further processing. After the update, the data point is no longer needed in this phase. Hence a space complexity of O(1) is achieved by immediately clearing the data point from memory. Since partitioning is performed on a point by point basis, the entire dataset will not be loaded into memory at any point of time. The fact that the memory is not heavily loaded is very helpful when it comes to large volume datasets.

3. The border points might belong to more than one partition due to the overlapping nature of the partitions. As dimensionality of the dataset increases, the border points can belong to multiple partitions (upper bound given by 2^d). Multiple partitions, that a data point belongs to, are found using a recursive algorithm and the time complexity associated with this operation is O($d \times 2^d$). But in the case of low dimensional datasets, there are very few border points that belong to multiple partitions. Also, the number of dimension is negligible compared to the dataset size. The time complexity of the Partitioner is therefore not dominated by this O($d \times 2^d$) recursion logic and hence the overall complexity can still be approximated to O($n \times d$) time and O(1) space.

4. Partitioning is further optimized (explained below) as the storage complexity is just dependent on the dataset size and the number of partition, and is independent of the total number of iterations required for obtaining clusters of different densities.

5. Auto-HDS can accept either a vector space matrix or a distance matrix as input. As mentioned earlier, the data points are partitioned based on the feature space. Hence, unlike Java based Auto-HDS, this implementation cannot just take the distance matrix as the input. Java based Partitioned Auto-HDS will need the vector space input matrix irrespective of the existence of the distance matrix.

Algorithm 3 Partitioner

Input: Dataset (feature space) $\mathbf{x}$, Partition Matrix **partition**, r_ϵ
Initialize: $\mathbf{max}_{axis}, \mathbf{min}_{axis}$
$n_{dp} = length(\mathbf{x})$
$n_{axis} = length(\mathbf{partition})$
$n_p = prod(\mathbf{partition})$
for $i = 1$ to n_{axis} **do**
  $\mathbf{max}_{axis}(i) = max(\mathbf{x}(:, i))$
  $\mathbf{min}_{axis}(i) = min(\mathbf{x}(:, i))$
end for
Initialize $\mathbf{bound}_{axis}$ for storing the boundary info
for $i = 1$ to n_{axis} **do**
  $\mathbf{bound}_{axis}(i, 1) = \mathbf{min}_{axis}(i)$;
end for
// find the boundaries for each partition along all the axes
for $i = 1$ to n_{axis} **do**
  for $j = 1$ to $\mathbf{partition}(i)$ **do**
    $\mathbf{bound}_{axis}(i, j+1) = (\mathbf{max}_{axis}(i) * j)/\mathbf{partition}(i)$
  end for
end for
// track the partition along each dimension
$\mathbf{partition}^{dp}_{axis} = ones(1, n_{axis})$
$\mathbf{final}_{axis} = zeros(1, n_{axis})$
$\mathbf{init}_{axis} = zeros(1, n_{axis})$
for $i = 1$ to n_p **do**
  $\mathbf{subset} = [1 : n_{dp}]$
  for $k = 1$ to n_{axis} **do**
    $\mathbf{init}_{axis}(k) = \mathbf{bound}_{axis}(k, \mathbf{partition}^{dp}_{axis}(k))$
    $\mathbf{final}_{axis}(k) = \mathbf{bound}_{axis}(k, \mathbf{partition}^{dp}_{axis}(k) + 1)$
    $\mathbf{final}_{axis}(k) = \mathbf{final}_{axis}(k) + (1.5 * r_\epsilon)$
    $\mathbf{init}_{axis}(k) = \mathbf{init}_{axis}(k) - (1.5 * r_\epsilon)$
    $\mathbf{subset}_{final} = intersect(find(\mathbf{x}(:, k) >= \mathbf{init}_{axis}(k)), find(\mathbf{x}(:, k) <= \mathbf{final}_{axis}(k)))$
    $\mathbf{subset}_{final} = intersect(\mathbf{subset}_{final}, \mathbf{subset})$
  end for
  $\mathbf{x}_{subset} = \mathbf{x}(\mathbf{subset}_{final}, :)$
  // update the subset data to the file system
  // update the partitions along each dimension
  for $j = 1$ to n_{axis} **do**
    if $\mathbf{partition}^{dp}_{axis}(j) < \mathbf{partition}(j))$ **then**
      for $m = 1$ to $j - 1$ **do**
        $\mathbf{partition}^{dp}_{axis}(m) = 1$
      end for
      $\mathbf{partition}^{dp}_{axis}(j) = \mathbf{partition}^{dp}_{axis}(j) + 1$
      break
    end if
  end for
end for

18.1.1 Optimized Partitioning

For a given r_ϵ, it has been proved that the dense clusters of Partitioned Auto-HDS are identical to the Auto-HDS clusters. In order to perform hierarchical clustering, a list of r_ϵ (represented by $\mathbf{r}_{\epsilon list}$) that is obtained in the $Parameter - Estimator$ phase is used. Let the total number of partitions be $n_{partition}$ and the total number of iterations be n_{iter}. Recall that for a single iteration, $n_{partition}$ partitions are created with the degree of overlap between adjacent partitions determined by r_ϵ. For n_{iter} different r_ϵ values maintained in the list $\mathbf{r}_{\epsilon list}$, the total number of partitions across all the iterations are given by O($n_{partition} \times n_{iter}$). This mandates the need for maintaining ($n_{partition} \times n_{iter}$) separate files in the file system. As interaction with the file system is an expensive operation, communication and maintenance of O($n_{partition} \times n_{iter}$) files is the most expensive step involved in this phase.

Note that $\mathbf{r}_{\epsilon list}$ gives the list of r_ϵ to be used for each iteration in descending order. From Property 13.1, the minimum overlap between adjacent partitions is $3 \times r_\epsilon$. Therefore, an overlap with the adjacent partitions that is greater than $3 \times r_\epsilon$ will still produce the same results. Hence unnecessary I/O File System Communication in this module as well as in the succeeding modules is avoided by just partitioning using the $maximum(\mathbf{r}_{\epsilon list})$. For the first iteration, the partitioning just meets the minimum overlap requirement of $3 \times \mathbf{r}_{\epsilon list}$, whereas for the rest of the iterations, overlap exceeds the minimum overlap requirement. This optimization gives a steep decrease in interaction with the file system as the files are reduced from O($n_{partition}$ $\times$ n_{iter}) to O($n_{partition}$). The decrease in file count improves the performance of Partitioner and reduces I/O communication overhead in Partitioner, SlaveDIVER and Stitcher.

18.2 Future Work

In a future extension, it would be possible to reduce the file system communication further by decreasing the file count from $n_{partition} \times n_{iter}$ to $n_{partition}$. This is possible because with high dimensional large datasets, due to the curse of dimensionality, an overlap of $3 \times r_\epsilon$ involves a lot of unnecessary computation. Recall from Property113.1 that identical results can be achieved with lesser computation and increased performance by using an overlap of just $3 \times r_\epsilon$.

The problem caused by the $Curse of Dimensionality$ can be addressed by maintaining an additional flag for each data point in the final partition. The flag indicates the maximum number of iterations that a data point can be used for. For instance, a flag of 1 indicates that the data point can be used only in the first iteration of the partition, whereas a flag of 2 indicates that the data point can be used in both first and second iterations. Recall that the $\mathbf{r}_{\epsilon list}$ list will be sorted in descending order. In a typical problem, flag is at least 1 for each data point, whereas it is n_{iter} for a few data points. This is explained by the decreasing nature of r_ϵ as the number of iterations increases. At the expense of an additional flag, this optimization will reduce a lot of unnecessary computation that might be performed due to the curse of dimensionality.

19 SlaveDIVER

In this module, the cluster label of each data point at the partition level is obtained using Auto-HDS. The parameters to this phase would include the estimated $\mathbf{r}_{\epsilon list}$ (from the Parameter-Estimator), a subset of the original dataset based on the partition (from Partitioner), distance

matrix and n_ϵ.

The optimizations included in GeneDIVER give very good scalability and performance. In the standard implementation of Auto-HDS, a lot of computation is involved in sorting the distance matrix (one data point at a time for scalability) to find the n_ϵ^{th} neighbor required for computing r_ϵ. We modified the Java code so that it is now possible to pass r_ϵ as a parameter to the GeneDIVER clustering module. Therefore sorting each data point now involves only finding the neighbors within the distance of r_ϵ. Thus the unnecessary computation associated with finding the first n_ϵ neighbors of both dense and non-dense points is avoided. Given that most datasets have few clusters and a large amount of irrelevant data, the overhead associated with sorting non-dense points is reduced.

The Density Shaving algorithm used in this phase is similar to the standard Density Shaving algorithm with a few modifications. The main change to Density Shaving is that $r_{\epsilon list}$ is passed as a parameter and hence the first few steps that are performed for calculating $r_{\epsilon list}$ can be ignored. The overall complexity of the algorithm remains the same but for a considerable reduction in computation. The pseudo-code of this Modified Density Shaving algorithm is shown in Algorithm 4.

