University of Alberta

Efficient Algorithms for Hierarchical Agglomerative Clustering

by

Ajay Anandan

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Abstract

This thesis proposes and evaluates methods to improve two algorithmic approaches for Hierarchical Agglomerative Clustering. These new methods increase the scalability and speed of the traditional Hierarchical Agglomerative Clustering algorithm without using any approximations. The first method exploits the characteristics of modern Non-Uniform Memory Access architectures, resulting in a parallel algorithm for the stored matrix version of Hierarchical Agglomerative Clustering. The second method uses a data structure called the Cover Tree to speed up the stored data version of the Hierarchical Agglomerative Clustering. For the second method, the thesis proposes both sequential and parallel algorithms. All methods were experimentally evaluated and compared against the state-of-the-art approaches for high performance clustering. The results demonstrate the superiority of the parallel approaches with respect to all baselines and previous work, and the comparison between the stored matrix and the stored data approaches illustrate interesting performance trade-offs.

Acknowledgements

I would like to thank my supervisor, Professor Denilson Barbosa, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject and complete this project successfully. I would also like to thank the Department of Computing Science for providing me with the funding for these two years. Finally, I would like to thank my friends and family, whose love and support gave me the best possible platform to work on my Masters.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A massive amount of data is produced everyday and a general question that arises is how to organize this data so that something meaningful can be derived from it. One very good way to solve this problem is *Clustering*, which is the task of finding natural groupings among data items (also called data points or just points), according to some measure of similarity. In other words, clustering is the task of dividing N points with D dimensions into K groups according to some measure of similarity. Clustering has many applications in fields like data mining, gene categorization and data classification and so different clustering algorithms have been developed for these various purposes. Many of these clustering algorithms require the user to specify the number of groups that are present in the data before the execution of the algorithm. But in real life scenarios, the user often does not know the number of groups in the data in advance and so those clustering algorithms are not very effective in those cases. A popular method which does not require the user to specify the number of groups present in the data is Hierarchical Clustering. Hierarchical Clustering produces a hierarchy of clusters which can be visualized by a tree-like hierarchical structure called *Dendogram*. In each level of the dendogram, the two clusters which are closest to each other at that level are merged. Thus slicing the dendogram at any level gives a set of clusters. Figure 1.1 gives an example dendogram.

There are two variations of the Hierarchical Clustering methods:

- *Hierarchical Agglomerative Clustering*: Initially each point is considered

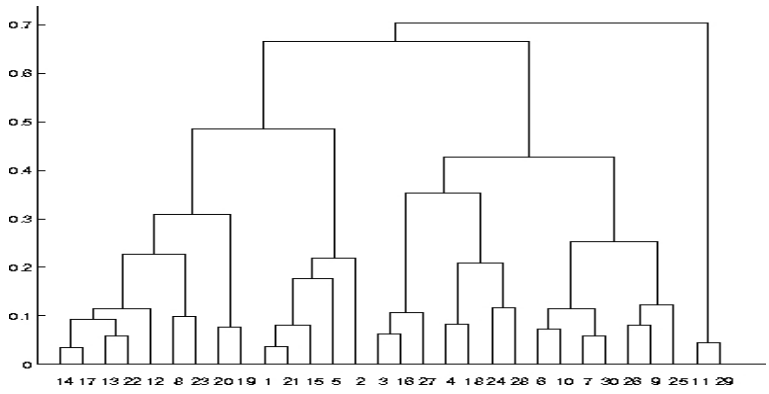


Figure 1.1: An Example Dendogram. Taken from [34]

its own cluster and at each iteration, the two closest clusters are merged to form a single cluster. In other words, this is a *Bottom-Up Approach*.

- *Division based Hierarchical Clustering*: All points are considered to be one big cluster. This big cluster is repeatedly split to obtain smaller clusters. In other words, this is a *Top-Down Approach*.

During each step of the division based Hierarchical Clustering we have to divide a cluster into two smaller clusters. If the cluster consists of N objects, it can be divided into two in $2^{N-1} - 1$ ways and we have to choose the best way to split the cluster among all these ways. The only way to find the best possible way to split the cluster is by comparing all possibilities which leads to exponential time complexity $O(2^N)$ and so we have to use some approximation or heuristic to find the best split. In contrast, for Hierarchical Agglomerative Clustering (referred to as HAC in the rest of the thesis), we have to merge two clusters out of N clusters and this can be done in NC_2 ways ($\frac{N(N-1)}{2}$ ways). So finding the best pair of clusters to merge has a time complexity of $O(N^2)$ which is much better than the time complexity of division based Hierarchical Clustering. So of these two kinds of algorithms, the HAC algorithms are used more often.

1.1 Thesis Motivations and Contributions

In the previous Section, we explained that the time complexity to find the best pair of clusters to merge in the HAC is $O(N^2)$. This merging operation has to be repeated for N-1 iterations until there is only one cluster left and so the total time complexity of the whole algorithm is $O(N^3)$. This algorithm can be sped up by storing the distance between all the pairs of clusters in memory. This has a space complexity of $O(N^2)$. So even though the HAC algorithm has the advantages mentioned in the previous chapter, it is very expensive to be applied to big datasets due to space and time requirements. There have been a number of algorithms proposed to make use of approximations which reduce the space and time complexity so that HAC can be applied to big datasets. To scale the traditional algorithm without any approximations many processors should be used in parallel to do the clustering or new innovative data structures that can reduce the space and time complexity have to be used. We focus on parallelizing the HAC algorithm in the first part of this thesis, and then we focus on the use of innovative data structures to speed up the traditional non-parallel HAC algorithm in the second part of this thesis.

The general trend in hardware and processor development is that the number of cores in a system is increasing. In such a case, it stands to reason that parallelizing the HAC algorithm might make it viable to run it on large datasets in the future. One of the popular hardware architectures for parallel machines is called the Non-Uniform Memory Access architecture or NUMA architecture (explained in detail in Section 4.2, page 27). This is a highly scalable architecture that has been used recently in many of the commercial systems that are produced by companies like AMD and Intel. So we have chosen to parallelize the HAC algorithm and to optimize it for the NUMA architecture. In the first part of this thesis we present the parallel form of the traditional HAC algorithm, and then we explain the unique aspects of the NUMA architecture, analyzing how it can be exploited to speed up the parallel version of the traditional HAC algorithm. We then present a modified parallel version of the traditional HAC algorithm which is NUMA-aware and exploits the parallelism

provided by the NUMA architecture.