Partitioned Auto-HDS results in a compact hierarchy of clusters that are obtained in $\lceil -\frac{\log(n)}{\log(1-r_{shave})} \rceil$ iterations. The SlaveDIVER module operates on one partition at any point of time. The SlaveDIVER module can be summarized as repeated calls to Modified Density Shaving for each r_ϵ in the estimated $r_{\epsilon list}$ (obtained from Parameter-Estimator). Since Auto-HDS typically involves repeated calls to the Density Shaving algorithm, the term 'Modified Auto-HDS' can be used to refer to repeated calls to Modified Density Shaving. The pseudo-code of SlaveDIVER is shown in Algorithm 5.

Recall that there are three types of data points in each partition. When the cluster labels are updated to the file system, labels are updated only for those data points that are classified correctly (Catefory **0** and Category **2**). The cluster label of each data point in Category **1** (may or may not be correctly classified) is updated as zero and is therefore never used in merging the clusters across partitions.

20 Stitcher

The next phase is the Stitcher that stitches the clusters obtained from different partitions in order to generate the final clusters. The pseudo-code of the Stitcher is shown in Algorithm 6. The logic is based on the approach described earlier in Section 20.

20.1 Discussion

In Stitcher, a local copy of the cluster label (initialized to zero) for each data point is maintained in memory. The cluster labels corresponding to each partition are read from the file system sequentially and the labels are updated in the local copy. Some of the key features of this implementation are presented below:

1. Only one partition is considered at a time and hence only the file corresponding to the partition is opened for communication. Once a data point cluster label is read, either the same label is used or a new label is generated and the local copy is updated. As the

Algorithm 4 Modified DS

Input: Distance matrix $\mathbf{M}_S$, n_ϵ, r_ϵ
Output: Cluster labels $\{lab_i\}_{i=1}^n$ corresponding to the n data points.
Initialize: $\{lab_i\}_{i=1}^n = 0$
$n_c = \lceil n(1 - f_{shave}) \rceil$
// sort each data row of the distance matrix such that all the
// neighbors within a distance of r_ϵ are sorted
$[\mathbf{M}_{rad}^{nbr}, \mathbf{M}_{idx}^{nbr}, \mathbf{M}_{len}^{nbr}] = sortrows(\mathbf{M}_S, r_\epsilon)$
// find the data points that has at least n_ϵ neighbors
$idx^{n_\epsilon} = find(\mathbf{M}_{len}^{nbr} \geq n_\epsilon)$
// recover the dense points that has at least n_ϵ
// neighbors within the distance of r_ϵ
$\mathcal{G} = \mathbf{x}(idx^{n_\epsilon})$
// update the dense points count in n_c
$n_c = length(idx^{n_\epsilon})$
/* Lines 17-33: For each point in $\mathcal{G}$, find other dense points
within r_ϵ distance of it and make sure they have the same
labels, if not, relabel */
for $i = 1$ to n_c **do**
  /* Find the position of the last point within
  distance r_ϵ of dense point $\mathbf{x}(idx^{n_\epsilon}(i))$. */
  $idxb = \mathbf{M}_{len}^{nbr}(i)$
  /* Neighbors of $\mathbf{x}(idx^{n_\epsilon}(i))$ are the $idxb$ closest points, all
  within r_ϵ distance. */
  $\mathcal{X}_{nbrs} = \mathbf{M}_{idx}^{nbr}(idx^{n_\epsilon}(i), l)_{l=1}^{idxb}$
  // save the neighbors
  // Identify neighbors that are dense points
  $\mathcal{X}_{dnbrs} = \mathcal{X}_{nbrs} \cap \mathcal{G}$
  // Recover their labels that are not 0
  $L_{dnbrs} = unique(lab(\mathcal{X}_{dnbrs}))/\{0\}$
  // Relabel all points that share this label to label i
  $\forall y \in \mathbf{lab}$ if $\exists y \in L_{dnbrs} : y = i$
  $\mathbf{lab}(indexOf(\mathcal{X}_{dnbrs})) = i$
end for
Count clusters: $k = |unique(\mathbf{lab})|/\{0\}$
Remap the non-zero labels in **lab** to the range 1 to k.

Algorithm 5 SlaveDIVER

Input: n_ϵ, $r_{\epsilon list}$, $n_{partition}$
Output: Cluster labels $\{lab_i\}_{i=1}^{n}$ corresponding to the n data points.
Initialize: $lab = 0$
for $p = 1$ to $n_{partition}$ **do**
Compute Distance matrix: $\mathbf{M}_S^p$
// compute the total number of iterations
$n_{iter} = length(r_{\epsilon list})$
// call Modified Density Shaving for each r_ϵ
for $i = 1$ to n_{iter} **do**
// extract the r_ϵ value
$r_\epsilon = r_{\epsilon list}(i)$
// extract the cluster labels for each r_ϵ on all partitions
lab_i^p = ModifiedDS($\mathbf{M}_S^p$, n_ϵ, r_ϵ)
end for
end for

data point can be immediately cleared from memory, it is not required to load the entire partition into memory.

2. As discussed in the algorithm, once a new label is generated, the data points with the old labels are updated. Hence whenever stitching is performed, it is mandatory to perform a linear search to extract the subset with the old labels. Therefore the time complexity associated with this operation is O($n \times cluster_{avg}$), where n is the original dataset count and $cluster_{avg}$ is the average number of clusters that are spread across partitions. The time complexity associated with this merging of clusters across partitions increases with increase in the complexity of the dataset. This problem has been addressed by maintaining a two way association such that given a label, the list of data points with the label can be retrieved in time O(1). This two way association is obtained by maintaining an array of labels and a hash map of lists of data points. Though the space complexity is increased by n, the overall space complexity is still O(n), but the time complexity has been reduced considerably from O($n \times cluster_{avg}$) to O($cluster_{avg}$). A linear search is not required whenever a new label is generated; however, cleanup should be performed immediately. The hash map has to be updated with the new label and cleared of the old labels, but still, these operations can be performed in constant time. Hence this two way association gives considerable improvement in performance, even though the space complexity is increased by n.

21 Auto-HDS

With Stitcher, the cluster labels generated are as accurate as the Auto-HDS algorithm. The next task is to smoothen the clusters and to generate a compact hierarchy of clusters. In Partitioned Auto-HDS, the Auto-HDS hierarchy generation module of GeneDIVER has been reused to generate a compact hierarchy of clusters and this hierarchy is presented to the user using the interactive 2D visualization framework that comes with the GeneDIVER.

Algorithm 6 Stitcher

Input: $n_{partition}$, n_{iter}
Output: Clusters $\mathbf{label} - n \times n_{iter}$ matrix
Initialize: $\mathbf{label} = 0$
Initialize: $lab_c = 0$
for $p = 1$ to $n_{partition}$ **do**
Extract from File System: $\mathbf{index}_p$, $\mathbf{lab}_p$
// generate a new set of labels to make sure labels across
// all partitions are unique - usually by adding lab_c
$\mathbf{lab}_{new} = \mathbf{lab}_p + lab_c$
$lab_c = lab_c + length(unique(\mathbf{lab}_p))$
// track the data points that belong to this partition
$n_c = length(\mathbf{index}_p)$
// perform stitch on all iterations
for $i = 1$ to n_{iter} **do**
for $d = 1$ to n_c **do**
// find the global index from the local index
$global = \mathbf{index}_p(d)$
if $\mathbf{label}_i(global) == 0$) **then**
// point does not lie in the Section B - straightforward
$\mathbf{label}_i(global) = \mathbf{lab}_{new}(d)$
else
if $\mathbf{label}_i(global) \neq \mathbf{lab}_{new}(d)$ **then**
// point lies in Section B
// find the points that should be relabeled
$idx_1 = find(\mathbf{lab}_{new} == \mathbf{lab}_{new}(d))$
$idx_2 = find(\mathbf{label}_i == \mathbf{label}_i(global)$
// generate a new unique label and update the points
$lab_c = lab_c + 1$
$\mathbf{lab}_{new}(idx_1) = lab_c$
$\mathbf{label}_i(idx_2) = lab_c$
end if
end if
end for
end for
end for

22 Evaluation

The results for Java based Partitioned Auto-HDS are described in this section. A brief introduction of the datasets used in the experiments and results from scalability testing are presented in this section.

22.1 Datasets

We have used two datasets for testing Partitioned Auto-HDS. A brief summary of the datasets used is presented in the Table 22.1. The Sim-2 dataset [10] is an artificial dataset generated using five 2-D Gaussian distributions of different variances, where each distribution can be considered as a cluster. Since the ground truth of the Sim-2 dataset is known, this dataset is very useful in the verification phase of the clustering algorithms. In Table 22.1, n refers to the dataset size, d refers to the dimension and D refers to the distance measure employed in Partitioned Auto-HDS.

Table 1: Datasets used for evaluating Java based Partitioned Auto-HDS.

Dataset	Source	n	d	D
Halo	Astronomy	600-18,000	3	Euclidean
Sim-2	Synthetic	1,298	2	Euclidean

22.2 Results

From Figure 8, it can be seen that as number of partitions increases, Partitioned Auto-HDS takes relatively less time compared to Auto-HDS. However after a certain point, the scale-up achieved is very small and this can be attributed to the Stitcher module. The time complexity of the Stitcher is heavily dependent on the data points in the overlapping region between any two adjacent partitions. With increase in the number of partitions, the overlapping region volume increases and hence the number of overlapping data points increases. Figure 8 demonstrates that a scale-up factor as high as 4 is easily obtained on a dataset of size 17500.

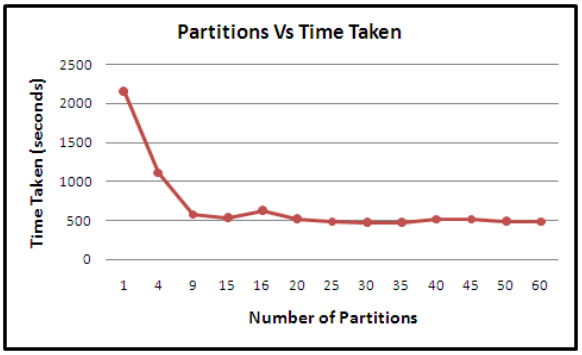

Figure 8: Execution time of Java Based Partitioned Auto-HDS for varying number of partitions on the 2-D Sim-2 Dataset of size 17.5k.

Figure 9 shows the average running times of Auto-HDS and Partitioned Auto-HDS on the Sim-2 dataset of various sizes. It can be seen from Figure 9 that as the dataset size increases, the scale-up achieved with Partitioned Auto-HDS relative to Auto-HDS increases. Scale-up factor as high as 2.5 is obtained in Figure 9. However these experiments were performed on 2-D datasets with a fixed number of partitions (2*2) and from Figure 8, it is evident that the scale-up achieved increases with increase in the number of partitions.

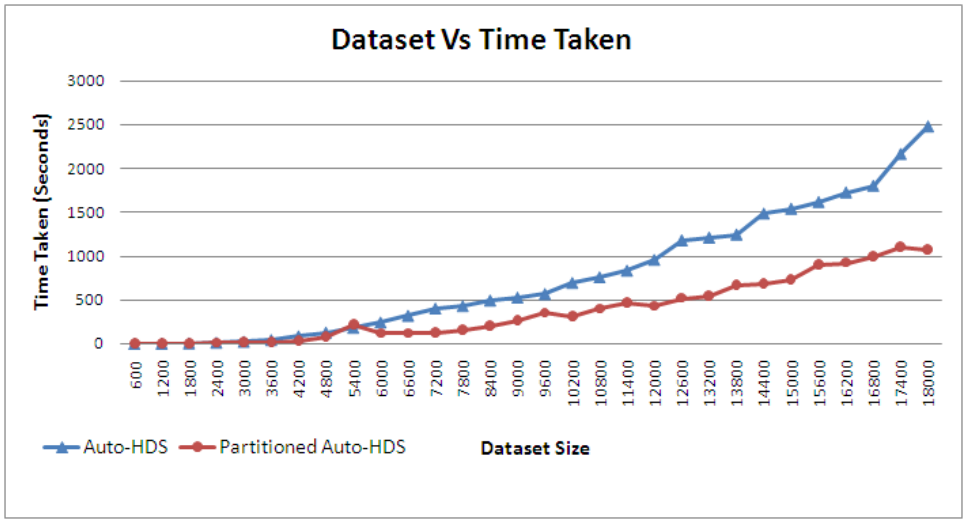

Figure 9: Execution time of Java Based Auto-HDS and Partitioned Auto-HDS for varying size of the 2-D Sim-2 Dataset (4 Partitions).

As part of scalability testing, Partitioned Auto-HDS was used to identify halos in the astronomy dataset and the experimentation results are shown in Table 22.2. It can be seen that it takes approximately 70-80 minutes to find clusters on the dataset of size in 100Ks using a normal desktop machine, whereas Auto-HDS on a normal desktop cannot handle such a huge dataset due to the huge computational and storage complexity associated with it. Notice that the number of partitions is in 100s and it is important to recall that the scale-up increases with increase in the number of partitions. Hence there is a possibility of achieving a better scale-up if more number of partitions are used to solve the astronomy problem.

Table 2: Execution Time of Java based Partitioned Auto-HDS on the Astronomy Dataset.

Dataset	Size	Time (minutes)	Partitions
Halo100	110K	77	1000
Halo125	200K	83	500
Halo150	350K	90	1000

23 Parallel Auto-HDS - A Distributed Implementation using Map-Reduce

24 Introduction

In this chapter, we present Parallel Auto-HDS, an extension to Partitioned Auto-HDS (discussed in Chapter 9) that takes advantage of the inherent parallelism in Partitioned Auto-HDS. A typical implementation of Parallel Auto-HDS will involve multiple machines and therefore a single machine implementation of Parallel Auto-HDS is identical to Partitioned Auto-HDS except for the initialization costs. A Hadoop Map-Reduce based Parallel Auto-HDS has been implemented and this implementation re-uses a few modules of Java based Partitioned Auto-HDS. An introduction to Parallel Auto-HDS and Hadoop Map-Reduce framework followed by the implementation details of the Hadoop Based Parallel Auto-HDS is discussed in the following sections.

25 Parallel Automated Hierarchical Density Shaving (Parallel Auto-HDS)

Similar to Partitioned Auto-HDS, Parallel Auto-HDS includes the following modules:

- Parameter-Estimator
- Partitioner
- SlaveDIVER
- Stitcher
- Compact-HDS

The massive scope for parallelization in the SlaveDIVER and Stitcher modules have been exploited to a certain extent in Parallel Auto-HDS. The modules that have been modified in Parallel Auto-HDS are being discussed in this chapter.

25.1 SlaveDIVER

In this module, dense clusters are identified by repeated calls to Modified Density Shaving (Algorithm 4) on each partition. Since the partitions are independent of each other, Modified Auto-HDS can be run on multiple partitions simultaneously using groups of hundreds and thousands of machines. Parallel Auto-HDS on a distributed environment gives an almost linear speed-up with increase in the number of machines.

25.2 Stitcher

Stitching, like discussed in Partitioned Auto-HDS, includes three stages - Stitching Between Partitions, Stitching Along One Dimension and Stitching Along Multiple Dimensions. Stitching Between Partitions and Stitching Along One Dimension have been optimized further and this optimization makes best use of the distributed environment.

25.2.1 Parallel Implementation and Stitching (Between Partitions) Optimization

A typical parallel implementation of SlaveDIVER involves a group of machines that independently run Modified Auto-HDS on each partition to find dense clusters. In the stitching logic discussed in Section 13, the amount of communication between any two partitions is linearly dependent on the dense points in Section B. An improvement in performance is possible if there is much less communication between machines in the distributed environment. This optimization is achieved by identifying the clusters in Section B. Once the clusters are identified, the first point in each cluster (based on the feature space) and the corresponding cluster labels from PartitionOne and PartitionTwo are determined. In the stitching module, with labels of the first point from each cluster as input, a new label 'LabelNew' is presented as the output. The next step is to relabel the dense points with the old labels from PartitionOne and PartitionTwo with the new label. This way, the clusters that are spread across PartitionOne and PartitionTwo are merged and by giving a new label (usually greater than the total number of clusters identified so far), it is easy to keep track of the new merged clusters as well as to maintain the unique cluster label invariant.

The correctness of this optimization step is proved as follows: The dense points of the same cluster should have the same label. This optimized stitching involves generating a new label for the two labels passed to the algorithm. The major step in this algorithm is therefore to ensure that the labels that are passed should belong to the same cluster. Recall that the clusters and dense points in Section B are correctly identified. By considering the first point in each cluster of Section B from PartitionOne and PartitionTwo, it is guaranteed that the (old) cluster labels of the same dense point and hence the same cluster is passed to the algorithm. The dense points of the same cluster from PartitionOne and PartitionTwo are finally relabeled to the new label returned from the algorithm. Hence the correctness of this optimization is proved.

In this case, it is evident that the amount of communication between partitions is totally dependent on the number of clusters in Section B, as just the first point in each cluster is required. For this stitching to be implemented in a distributed environment, the first dense point from each cluster (in Section B of the overlap) along with its labels from PartitionOne and PartitionTwo has to be communicated. The final output of this operation is a new label 'LabelNew' for the old set of labels (LabelOne and LabelTwo from both the partitions) for the clusters in Section B.

25.2.2 Stitching Along One Dimension

In Partitioned Auto-HDS, Stitching along one dimension is performed using Linear Traversal (Section 13.2). We present Hierarchical Traversal that improves the overall performance of Stitching when used in a distributed environment.