In the second part of the thesis, we use a relatively new data structure known as the Cover Tree [1] to speed up the traditional non-parallel HAC algorithm. This version of the HAC algorithm is based on the concept of nearest neighbors and is a sequential algorithm. We have optimized this algorithm to run in a single NUMA region when ever there is enough space, thereby reducing cross-region access. By using the Cover Tree datastructure, we show considerable speed ups for the HAC algorithm on many datasets while having a space complexity of only $O(N)$. We also evaluate how the Cover Tree based HAC behaves when different kinds of data of various dimensionality and structure is used as the input for clustering. Finally we have implemented a parallel version of this algorithm, where we have multiple Cover Trees - one for each thread and we analyze the performance and scalability of this algorithm.

Chapter 2

Related Work

As mentioned in the previous chapter, the main focus of this thesis is on HAC. There have been many algorithms proposed to do the traditional form of HAC without any approximations. In the single link method, the distance between two clusters is calculated as the smallest distance between any two points in the two clusters and in the complete link method, the distance between two clusters is calculated as the maximum distance between any two points in the two clusters. The different linkage criteria and distance metrics are explained in detail in Section 3.2 (page 20). Sneath has described Single link clustering in his paper *The Application of Computers to Taxonomy* [28]. The naive single link clustering method requires $O(N^3)$ time and $O(N^2)$ space for computing the clusters. Sibson [27] proposed an optimal version of single link clustering algorithm. This method reduces the time requirement to $O(N^2)$. A hierarchical clustering algorithm based on the complete link method was described by Sorensen [29]. These are the two most common HAC methods that have been used. Both the single link and the complete link methods can be generalized using the recurrence formula proposed by Lance and Williams [19]

$$d(C_l,(C_i,C_j)) = \alpha_i d(C_l,C_i) + \alpha_j d(C_l,C_j) + \beta d(C_i,C_j) + \gamma |d(C_l,C_i) - d(C_l,C_j)|$$

where d($\cdot$, $\cdot$) is the distance function and α_i, α_j, β and γ are co-efficients that take the value depending on the scheme used. tation The formula describes the distance between a cluster C_l and a new cluster formed by the

merger of two clusters C_i and C_j. By adjusting the values of the coefficients we can arrive at the formula for both single link and complete link clustering.

Co-efficient	**Value**
α_i	$\frac{1}{2}$
α_j	$\frac{1}{2}$
β	0
γ	$-\frac{1}{2}$

Table 2.1: Single-Link Co-efficient Values

Co-efficient	**Value**
α_i	$\frac{1}{2}$
α_j	$\frac{1}{2}$
β	0
γ	$\frac{1}{2}$

Table 2.2: Complete-Link Co-efficient Values

The different HAC algorithms based on different linkage criteria like single linkage, complete linkage and centroid linkage can be constructed by selecting appropriate co-efficients in the formula. These linkage criteria are explained in Section 3.2 (page 20). A detailed table of co-efficient values for different algorithms was presented by Jain and Duin [13].

The computational cost of most HAC Algorithms is at least $O(N^2)$ [31]. This limits their use in practical application for mid-sized and large datasets. In recent years, due to the explosion in the size of datasets and increased processing power, new variations of the HAC Algorithms have been implemented. These algorithms use approximations like summarization or random sampling to make the HAC algorithm more scalable. The difference between these

methods and our proposed methods is that, our algorithm does not use any approximations while increasing the scalability of the traditional algorithm.

BIRCH [32] is an algorithm designed to deal with large datasets and also to have good robustness to outliers. A new data structure named the Clustering Feature (CF) tree is proposed in BIRCH to achieve this. The CF tree creates the summaries of the original data which are called the *Clustering Feature* and stores them. The clustering feature of cluster C_i is defined as the triplet:

$$CF_i = (N_i, LS, SS)$$

N_i is the number of the points in the cluster i
LS is the linear sum of the vectors of the points and
SS is the squared sum of the vectors of the points.

The CF-tree is a height-balanced tree with two parameters,

- Branching factor (B for non leaf node and L for leaf node).
- Threshold T.

Each non-leaf node contains at most B entries of the form

$$[CF_i, child_i], i = 1, ..., B$$

where $child_i$ is a pointer to its i-th child node and the CF_i is the CF entry of the sub cluster represented by this child.

A leaf node contains at most L CF entries. These leaf nodes are chained to other leaf nodes by using two pointers for each leaf node : $next$ and $prev$.

The CF tree captures the important clustering information of the original data while reducing the required storage. The algorithm involves two phases,

- Construct the CF-tree using the two parameters B and T.
- Cluster the set of summaries in the CF-tree using a well know algorithm like the HAC Algorithm.

An additional step can be performed to refine these clusters. During the construction of the CF-tree, outliers are eliminated from the summaries by identifying the points sparsely distributed in the feature space. BIRCH can achieve a computational complexity of $O(N)$ without the final step (clustering the set of summaries using a well know algorithm).

Guha, Rastogi and Shim developed a HAC algorithm named CURE [11] which can detect more complex shapes. The main feature of CURE is that it uses a set of scattered points as a representative of each cluster in addition to the centroid. The representative scattered points are shrunk further towards the centroid of the cluster. These points are then used for clustering. This makes it possible to detect clusters of arbitrary shapes. It also helps to prevent the chaining effect that is common in the Single-Link cluster method. They also use sampling and partition to reduce the computational complexity of CURE and make it more scalable.

Another HAC algorithm named ROCK [12] was proposed by Guha, Rastogi and Shim. They use a new measure named *link* to describe the relation between a pair of objects and their common neighbors. A random sample strategy is used here too to handle large datasets.

The traditional HAC algorithms are usually sensitive to outliers and so lack robustness [31]. There have been some methods that were proposed to improve the accuracy of the traditional HAC algorithm. These algorithms do not usually concentrate on scalability, instead focussing on the quality of the clustering result.