25.2.3 Hierarchical Traversal

In this case, stitching along a dimension is performed in a hierarchical fashion. The logic behind Hierarchical Traversal is that when the partitions 1 and 2 are stitched, the partitions (3, 4), (5, 6), etc., can be stitched in parallel as they are independent of each other. Please note that once the partitions are stitched, they can be considered as a single unit. For example, the partition (1) and (2), once stitched, can be represented as (1, 2).

Therefore, if there are 8 partitions along a dimension, the initial representation is (1), (2), (3), (4), (5), (6), (7), (8). After the first round of stitching (that involves multiple parallel stitches), the state can be represented as (1, 2), (3, 4), (5, 6), (7, 8). A clearer and easier representation of the same is presented below.

```
Round 0: (1), (2), (3), (4), (5), (6), (7), (8)
Round 1: (1 ,  2), (3 ,  4), (5,   6), (7,   8)
Round 2: (1,   2,   3,   4) ,(5,   6,   7,   8)
Round 3: (1,   2,   3,   4,   5,   6,   7,   8)
```

It is important to note that all the stitches (represented by brackets) that belong to the same round can be performed in parallel.

The total number of stitches performed is $p - 1$, which is the same as Linear Traversal. However, there is massive scope for parallelization in this approach as stitches can be performed simultaneously and independently in each round using multiple machines. Hence the time complexity associated with this traversal is not dependent on the total number of stitches required. But it is dependent on the total number of rounds involved, which is given by $O(logp)$. The overall complexity associated with this approach can therefore be approximated as O(log(p) $\times$ $\mathbf{clus}_{avg}$) where $\mathbf{clus}_{avg}$ is the average number of clusters in Section B.

To summarize, two main categories of stitching, namely hierarchical and linear stitching, have been discussed so far. The same amount of computation is involved in both the stitching techniques; however, hierarchical stitching has an edge over linear stitching in a distributed environment.

25.2.4 Stitching Along Multiple Dimensions

Stitching is performed along one dimension at a time and is similar to the Stitching performed in Partitioned Auto-HDS. However with hierarchical stitching, the overall performance of this module can be improved further. On a 2-D dataset with m $\times$ n partitions, if linear stitching is performed, the time complexity can be approximated as O(m $\times$ n $\times$ $\mathbf{clus}_{avg}$) whereas the time complexity of hierarchical stitching is given by O(log(m) $\times$ log(n) $\times$ $\mathbf{clus}_{avg}$). It can therefore be seen that Hierarchical stitching gives increased speed-up on large volume datasets.

25.3 Conclusion

Parallel Auto-HDS is similar to Partitioned Auto-HDS, with the single machine implementation of Parallel Auto-HDS being identical to that of Partitioned Auto-HDS. Partitioned Auto-HDS optimizes Auto-HDS by using the divide-and-conquer approach. Parallel Auto-HDS takes a step further and optimizes Auto-HDS by using the divide-and-conquer approach and using groups of machines that make effective use of the massive scope for parallelization in Partitioned Auto-HDS. Using multiple machines, Parallel Auto-HDS can easily scale up to applications that involve billions of data points.

26 Introduction to Map-Reduce

In early days, serial programs were used where one instruction was executed after another. Such serial programs were sufficient to solve medium size clustering problems in reasonable

time. However these days, some clustering problems involve large amounts of data in terabytes and petabytes [12]. Serial execution is very slow for today's problems that involve processing large amounts of data. Parallel programming reduces the time complexity by making better use of hardware resources and will usually involve multiple processors. Given that the cost of medium hardware is relatively cheap compared to a supercomputer, parallel programming that uses multiple cheap machines is the solution to handle cumbersome tasks. Map-Reduce is one such parallel programming framework that has been used to implement Parallel Auto-HDS.

Map-Reduce [6] is a parallel programming framework developed by Google that provides a simple abstraction that hides the complications behind distributed systems like task scheduling, fault tolerance, task monitoring and load balancing. The simple abstraction enables the user to be concerned only about the computations involved in the problem and not to worry about the huge data and the multiple machines employed to solve the problem. Map-Reduce is getting popular in the machine learning community with numerous machine learning algorithms being implemented in Map-Reduce [4]. Hadoop [`hadoop.apache.org`] is an open source Java implementation of Map-Reduce software framework.

With a single machine, Parallel Auto-HDS performs well compared to Auto-HDS. Parallel Auto-HDS has massive parallelism inherent in the algorithm that can be effectively used in a distributed system. Taking advantage of the parallelism, a Hadoop Map-Reduce based Parallel Auto-HDS has been implemented to handle large volume datasets. The implementation details of Hadoop Map-Reduce based Parallel Auto-HDS are presented in this chapter.

27 Background - Map-Reduce

Map-Reduce is a parallel programming framework developed for processing huge volumes of data using hundreds or thousands of machines together. Map-Reduce is a functional programming framework that solves a problem in terms of map and reduce functions. The communication to, from and between the mapper and reducer functions is in terms of $\langle key, value \rangle$ pairs. The user input is the input to the *Mapper* stage and the *Reducer* stage output is the final output that is presented to the user. The intermediate output from the *Mapper* stage is the input to the *Reducer* stage.

```
map: <key_in, value_in> -> <key_inter_out, values_inter_out>
reduce: <key_inter_out, list<values_inter_out>> -> <key_out, value_out>
```

A quick overview of the Map-Reduce framework [6] (Figure 10) is presented to get a clear picture of the restricted data flow model. Map-Reduce is based on the master-slave architecture with a single master JobTracker and multiple slaves, also known as worker nodes. JobTracker schedules tasks on the mapper/reducer worker nodes, re-schedules tasks on the workers in case of a failure, monitors and reports the progress. With the user input, the JobTracker schedules tasks on the mapper worker nodes. Once the TaskTracker in the mapper worker nodes completes the assigned task, the intermediate output file locations are reported to the JobTracker. With the mapper output from JobTracker, the TaskTracker in the reducer worker nodes sorts the intermediate data based on the key and processes the data. The final output file locations from the reducer stages are reported back to the JobTracker.

In addition to the mapper and the reducer stages, there is an optional stage called the combiner. The combiner is similar to the reducer functionally and is basically an optimization

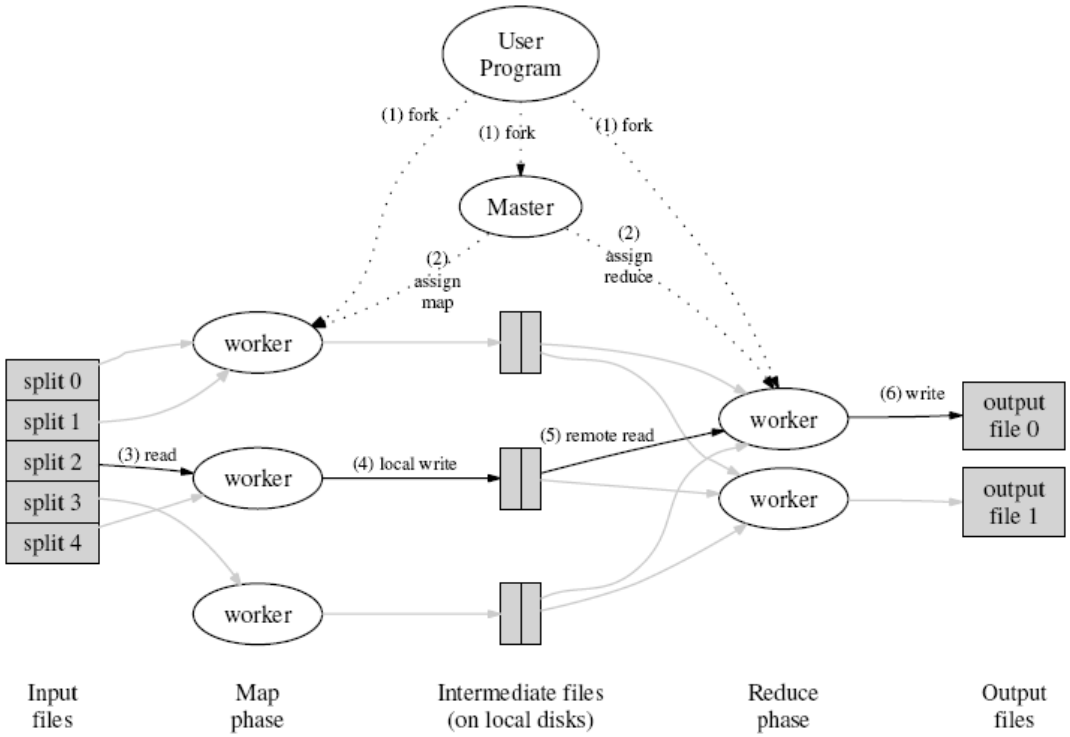

Figure 10: Execution Overview - Source [6]

step. In most cases, the combiner class is same as the reducer class. The only difference between the combiner and the reducer stage is that the *combine* job is run on each mapper output independently, whereas the *reduce* job is applied to the accumulated output from the mapper stage. By adding the combiner stage, the amount of communication from the mapper to the reducer stage is reduced to a reasonable extent. Also, notice that as the reduce jobs work on the accumulated output from the mapper stage, the reducer stage cannot start before the user input is completely converted into mapper intermediate output.

27.1 Example

Before getting into the implementation details of Hadoop Map-Reduce based Parallel Auto-HDS, a simple Map-Reduce application is presented for better understanding of the Hadoop Map-Reduce framework. WordCount application [6], that counts the frequency of each word in the documents provided, is the most common example used for explaining the hadoop architecture. The pseudo-code of the 'WordCount' application is shown below:

```
Mapper(String key_in, String value_in):
// key_in: document name
// value_in: document contents
for each word w in value_in:
EmitIntermediate(w, "1");
```

```
Combiner(String key_inter_out, Iterator values_inter_out):
// key_inter_out: a word
// values_inter_out: a list of counts
int result = 0;
for each v in values_inter_out:
result += ParseInt(v);
Emit(AsString(result));

Reducer(String key_inter_out, Iterator values_inter_out):
// key_inter_out: a word
// values_inter_out: a list of counts
int result = 0;
for each v in values_inter_out:
result += ParseInt(v);
Emit(AsString(result));
```

Input to the 'WordCount' application is a list of documents. The mapper will process one line of input at any point of time. Each mapper takes 'document name + line number' as the input key and the document content as the input value. In the mapper function, for each word in the document content, a value of '1' is emitted as the intermediate output. Notice that there is an additional stage - combiner - before the reducer and in this case, combiner class is same as reducer at the functional level. This combiner works on the output of each mapper before moving onto the reducer stage thereby cutting down on the network communication. In the reducer function, the intermediate output values are added to get the frequency of each word in the document.

```
Input to the Map-Reduce application includes 2 documents:
document 1: Hadoop Reduce Hadoop Reduce
document 2: Map Hadoop Map Hadoop

For instance, the output of the first mapper:
<Hadoop, 1>
<Reduce, 1>
<Hadoop, 1>
<Reduce, 1>

The output of the second mapper:
<Hadoop, 1>
<Map, 1>
<Hadoop, 1>
<Map, 1>

With a combiner stage, the optimized first mapper output:
<Hadoop, 2>
<Reduce, 2>
```

```
With a combiner stage, the optimized second mapper output:
<Hadoop, 2>
<Map, 2>

The final output from the reducer:
<Hadoop, 4>
<Map, 2>
<Reduce, 2>
```

WordCount application, once written in the form of mapper and reducer functions, can be effortlessly scaled to huge number of machines.

28 Hadoop based Parallel Auto-HDS

The main ideas discussed in Parallel Auto-HDS are generic and may have to be modified to a certain extent to suit the distributed environment. Since Hadoop Map-Reduce framework has a pre-determined and restricted data flow model, not all the ideas discussed in Parallel Auto-HDS have been implemented in the Hadoop based Parallel Auto-HDS. As discussed in Section 25, Parallel Auto-HDS includes the following modules:

- Parameter-Estimator
- Partitioner
- SlaveDIVER
- Stitcher
- Compact-HDS

Of all the modules specified above, the most time consuming and parallelizable operations are performed by the Partitioner, SlaveDIVER and Stitcher. Multiple Map-Reduce jobs are used to implement the Partitioner, SlaveDIVER and Stitcher. The reasons for not handling the Compact-HDS and Parameter-Estimator modules in the Map-Reduce environment are as follows: In Compact-HDS, the input data should be loaded into memory at a single point for compacting the clusters. However in Map-Reduce, the input data is usually split into fixed width blocks called shards and the shards are distributed across multiple machines. Hence a distributed system like Map-Reduce cannot be used to implement Compact-HDS. Since Parameter-Estimator has low time complexity relative to the other modules, a map-reduce job is not used for this phase.

28.1 Implementation Details

In a map-reduce application, multiple jobs can be configured within the same application. Each job will have the mapper, combiner (optional) and reducer classes defined in the job configuration. Recall that the reducer stage will begin right after the completion of the mapper stage and therefore both the mappers and the reducers cannot run in parallel. Similarly, even

though multiple jobs can be configured in a map-reduce application, jobs are executed in a serial fashion. In a map-reduce application, if the input to each job is the same, reducing the number of jobs will improve the performance to a considerable extent as unnecessary network communication and computation are avoided.

The main tasks that are to be solved using Map-Reduce involve extracting a subset from the dataset (for the Parameter-Estimator phase), dividing the dataset into partitions (Partitioner), finding the cluster labels of each partition (SlaveDIVER) followed by stitching (Stitcher). To perform these, three map-reduce jobs have been designed and each of these will be explained in the following sections. The first map-reduce job performs some pre-processing required for the Parameter-Estimator, Partitioner and Stitcher phases. The second map-reduce job includes both the Partitioner and SlaveDIVER, whereas the third map-reduce job is the Stitcher.

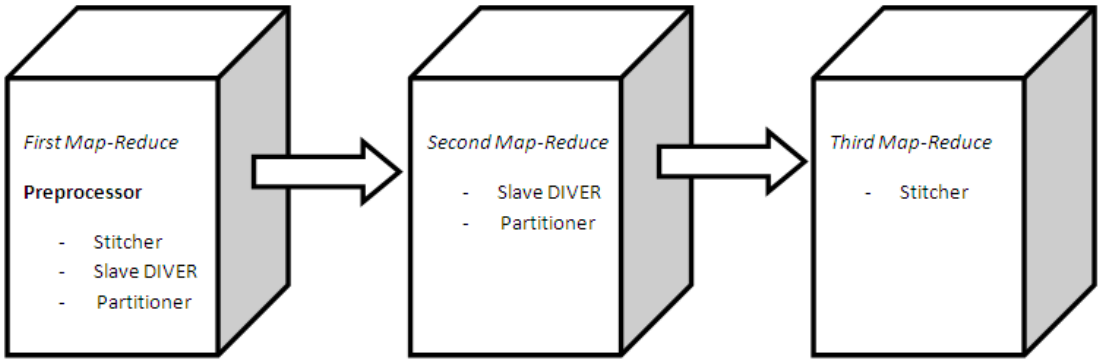

Figure 11: Overview of Hadoop Map-Reduce Based Parallel Auto-HDS

28.1.1 First Map-Reduce Job: Preprocessor for the Parameter-Estimator, Partitioner and Stitcher

For Parameter-Estimator phase, a subset of the dataset is extracted so that an estimate of $\mathbf{r}_{\epsilon list}$ is obtained by running GeneDIVER on the subset. This is achieved by randomly selecting data points using a random number generator and 'random rate' parameter as discussed in Section 17. For the Partitioner, in order to divide the dataset into multiple partitions, the maximum and minimum along each dimension are required (Algorithm 3). For the Stitcher, dataset size is required for tracking cluster labels.

Notice that the input to subset extraction (for Parameter-Estimator), maximum and minimum along each dimension (for Partitioner) and dataset size (for Stitcher), is the original dataset. In an effort to reduce the number of map-reduce jobs, these three tasks have been combined to a single map-reduce job. The pseudo-code of the first map-reduce job is shown below:

```
Mapper(String key_in, String value_in):
// key_in: line number
// value_in: data point feature space (list)
for each dim in dimension_list:
```