Relative hierarchical clustering (RHC) is an approach which uses the following ratio to decide the distances between clusters:

$$Distance = \frac{Distance\ between\ a\ pair\ of\ clusters\ to\ merge}{Sum\ of\ the\ distance\ from\ those\ two\ clusters\ to\ the\ rest}$$

Mollineda and Vidal [21] describe one such approach and compared the Relative Hierarchical Clustering algorithm to the traditional HAC algorithm on three datasets - an artificial dataset, the Iris dataset and the Vehicle Silhouette dataset [22]. They conclude that the Relative Hierarchical Clustering outperforms the traditional HAC algorithm on these three datasets.

Karypsis et al. [16] propose a new HAC Algorithm called CHAMELEON. According to the authors, this algorithm can better detect clusters when the dataset contains clusters of diverse shapes, density and sizes. This algorithm determines the pair of the most similar sub-clusters by taking into account the *Relative Inter-connectivity* as well as the *Relative Closeness* of the clusters. The *Absolute Inter-connectivity* between two clusters C_i and C_j is defined as the sum of the weights of the edges that connect the two clusters. The *Internal Inter-connectivity* of a cluster is defined as the sum of the weights of the edges crossing a min-cut bisection that splits the clusters into roughly two equal parts. *Relative Inter-connectivity* between two clusters is their Absolute Inter-connectivity normalized to their Internal Inter-connectivity. The *Absolute Closeness* between two clusters is the average weight of the edges that connect cluster C_i to cluster C_j (Inter-connectivity has the sum of the weights instead of the average). The *Internal Closeness* of a cluster is defined as the average weight of the edges that cross a min-cut bisection that splits the clusters into roughly two equal parts. *Relate Closeness* between two clusters is their Absolute Closeness normalized to their Internal Closeness. The paper claims that by focusing on both the Relative Inter-connectivity and the Relative Closeness between the two clusters, CHAMELEON can overcome the limitations of existing algorithms that use static inter-connectivity models.

Li and Biswas [20] extended HAC to deal with both numeric and nominal data. The proposed algorithm, called Similarity-Based Agglomerative Clustering (SBAC), gives a greater weight to the uncommon feature matches and makes no assumptions on the underlying distributions of the data when calculating the similarity measure between clusters.

2.1 Parallelizing the HAC Algorithm

There has been some work on parallelizing HAC algorithms. Rasmussen and Willet [26] have implemented parallel algorithms for the single linkage and Ward's HAC methods on the ICL Distributed Array Processor, which was one of the earliest commercial massively parallel computers. These computers ex-

hibit data-level parallelism, i.e. every processor executes the same instruction on different datasets. Rasmussen and Willet optimized those HAC methods for this parallel computer. Driscoll et al. [6] have described a new data structure for the parallel computation of minimum spanning trees. This data structure is a parallel implementation of the Djikstra's minimum spanning tree algorithm [5] in $O(nlogn)$ time using $\frac{e}{vlogv}$ processors, where v is the number of vertices in the graph and e is the number edges. The minimum spanning tree can be used to perform the single link HAC algorithm. Olson [24] describes parallel algorithms using Nearest Neighbors. This algorithm maintains an array of the Nearest Neighbors to perform the HAC. The naive Nearest Neighbor algorithm that we state in chapter 6.2 is similar to the algorithm explained by Olson. We then proceed to improve on this algorithm using special data structures.

There has also been some work on parallelizing HAC using the MapReduce framework [4]. MapReduce is a framework for parallelizing algorithms over a large number of computers. These computer groups are called clusters. A big advantage of MapReduce is that, the computers that form the clusters can be low end machines. MapReduce leverages the collective power of these low end machines to acheive the parallelism at a low initial cost. To implement HAC over MapReduce, we need to break the algorithm into one or more Map tasks and a Reduce tasks. During every iteration of the algorithm, we have to find the closest pair of clusters from all the cluster pairs. This task can be parallelized and made into a number of Map tasks. Each Map task takes as input a subset of the cluster pairs and emits out the closest cluster pair among them along with the corresponding distance. The next step is to find the global minimum of these Map outputs. This can be done by a Reduce task. This method was used by Gao et al. [10]. One disadvantage of using MapReduce for HAC is that MapReduce frameworks do not store the state between successive iterations. That is, if a MapReduce node finds the distance between a number of cluster-pairs, there is no way of storing the calculated distances in the same MapReduce node. So, we need to re-send this data at the beginning of each iteration. This communication is very expensive and so

the traditional MapReduce is not ideal for the HAC.

There have been a few modifications to the MapReduce framework for adapting it to iterative algorithms. Twister [7] is one such implementation of MapReduce for iterative algorithms. Twister allows the user to specify static data that remains constant during each iteration and dynamic data that change during each iteration. This avoids the problem encountered by traditional MapReduce when implementing HAC. There are other iterative MapReduce framework implementations like Incoop [2] and iMapReduce [33]. There is no HAC algorithms that have been developed over Twister or any other iterative MapReduce platforms yet.

Chapter 3

Hierarchical Clustering Algorithm - Theory

In this chapter, we discuss the HAC algorithm. We first outline the algorithm and then explain the different ways in which the dissimilarity between the clusters can be found. Then we conclude by explaining the two broad categories of the HAC algorithm based on the amount of space used.

3.1 The Basic HAC Algorithm

The aim of the Hierarchical Clustering Algorithm is to build a hierarchy of clusters. Let the dataset consist of N data points with dimensionality D. Algorithm 1 gives an outline of the basic HAC Algorithm.

Algorithm 1: HAC

Input: Data(N, D)

```
for i = 1 to N − 1 do
    Find closest pair of clusters
    Merge the closest pair of clusters
    Output the pair that has been merged.
end
```

3.2 Cluster Dissimilarity Measures

Clusters are comprised of a number of points. In a multi-dimensional space, each point is usually represented by a vector of values. To decide which clusters should be merged or split, a measure of dissimilarity between clusters is needed. This measure of dissimilarity is usually obtained by using a combination of two factors:

- A Distance Metric - A measure of distance between a pair of points which are stored as vectors.
- A Linkage Criteria - This gives the dissimilarity of clusters as a function of the pair-wise distances between the points belonging to the two clusters.