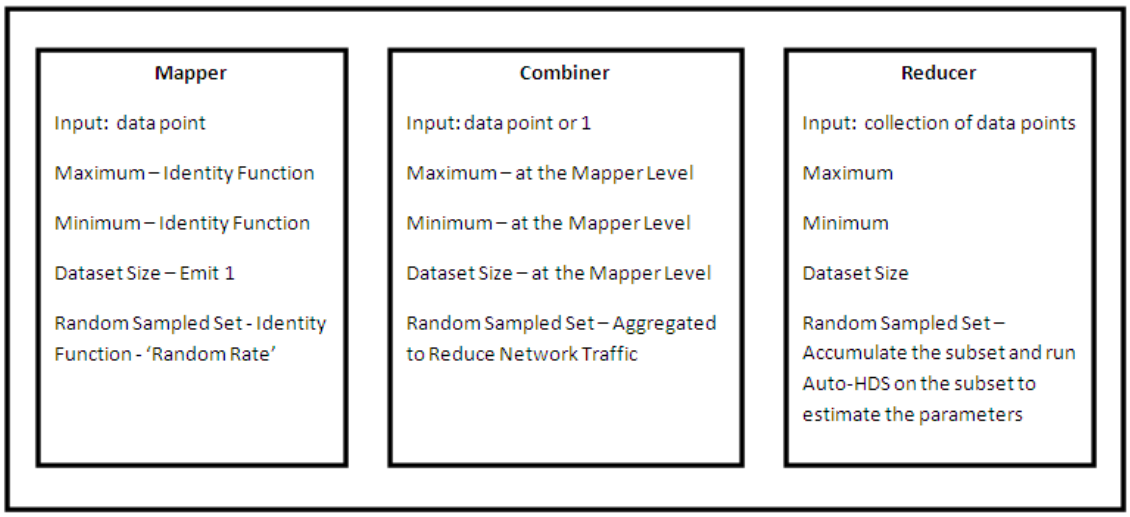

Figure 12: First Map-Reduce Job - Parallel Auto-HDS

```
EmitIntermediate(dim, (value_in(dim), value_in(dim)));

// extract subset
value = Math.rand();
if (value < random_rate)
EmitIntermediate(dim + 1, value_in);

// calculate total
EmitIntermediate(dim + 2, 1);

Combiner(String key_inter_out, Iterator values_inter_out):
// key_inter_out: a dimension
// values_inter_out: a list of data points
int min_result = 0;
int max_result = 0;
int total = 0;
list subset;

// max and min calculation
if (key_inter_out is a dimension)
for each v in values_inter_out:
min_result = Math.min(ParseInt(v), min_result);
max_result = Math.max(ParseInt(v), max_result);
end for
Emit(AsString((min_result, max_result)));
elseif (key_inter_out is (dimension + 1))
// subset extraction
```

```
for each v in values_inter_out:
subset.add(v);
Emit(AsString(subset));
else
// dataset count
for each v in values_inter_out:
total += 1;
Emit(AsString(total));
endif

Reducer(String key_inter_out, Iterator values_inter_out):
// key_inter_out: a dimension
// values_inter_out: a list of data points
int min_result = 0;
int max_result = 0;
int total = 0;
list subset;

// max and min calculation
if (key_inter_out is a dimension)
for each v in values_inter_out:
min_result = Math.min(ParseInt(v), min_result);
max_result = Math.max(ParseInt(v), max_result);
end for
Emit(AsString((min_result, max_result)));
elseif (key_inter_out is (dimension + 1))
// subset extraction
for each v in values_inter_out:
subset.add(v);
Emit(AsString(subset));
else
// dataset count
for each v in values_inter_out:
total += 1;
Emit(AsString(total));
endif
```

Notice that there is a combiner stage in addition to the mapper and the reducer stages. The combiner has been added to reduce the volume of communication between the mapper and the reducer. The total number of reduce jobs has been set as d+2, where d is the number of dimensions. First d reducer jobs find the maximum and minimum value along each dimension. The $(d+1)^{th}$ reducer job collects the sampled data points to create a subset for Parameter-Estimator. The $(d+2)^{th}$ reducer job calculates the dataset size. In the reducer class, there is an if-else condition that handles this logic.

28.1.2 Second Map-Reduce Job: Partitioner and SlaveDIVER

After the first Map-Reduce job, GeneDIVER is run on the extracted subset for estimating $\mathbf{r}_{\epsilon list}$. In the second Map-Reduce job, the dataset is partitioned into overlapping subsets using the estimated $\mathbf{r}_{\epsilon list}$. The pseudo-code of the second map-reduce job is as follows:

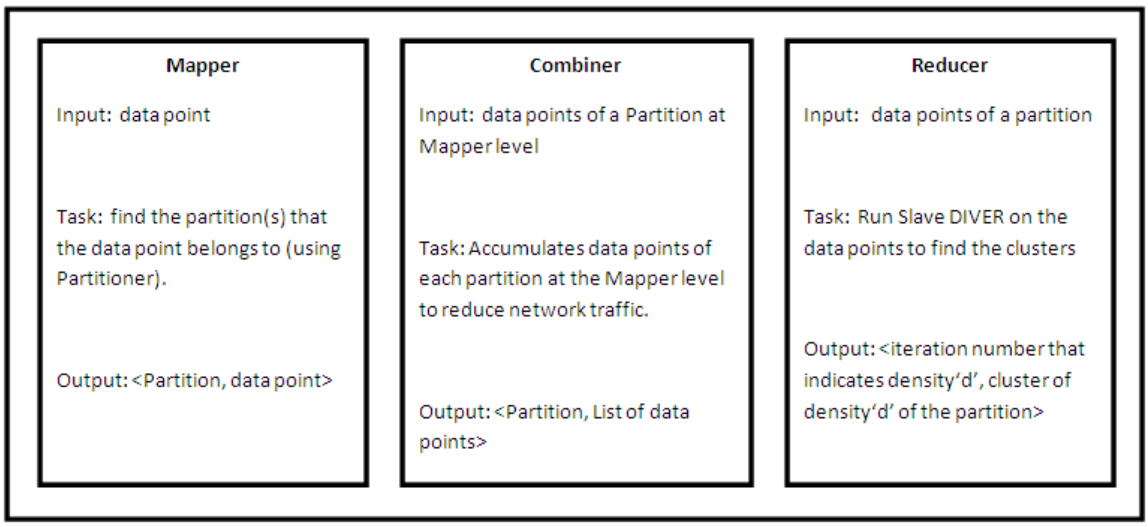

Figure 13: Second Map-Reduce Job - Parallel Auto-HDS

```
Mapper(String key_in, String value_in):
// key_in: line number
// value_in: data point feature space (list)

// find the partitions
partition_list = Partitioner(value_in);
for each partition in partition_list
EmitIntermediate(partition, value_in);

Combiner(String key_inter_out, Iterator values_inter_out):
// key_inter_out: partition
// values_inter_out: a data point
list subset;

// collects the subset
for each v in values_inter_out:
subset.add(v);
Emit(AsString(subset));

Reducer(String key_inter_out, Iterator values_inter_out):
// key_inter_out: partition
// values_inter_out: a data point
```

```
list subset;

// collects the subset
for each v in values_inter_out:
subset.add(v);

// run SlaveDIVER to find dense clusters
clusters = SlaveDIVER(subset);
Emit(AsString(clusters));
```

The *Partitioner* module is used in the mapper, whereas the $SlaveDIVER$ logic is incorporated in the reducer stage. Each data point passed as an input to the mapper uses the 'Partitioner' to determine the partitions that the data point belongs to. Once the partitions are determined, ⟨partition, data point⟩ is emitted for each partition from the mapper stage. The combiner collects the data points based on the 'partition' key at the mapper level. The number of reducer jobs is equal to the total number of partitions created. Each reducer collects the data points of a specific partition based on the input key which denotes the partition. Once the data points are collected, the dense clusters in each partition are identified using $SlaveDIVER$. The clusters generated are presented as the output of the second Map-Reduce job. The Auto-HDS in SlaveDIVER includes multiple iterations that generate dense clusters of different densities. The output of each partition (same as reducer) is in the form of $\langle iteration, cluster - labels \rangle$ pairs where *iteration* is the iteration number and $cluster - labels$ are the clusters found in iteration *iteration*.

28.1.3 Third Map-Reduce Job: Stitcher

The output of the second Map-Reduce Job is the input to the last Map-Reduce Job. Final Map-Reduce job is the stitcher for merging the dense clusters across all partitions. Notice that the optimizations added to the Stitcher module in Parallel Auto-HDS have not been implemented in this Hadoop based Parallel Auto-HDS. This is because the communication among the machines is handled by the Hadoop framework itself and not by the user. The pseudo-code of the third map-reduce job is shown below:

```
Mapper(String key_in, String value_in):
// key_in: iteration
// value_in: data points and cluster labels

// identity function
for each partition in partition_list
EmitIntermediate(key_in, value_in);

Combiner(String key_inter_out, Iterator values_inter_out):
//key_inter_out: iteration
// values_inter_out: list of data points and cluster labels
list label-list;
```

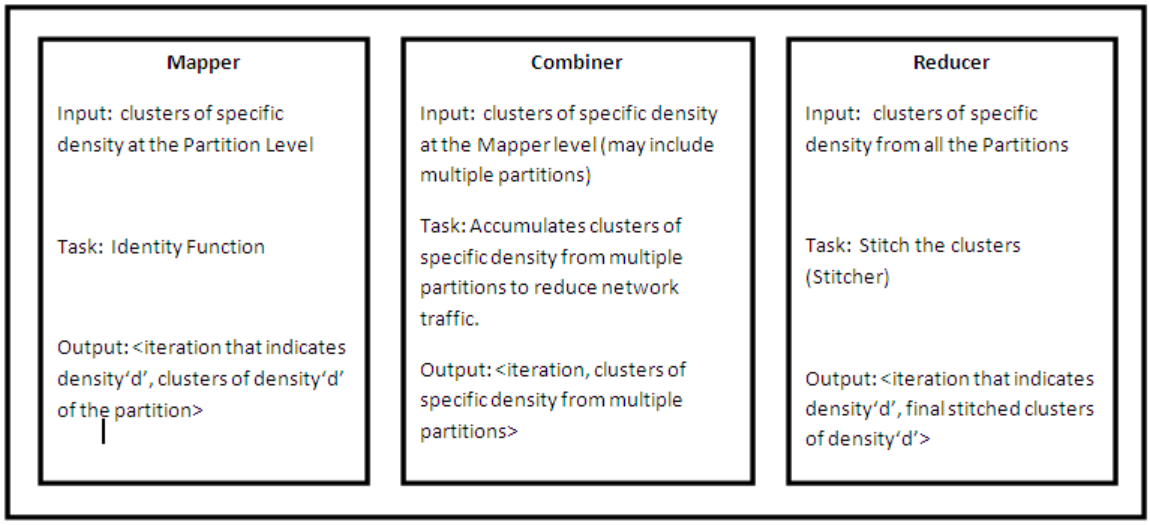

Figure 14: Third Map-Reduce Job - Parallel Auto-HDS

```
// collect the data points and cluster labels
for each v in values_inter_out:
subset.add(v);
Emit(AsString(label-list));

Reducer(String key_inter_out, Iterator values_inter_out):
//key_inter_out: iteration
// values_inter_out: list of data points and cluster labels
list label-list;

// list of data points and cluster labels from each partition
for each v in values_inter_out:
label-list.add(v);

// perform stitching
clusters = Stitcher(label-list);
Emit(key_inter_out, AsString(clusters));
```

The input to the mapper class is the iteration number and the cluster label of each data point in the partition (second map-reduce output). The mapper acts as an identify function and therefore the mapper input $\langle iteration, cluster - labels \rangle$ is re-emitted as the intermediate output. In the combiner stage, labels corresponding to an iteration (r_ϵ) are collected at each mapper level. The number of reducer jobs is equal to the total number of iterations that is determined using $\mathbf{r}_{\epsilon list}$. In the reducer stage, clusters corresponding to an iteration (r_ϵ) from the partitions are collected and stitched. In other words, each reducer stitches the clusters of a specific density corresponding to the iteration given by key_inter_out. The final stitched clusters of different densities are presented as the output of the third Map-Reduce job.

28.1.4 Compact-HDS

Since clusters from each partition should be loaded into the memory for compacting and generating a hierarchy of dense clusters, no Map-Reduce jobs are used for this operation. Once the clusters identical to the clusters in Auto-HDS are obtained from the Stitcher, the Java based Auto-HDS is reused to generate the compact hierarchy of the dense clusters identified.

29 Experiments

The term 'speed-up' can be defined as the ratio of the time taken by one processor to the time taken by multiple processors for a computation. With Parallel Auto-HDS in a distributed environment, a speed-up proportional to the number of cores used is achievable.

29.1 TACC Hadoop Map-Reduce Framework

The Hadoop Map-Reduce Implementation of Parallel Auto-HDS was initially tested in the pseudo-distributed mode on a single machine. After the initial testing, the Hadoop Map-Reduce Parallel Auto-HDS was tested in a distributed environment, Longhorn, provided by Texas Advanced Computing Center (TACC). The Longhorn system consists of 256 dual-socket nodes, each with significant computing and graphics capability. Total system resources include 2048 compute cores (Nehalem quad-core), 512 GPUs (128 NVIDIA Quadro Plex S4s, each containing 4 NVIDIA FX 5800s), 13.5 TB of distributed memory and a 210 TB global file system. Up to 128 compute cores in the Longhorn system have been used for testing the Hadoop Map-Reduce based Parallel Auto-HDS.

29.2 Dataset

The Hadoop Map-Reduce Parallel Auto-HDS was tested on the Sim-2 dataset [10], an artificial dataset generated using five 2-D Gaussian distributions of different variances. The correctness of the implementation was verified by comparing with the results from Partitioned Auto-HDS on relatively small datasets. As part of scalability testing, the implementation was also tested on the Astronomy Halo dataset [12] that has information about 14 Million celestial bodies. Parallel Auto-HDS was tested on a subset of the Astronomy Halo dataset with the subset size ranging from 50K to 450M data points. The total number of cores used for running the Hadoop Map-Reduce application have been varied ranging from 16 to 128 compute cores. TACC Hadoop Map-Reduce framework has a minimal requirement of 16 cores, as up to 8 cores are used just for scheduling the task, fault tolerance, etc. The additional number of cores provided are used for executing the map-reduce tasks scheduled by the JobScheduler. The correctness of Parallel Auto-HDS on the Astronomy dataset was not tested, as the dataset and the partitions created from the dataset are too large to fit in a single machine (Partitioned Auto-HDS).

29.3 Experiments

Table 29.3 shows the execution time of Parallel Auto-HDS on the Astronomy dataset [12]. The column 'Subset Size' refers to the size of the sampled subset extracted in the Parameter-Estimator phase. Note that the 1500k Halo dataset takes lesser time than the 750k Halo

dataset. This behavior is due to two reasons: the number of partitions created from the dataset and the sampled subset size in the Parameter-Estimator phase. As the number of partitions increases, there is more scope for parallelism in a distributed environment. As the sampled subset in the 'Parameter-Estimator' phase is used for estimating the parameters, a small subset size will result in weak estimation of the parameters, whereas a large subset will increase the computational and storage complexity associated with this phase. Since the overall time complexity associated with Parallel Auto-HDS is much less compared to Auto-HDS, the optimal subset size can be estimated by just re-running Parallel Auto-HDS multiple times. The number of map and reduce tasks was set dynamically based on the dataset size and in most cases, it was usually set higher than the number of cores (maximum of 128 cores) used by the Hadoop Map-Reduce Framework. With more cores, there is scope for improvement in performance, provided the number of map-reduce tasks outweighs the number of cores.

Table 3: Execution Time of Hadoop Map-Reduce Parallel Auto-HDS on the Astronomy Dataset.

Dataset	Size	Time (seconds)	Partitions	Subset Size
Halo50	25k	141.734	27	1/10
Halo100	110k	330.925	125	1/10
Halo150	340k	341.945	125	1/10
Halo200	750k	923.418	375	1/20
Halo200	750k	1009.807	1000	1/10
Halo200	750k	642.317	1000	1/20
Halo250	1500k	555.638	1000	1/100

Figure 15 shows the execution time of Parallel Auto-HDS on the astronomy dataset of size 100k and 125 partitions, with the number of cores varied between 16 and 128. From Figure 15, the execution time decreases gradually with increase in the number of cores used. There is not much improvement in performance with the number of cores ≥ 80 and this could be because there are just 125 partitions in the dataset. Note that 80 cores[14] (roughly half the number of partitions used), with each core running approximately two partitions, seems to be an optimal setting as the curve flattens out beyond 80.

Figure 16 shows the speed-up achieved by Parallel Auto-HDS on the astronomy dataset of size 100k and 125 partitions. It can be seen that there is a close to linear increase in speed-up as the number of cores increases. After a certain threshold, that is approximately equal to half the number of partitions created, there is not much speed-up and hence the curve flattens out beyond this threshold. If the number of cores is greater than half the number of partitions, the additional cores (beyond the threshold) do not improve the speed-up further.

30 Limitations and Future Work

In this implementation, parallelization at the partition level of Parallel Auto-HDS has been exploited. Recollect that in Auto-HDS, there are multiple iterations for generating clusters

[14]note that 8 cores are responsible for job scheduling, fault tolerance, etc., and hence only up to 72 cores are used for the Map-Reduce jobs.

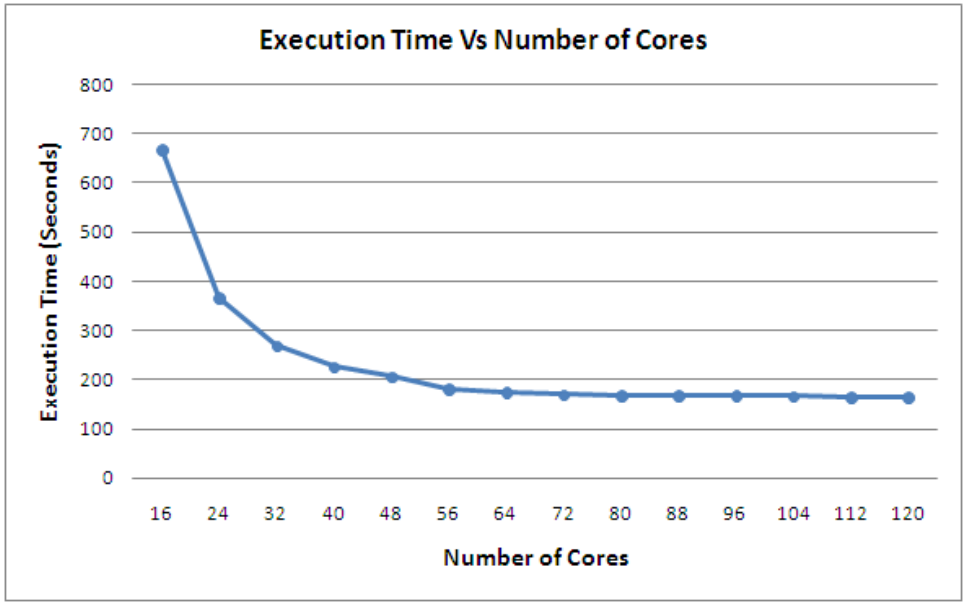

Figure 15: Execution time of Hadoop Map-Reduce based Parallel Auto-HDS for varying number of cores on the Astronomy dataset of size 100k and 125 partitions.