3.2.1 Distance Metric

Distance metrics are used to find the distances between two points which are represented by vectors. These are a few of the distance metrics that can be used:

- Euclidean distance: If there are two points that are represented by vectors $p = (p_1, p_2, ..., p_n)$ and $q = (q_1, q_2, ..., q_n)$ then the Euclidean distance between these two points are given by

$$d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + ... + (q_n - p_n)^2}$$

- Manhattan Distance: The Manhattan distance between two points represented by vectors $p = (p_1, p_2, ..., p_n)$ and $q = (q_1, q_2, ..., q_n)$ is given by

$$d(p, q) = \sum_i |p_i - q_i|$$

- Cosine Distance: The Cosine distance between two points represented by vectors $p = (p_1, p_2, ..., p_n)$ and $q = (q_1, q_2, ..., q_n)$ is given by

$$\frac{p \cdot q}{\|p\|\|q\|}$$

3.2.2 Linkage Criteria

Section 3.2.1 (page 20) gives the metrics that can be used to find the distance between two points. But every cluster is made up of of one or more points. Consider two clusters, cluster C_a with n points and cluster C_b with m points. If we need to find the distance between these two clusters, we need to decide how to use the points in each of the clusters to calculate the distance. This is done using the Linkage criteria. To find the distance between the clusters, a particular Linkage criteria is selected and used in conjunction with a distance metric. Here are some of the Linkage criteria that are commonly used:

- **Single Linkage**: Here the distance between two clusters is calculated as the distance between the two closest points in the two clusters X, Y. This can be described by

$$d(X, Y) = \min_{x \in X, y \in Y} d(x, y)$$

  This distance calculation is dependent on the number of points present in the two clusters and so is not of constant time complexity.

- **Complete Linkage** : Also known as the farthest neighbor method. Here the distance between two clusters is computed as the maximum distance between any two points in the two clusters.

$$d(X, Y) = \max_{x \in X, y \in Y} d(x, y)$$

  Here too the distance calculation is dependent on the number of points present in the two clusters and so is not of constant time complexity.

- **Unweighted Pair Group Method with Arithmetic Mean (UPGMA)** : The distance between two clusters is the average of the distances between all the points in the two clusters.

$$d(X,Y) = \frac{1}{|X| \cdot |Y|} \sum_{x \in X} \sum_{y \in Y} d(x \cdot y)$$

This distance calculation is dependent on the number of points present in the two clusters and so is not of constant time complexity.

- **Centroid Linkage** : In centroid based clustering, each cluster is represented by a single vector - the centroid of all the points in the cluster. The centroid of a cluster X containing points $x_1, x_2, ..., x_n$ is given by

$$C_x = \frac{1}{n} \sum_{i=0}^{i \leq n} x_i$$

To find the distance between two clusters using the centroid linkage, we find the distance between the centroids of the two clusters. The time complexity of this operation depends only on the dimensionality D of the dataset and not on the number of points in each cluster. So if the dimensionality of the dataset D is a constant, finding the distance using the centroid linkage is of constant time complexity.

3.3 Stored Matrix and Stored Data Algorithms

In Algorithm 1, during each iteration we need to find the closest pair of clusters. One way of finding the closest pair of clusters is to find the distance between all pairs of clusters and take the pair with the minimum distance. To avoid re-computations of these distances in each iteration of the algorithm, we store the computed distances in a matrix called the *Dissimilarity Matrix*. At each iteration of the algorithm, when the clusters are merged, the dissimilarity matrix is updated to reflect the merging of the clusters. The dissimilarity matrix requires $O(N^2)$ space. The HAC algorithm can be also written without storing the whole dissimilarity matrix. Anderberg [23] divides Hierarchical Clustering

Algorithms into the following types, based on whether a dissimilarity matrix is used or not.

- Stored Matrix Algorithms.
- Stored Data Algorithms.

3.3.1 Stored Matrix Algorithms

In the Stored Matrix algorithms, the entire dissimilarity matrix is maintained in memory. A naive way to implement a dissimilarity matrix is a two dimensional array. During each iteration, the dissimilarity matrix is searched to find the closest pair of clusters. The closest pair is merged and then the dissimilarity matrix is updated by deleting the entries for the merged clusters and adding a new entry for the newly created cluster. Using a naive brute force method, finding the closest pair of clusters has a time complexity of $O(N^2)$. Because there are N iterations, the final time complexity for this is $O(N^3)$.

Alternatively, we can use a priority queue to implement the dissimilarity matrix. The distances between the cluster-pairs can be used as the priority for the priority queue - smaller distances have a higher priority. To find the closest pair of clusters from this priority queue, we have to find the element with the highest priority. The time complexity to find the element with the highest priority in a priority queue is $O(1)$. Insertions and deletions in a priority queue have time complexity $O(logN)$. So, if we use a priority queue, the time complexity of the algorithm is $O(N^2 logN)$. Even when we use the priority queues, the space requirement is $O(N^2)$.

3.3.2 Stored Data Algorithms

In this method, we do not store the whole dissimilarity matrix. Instead, we only store a part of the dissimilarity matrix and calculate the other distances as needed. One way of implementing this is to maintain a Nearest Neighbor array which stores the Nearest Neighbor of each cluster. We refer to this method as the *Nearest Neighbor based HAC*. This method is efficient in terms of time and space if finding the distance between clusters is of constant time complexity. As

mentioned in Section 3.2.2 (page 21), calculating the distance between clusters is of constant time complexity when the centroid linkage is used and it is not of constant time complexity when other linkage criteria like single linkage, complete linkage and average linkage are used. So when we use the centroid linkage, the naive Nearest Neighbor search has $O(N)$ space complexity and $O(N^2)$ time complexity to compute the Nearest Neighbor array. Finding the closest pair of clusters requires scanning the Nearest Neighbor array to find the cluster with the closest Nearest Neighbor. These two clusters are merged to create a new cluster. After the merge, we have to update the Nearest Neighbor of any cluster that had one of the merged clusters as the Nearest Neighbor. In each iteration, let α be the maximum number of clusters that had one of the merged clusters as the Nearest Neighbors. Then the time complexity to update their Nearest Neighbors is $O(\alpha N)$. Thus the total complexity of this algorithm becomes $O(\alpha N^2)$ when the centroid linkage is used. For other linkage criteria, computing the distance between clusters is not constant time. So in those cases, we also need to maintain a dissimilarity matrix in addition to the Nearest Neighbor array to avoid costly re-computations (a stored matrix method).