of different densities and these iterations can be run independently. This parallelism at the Auto-HDS level has not been exploited. Exploiting this parallelism will involve updating both the $Parameter - Estimator$ and $SlaveDIVER$ phases, which in turn use Auto-HDS. By performing each iteration in Auto-HDS independently (in parallel), the overall performance and speedup can be improved further. In the $Parameter - Estimator$ and $SlaveDIVER$, the initial step is to sort each data point such that either first n_ϵ neighbors or all the neighbors within a certain distance r_ϵ are in sorted order. Recall that sorting each data point is again independent of any other data point and this fact leaves room for further optimization in a distributed environment. The speed up achieved is dependent on the number of machines used. Hadoop Map-Reduce based Parallel Auto-HDS presented in this thesis does not support the GeneDIVER User Interface that enables interactive visualization of the cluster hierarchy identified. It will be useful to create a GeneDIVER framework with an abstract user interface hiding the technical details of the underlying environment (either a single machine or a distributed environment). Such an interface would also abstract the algorithm used (Auto-HDS or Partitioned Auto-HDS or Parallel Auto-HDS) to find the dense clusters.

31 Conclusions

In this thesis, two extensions to Auto-HDS have been presented for solving large clustering problems that exist in the fields of bioinformatics, astronomy, marketing, etc. A key limitation of both the approaches is that they are not suitable for high dimensional datasets due to the $curse of dimensionality$. A simple and scalable extension to Auto-HDS is Partitioned Auto-HDS that works on smaller subsets of a large dataset to identify the compact hierarchy of dense clusters. With this approach, if data points are uniformly distributed across the feature

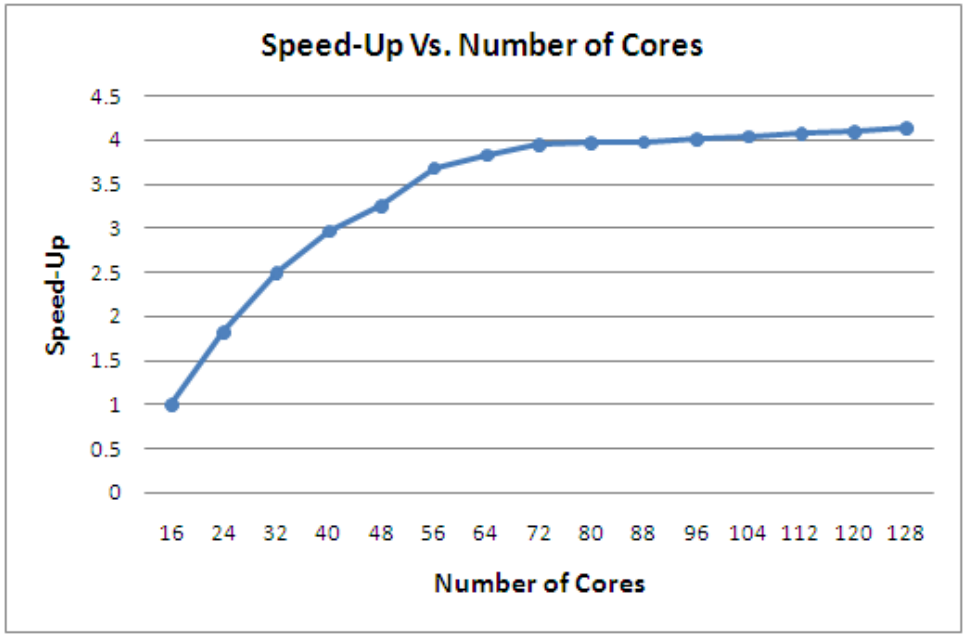

Figure 16: Speed-up of Hadoop Map-Reduce based Parallel Auto-HDS for varying number of cores on the Astronomy Dataset of size 100k and 125 partitions.

space, the computational and storage complexity associated with the operation is reduced by a factor of the number of subsets created. However, if data points are not uniformly distributed, the performance improvement is not as high. Therefore, it would be worthwhile to partition datasets in such a way that data points are uniformly distributed across all partitions.

We presented Parallel Auto-HDS which extended Partitioned Auto-HDS to a distributed environment. Parallel Auto-HDS facilitates the use of a large number of cheap machines in a distributed environment instead of expensive super computers. Parallel Auto-HDS was implemented on a Hadoop Map-Reduce framework and the results were presented in Chapter 23. Experiments revealed that when data points were uniformly distributed across partitions, Parallel Auto-HDS achieved linear speed up with the increase in the number of machines used. It would be interesting to compare the performance of Hadoop Map-Reduce based Parallel Auto-HDS to Parallel Auto-HDS implemented on top of several other Parallel Programming frameworks.

References

[1] *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006), 18-22 December 2006, Hong Kong, China*. IEEE Computer Society, 2006.

[2] M. Ankerst, M. Breunig, H. P. Kriegel, and J. Sander. Optics: Ordering points to identify the clustering structure. In *ACM SIGMOD Conference*, 1999.

[3] Arindam Banerjee, Srujana Merugu, Inderjit S. Dhillon, and Joydeep Ghosh. Clustering with bregman divergences. *J. Mach. Learn. Res.*, 6:1705–1749, 2005.

[4] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[5] Koby Crammer and Gal Chechik. A needle in a haystack: Local one-class optimization. In *In Proc. ICML*, Banff, Alberta, Canada, 2004.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150.

[7] Meghana Deodhar, Gunjan Gupta, Joydeep Ghosh, Hyuk Cho, and Inderjit S. Dhillon. A scalable framework for discovering coherent co-clusters in noisy data. In *ICML*, page 31, 2009.

[8] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *In Proc. KDD-96*, 1996.

[9] Gasch A. P. et al. Genomic expression programs in the response of yeast cells to environmental changes. *Mol. Bio. of the Cell*, 11(3):4241–4257, December 2000.

[10] Gunjan Gupta, Alexander Liu, and Joydeep Ghosh. Hierarchical density shaving: A clustering and visualization framework for large biological datasets. In *accepted, IEEE ICDM 2006 Workshop on Data Mining in Bioinformatics (DMB 2006), 5 pages*, Hong Kong, December 2006.

[11] Gunjan K. Gupta and Joydeep Ghosh. Detecting seasonal trends and cluster motion visualization for very high dimensional transactional data. In *Society for Industrial and Applied Mathematics (First International SIAM Conference on Data Mining (SDM2001))*, April 2001.

[12] Ilian T Iliev, Daniel Whalen, Garrelt Mellema, Kyungjin Ahn, Sunghye Baek, Nickolay Y Gnedin, Andrey V Kravtsov, Michael Norman, Milan Raicevic, Daniel R Reynolds, Daisuke Sato, Paul R Shapiro, Benoit Semelin, Joseph Smidt, Hajime Susa, Tom Theuns, and Masayuki Umemura. Cosmological radiative transfer comparison project ii: The radiation-hydrodynamic tests. Technical Report arXiv:0905.2920, May 2009. Comments: 36 pages, 44 figures (most in color), submitted to MNRAS.

[13] M. Pellegrini, E. M. Marcotte, M. J. Thompson, D. Eisenberg, and T. O. Yeates. Assigning protein functions by comparative genome analysis: protein phylogenetic profiles. *Proc Natl Acad Sci U S A*, 96(8):4285–8+.

[14] B. Schölkopf, C. Burges, and V. Vapnik. Extracting support data for a given task. In *KDD*, Menlo Park, CA, 1995. AAAI Press.

[15] B. Schölkopf, J. C. Platt, J. S. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13(7):1443–1471, 2001.

[16] Alexander Strehl and Joydeep Ghosh. Relationship-based clustering and visualization for high-dimensional data mining. *INFORMS Journal on Computing*, 15(2):208–230, 2003.

[17] D. Tax and R. Duin. Data domain description using support vectors. In *Proceedings of the ESANN-99*, pages 251–256, 1999.

[18] Nicola Vitulo, Alessandro Vezzi, Chiara Romualdi, Stefano Campanaro, and Giorgio Valle. A global gene evolution analysis on *ibrionaceae* family using phylogenetic profile. *BMC Bioinformatics*, 8(S-1), 2007.

[19] D. Wishart. Mode analysis: A generalization of nearest neighbour which reduces chaining effects. In *Proceedings of the Colloquium in Numerical Taxonomy*, pages 282–308, University of St. Andrews, Fife, Scotland, September 1968. Academic Press.