Chapter 4

Multiprocessor Architectures

In this chapter we first outline the different kinds of mulitprocessor architectures. Then we elaborate on the Multiple Instruction Stream, Multiple Data Stream (MIMD) processors and outline the two broad categories of these processors - Uniform Memory Access architectures and Non-Uniform Memory Access architectures. In the rest of the chapter, we explain the unique attributes of these two kinds of MIMD processors and discuss their advantages and disadvantages.

A multiprocessor is a tightly coupled computer system which contains two or more processing units that share the tasks. Flynn [8] has categorized computer architectures into the following types:

- Single Instruction Stream, Single Data Stream (SISD) Processor - This is a sequential computer with no parallelism.
- Single Instruction Stream, Multiple Data Stream (SIMD) Processors - The same instruction is executed by multiple processors using different data streams. These are used in the multimedia fields like image processing and audio.
- Multiple Instruction Stream, Single Data Stream (MISD) Processors - Each processor executes different instructions on the same data stream. This is used when there is a need for fault tolerance.
- Multiple Instruction Stream, Multiple Data Stream (MIMD) Processors - Each processor executes different instructions on different data

streams. The MIMD processors are the most popular of all the above multiprocessor solutions because of their flexibility and cost-performance advantages. Most of the distributed systems that exist today fall under this category.

MIMD processors can be classified into two categories:

- Uniform Memory Access Architectures (UMA).
- Non-Uniform Memory Access Architectures (NUMA).

4.1 Uniform Memory Access Architectures (UMA)

In the Uniform Memory Access architecture (UMA) all the processors share the entire physical memory uniformly. Figure 4.1 gives an example of the uniform memory access architecture. The CPUs are connected to the memory using a memory bus. Since the memory is shared, the access times to any part of the memory is the same for all the processors and there is some built-in mechanism to synchronize memory access from different processors.

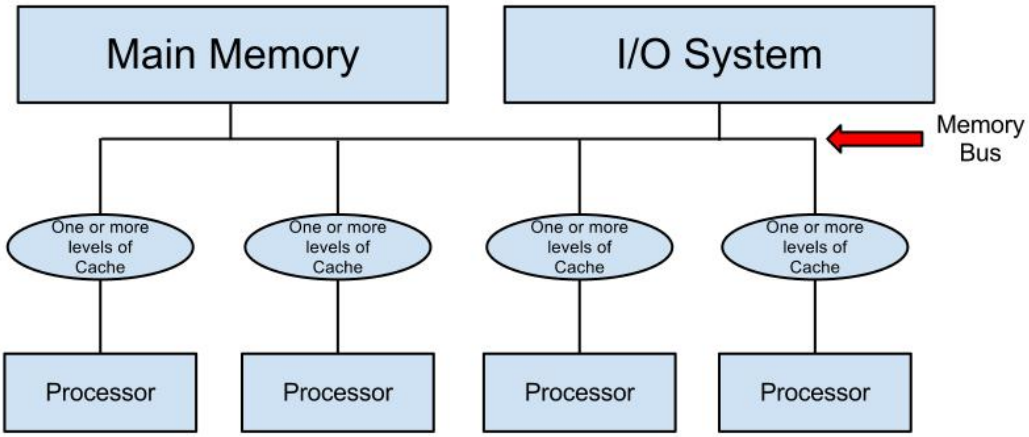


Figure 4.1: A High Level diagram of the Uniform Memory Access architecture.

The Uniform Memory Access architectures were among the earliest type of the multiprocessor architectures and are simple. But the CPU in most systems is faster than the memory. So when multiple processors use a single shared memory and execute rapid memory accesses, most of the processors are starved

(they do not have any data and so are not able to do any data-processing tasks) and they are idle during that time. Caches were added to each processor to reduce the starvation of the processors. A cache is a high speed memory which is individual to each processor. As long as the data is in the cache, the processor is not data starved. If the data is not present in the cache, the processor has to access the shared memory. So sophisticated algorithms are needed to reduce cache-misses by using data-locality. But the massive increase in the amount of data processed and the performance enhancements of recent processors have made these workarounds less efficient. Also if the number of processors in the multiprocessor architecture is increased, the memory access starvation time of each processor increases.

4.2 Non-Uniform Memory Access Architectures (NUMA)

Non-Uniform Memory Access architectures overcome the limitations in the Uniform Memory Access architectures. In this architecture, each processor has its own memory. This leads to improved performance as the processors do not contend for memory as long as the data is present in the processor's own memory. But, it is not always the case that the data can be confined to a single processor's memory. In such a case, a processor might need to access data from another processor's memory. To facilitate this, NUMA architectures usually have additional mechanisms for allowing processors to access the memory of another processor. This cross-region memory access creates memory contention between the two processors and also carries the overhead of transferring the data through a memory bus to the requesting processor. So in a NUMA architecture, memory access time is not uniform - memory access to cross-region memory takes more time when compared to accessing local memory. Figure 4.2 gives a high level representation of the Non-Uniform Memory Access architecture.

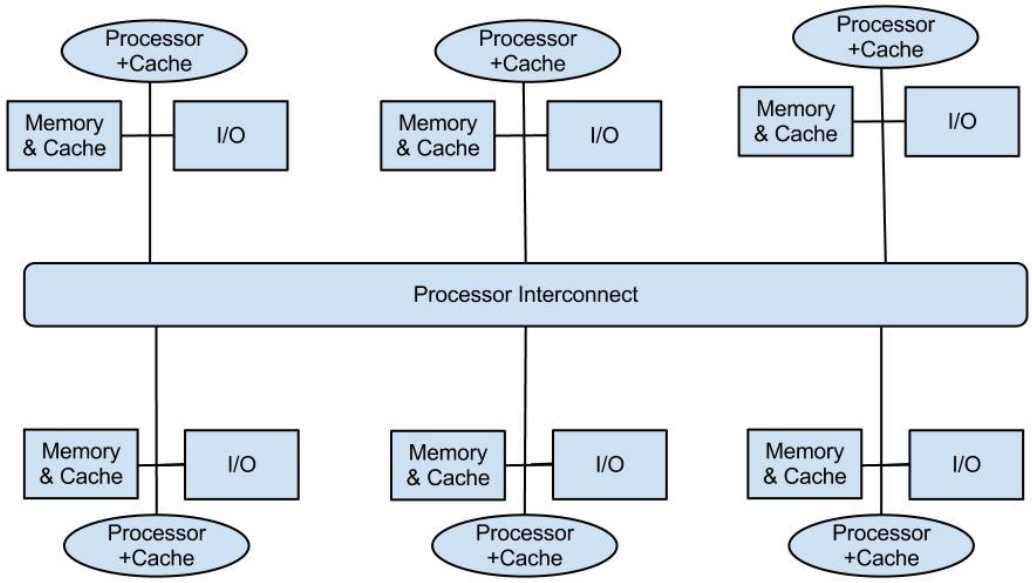


Figure 4.2: A High Level diagram of the NUMA architecture.

4.2.1 Cache Coherent NUMA

NUMA architectures also have caches for each processor to further exploit locality of reference. As there is a cache for every processor, the data present in the caches should be consistent. This is called *cache coherency*. All the NUMA computers in the market today are cache coherent. Each processor has a cache-controller and these cache-controllers interact with each other to maintain cache coherency. If multiple different processors access the same memory in rapid succession, then the overload for maintaining cache-coherency becomes increasingly expensive. Thus cross-region memory access and rapid successive access of the same memory location by different processors are very expensive in a NUMA architecture. If the hardware and Operating System support NUMA, then it tries to reduce the frequency of these two expensive operations by allocating memory in NUMA-friendly ways. This hides the overhead of NUMA from most of the user written programs. But for data-intensive applications, a considerable overhead is usually observed if they are not written in a NUMA-friendly way. So an improvement in performance can be achieved if the data intensive applications are written to be NUMA-aware

and to reduce these expensive operations.

4.2.2 NUMA Regions or NUMA Nodes

Allocating a separate memory for each processor is expensive. A reasonably small number of processors can share memory without encountering the limitations of the Uniform Memory Access architectures. So, in the NUMA based computers, the processors are divided into blocks called *NUMA regions* or *NUMA nodes*. Each region can house many processors and has its own local memory. Most of the NUMA-based architectures in the market today house either four or eight processors in every NUMA region. In a single processor CPU, the processor is connected to the memory controller by using an interface called the Front-side bus. In the NUMA architectures, a similar interface is used to connect each NUMA region to another. These interfaces are referred to as *Processor Interconnects*. The two main processor interconnects in the market are the HyperTransport from AMD and the QuickPath interconnect from Intel. Figure 4.3 gives a high level diagram of a NUMA architecture with four NUMA regions, each comprising of four processors. It can be seen that each NUMA region has its own memory and the NUMA regions are connected.

The processor interconnects need not form a fully connected mesh that connects each NUMA region to every other region. Instead they can use any topology and so the NUMA regions can be two or more hops apart from each other. In such cases, cross-region memory access to a neighboring NUMA region is faster than cross-region access to a NUMA region which is two or more hops apart. An advantage of this kind of architecture is scalability. A large number of processors can be added to this architecture, without affecting the throughput. In essence, the NUMA architecture has the following advantages:

- A small number of processors sharing a memory and a cache for maxi-

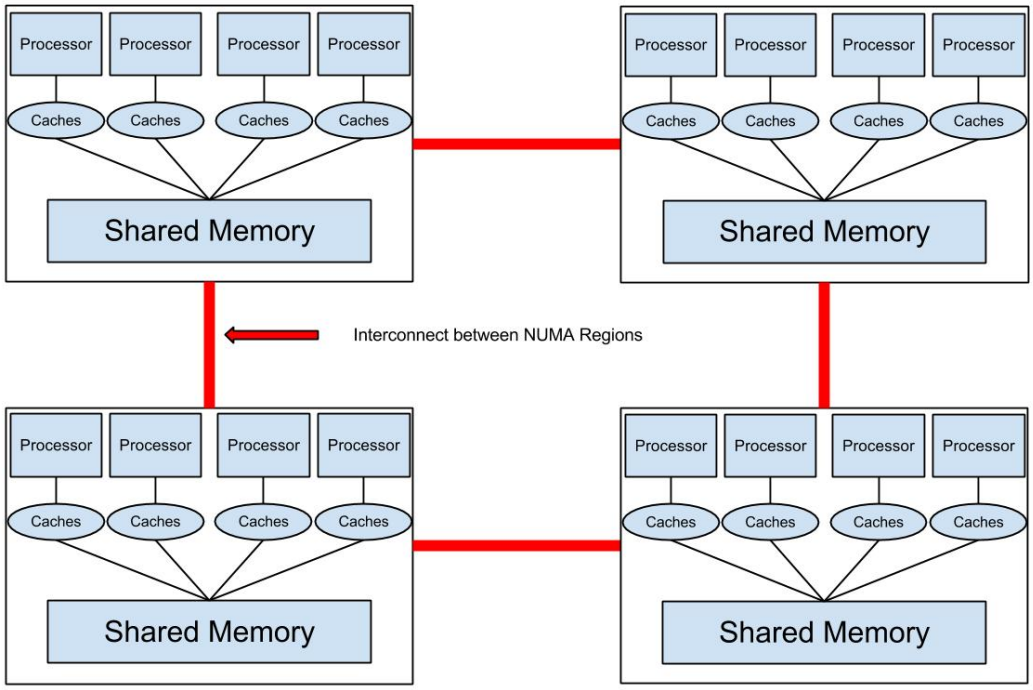


Figure 4.3: A NUMA architecture with four NUMA regions, each comprising of four processors

mum utilization.

- The processors are compartmentalized into NUMA regions to minimize the effects of memory contention and starvation.

Even though the NUMA architecture has the above advantages, cross-region memory access is still expensive compared to local memory access. So, the applications written for NUMA architectures can be made much more efficient by reducing cross-region memory accesses and ensuring that most of the cross-region memory accesses are done on NUMA regions which are only one hop apart (we refer to them as neighboring NUMA regions).

4.2.3 Architecture of machine that we used

The NUMA machine that we used for our experiments had four 8-core AMD 6100 series Opteron Processors making a total of 32 cores. Figure 4.4 gives the Processor and I/O Chipset block diagram for our system. From the figure it can be seen that there are four G34 Sockets numbered from one to four. There sockets house the four AMD Opteron processors and the Hyperlink inter-connects between these sockets can be seen from the figure. In this machine, every NUMA-node was connected to every other NUMA-node by one hop forming a fully connected mesh.

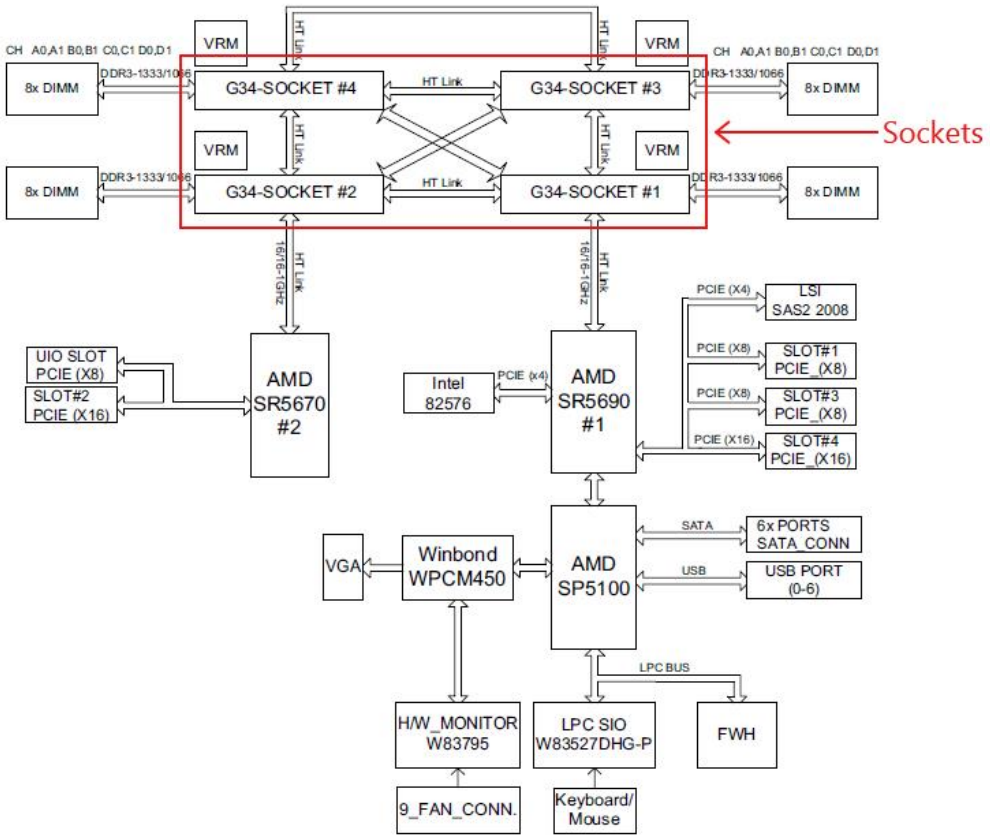


Figure 4.4: Processor and I/O chipset of our system. Taken from [38]

4.3 Memory Allocation Policy in a NUMA architecture

In a system with a NUMA architecture, there are multiple regions each with its own memory. If there is a request for memory allocation, there needs to be some way of deciding the NUMA region from which this memory is allocated. Most operating systems allocate only the virtual memory at the time of the request. This virtual memory is not mapped to any physical memory at that time. When a thread first accesses the allocated memory, the operating system maps the virtual memory to physical memory from the same NUMA region on which that thread runs on. After this thread uses up this memory, it is then returned to the operating system. When another thread of the same application running on a different NUMA region requests memory, the operating system might reuse the memory that was earlier allocated to the first thread. So, the later thread has a high probability of being allocated memory that was released by a different thread on another NUMA region . This causes considerable overhead due to cross-region accesses. So if the program is not NUMA-aware, the above reason might result in reduced efficiency.

Chapter 5

Parallel Hierarchical Clustering Algorithm

In this chapter, we first describe the process of parallelizing the HAC algorithm and outline one such parallel algorithm. Then we discuss how there are possibilities to optimize this parallel algorithm when the system is based on a NUMA architecture. We then present a parallel HAC algorithm which is NUMA-aware and explain the various optimizations and implementation details of this algorithm. Finally we conclude the chapter by comparing the performance of the parallel HAC algorithm to the NUMA-aware parallel HAC algorithm.

5.1 What can be parallelized?

The HAC Algorithm has the following main steps:

- Finding the distances between all pairs of clusters.
- Finding the cluster pair that is the closest for every iteration.

- Merging the cluster pair that is the closest for every iteration.

Of these steps, the first lends itself well to parallelization. For N clusters, a total of $(N^2/2)$ distances have to be found. If there are a total of P processors, then each processor can be assigned $(N^2/2P)$ pairs of clusters to process. Each processor finds the closest cluster pair among the $(N^2/2P)$ cluster pairs along with their distance. Then the globally closest cluster pair can be found and they can be merged. This can be done in a multi-threaded model using a master thread and worker threads. A master thread starts various worker threads. The worker threads process the cluster pairs and find the locally closest cluster pair. The master thread uses this to find the globally closest cluster pair and merges those clusters. Algorithm 2 explains how the master thread works and Algorithm 3 explains how the worker threads work.

Algorithm 2: A Parallel HAC - The Master Thread

Input: A set of N points to be clustered. Initially each point is its own cluster. Let the number of processors be P.

1 Allocate $(N^2/2P)$ cluster pairs to each thread.
2 Allocate memory for $DissimilarityMatrix$.
3 Let $TotalNumberOfClusters$ = N
4 **while** $TotalNumberOfClusters > 1$ **do**
5 Start worker threads and wait for their completion.
6 Get the closest cluster pair from each worker thread and find the closest clusters (C_x, C_y) of the whole dataset.
7 Merge the closest clusters (C_x, C_y).
8 Broadcast the newly formed cluster information to all the threads.
9 $TotalNumberOfClusters = TotalNumberOfClusters - 1$.
10 **end**

This algorithm requires a master thread for these three tasks:

- Dividing the data and assigning it to the respective processors

Algorithm 3: A Parallel HAC - For Worker Thread W_i

```
1  Wait for signal from Master thread.
2  foreach cluster pair allocated to this thread do
3  |  If the cluster pair distance is not stored in the DisimilarityMatrix
   |  find the distance and store it in the DisimilarityMatrix.
4  end
5  Send the two locally closest clusters (C_{x_i}, C_{y_i}) and the distance between
   them to master thread.
6  Wait for receiving the globally closest cluster pair.
7  Receive globally closest cluster pair (C_x, C_y).
8  if Currrent Worker Thread i has a cluster pair which has either C_x or
   C_y or both then
9  |  Update the DissimilarityMatrix to reflect the merging of that
   |  cluster.
10 end
11 Update the dissimilarity matrix to reflect the new cluster created by the
   merging of the two clusters.
```

- Collecting the local minimum from the different processors to find the global minimum.
- Merging the closest pair of clusters.

5.2 Overloading of the Point-to-Point Transport

All the NUMA regions are connected using the Point-to-Point link (shown in fig 4.3). In the parallel version of HAC presented in Algorithm 2 and Algorithm 3, the memory for the dissimilarity matrix is allocated by the master thread. When the other threads access this memory, the data is transferred through the Point-to-Point Transport. As mentioned in 4.2.1, some NUMA regions can be two or more hops apart from the NUMA region in which the Master thread runs. So when a thread requests cross-region access to the

dissimilarity matrix, the data needs to be transferred through the Point-to-Point Transports of other NUMA regions which act as intermediate carriers. An example is shown in figure 5.1. It can be seen that the Point-to-Point transport between *NUMA region 0* and *NUMA region 1* is overloaded as it is being used for data transfers between other NUMA regions. This whole process can be optimized for NUMA architectures by splitting the dissimilarity matrix among the different NUMA regions and accessing them in a NUMA aware manner.

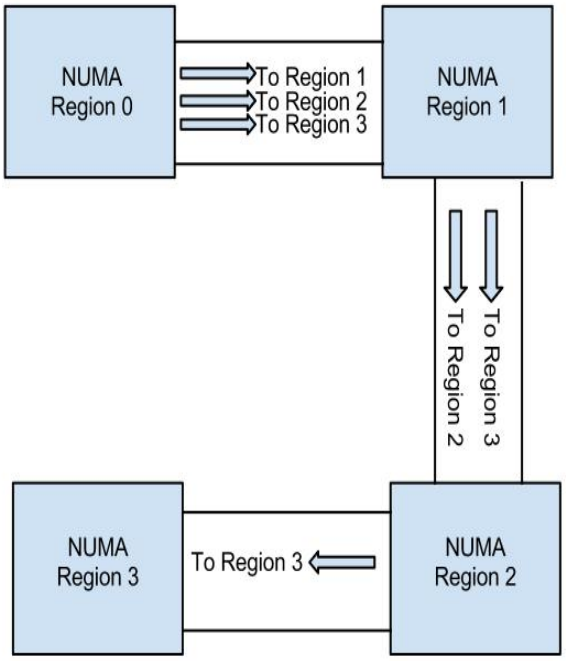


Figure 5.1: This figure shows how the point to point link between *NUMA region 0* to *NUMA region 1* is overloaded because of the data being transferred to the other NUMA regions .

Also, when the size of the data is huge and the number of cores is large, there is an obvious bottleneck in the NUMA region where the Master thread

runs. All the data is spread out to the different processors from this region and all the output is received back in this region. So in addition to splitting the dissimilarity matrix, we need to streamline the flow of data from one NUMA region to another.

5.3 Requirements for making the HAC Algorithm NUMA-aware

We need to ensure that the following criteria are met when designing a HAC algorithm that is NUMA-aware.

- Memory accesses should be as localized as possible.
- Data exchange among the processors should flow in a peer-to-peer fashion and most cross-region accesses should be restricted to nearby NUMA-regions.

5.4 Lesson learnt from Soccer Players

Teubner and Mueller [30] identify the way that soccer players organize themselves for the handshake before a soccer game and exploit this to do stream joins without a central co-coordinator. We have applied the same inspiration to parallelize the part of the HAC algorithm that finds the distance between all the pairs of clusters. Soccer players are aware of how to execute a handshake between all pairs of players from the two teams without any external co-ordination. They do so by walking across each other in opposite directions and shaking hands with players from the opposite teams. The same process can be used to find the distance between all pairs of clusters without the need for external co-ordination.

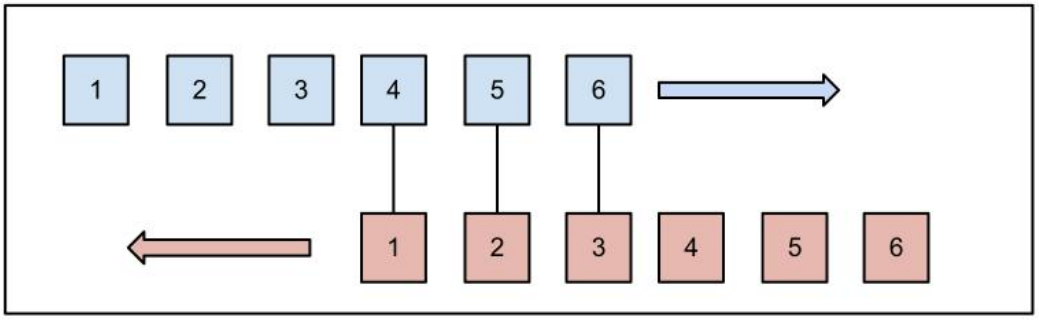


Figure 5.2: An illustration of the data flow using the inspiration from soccer players.

5.5 The Basic Idea of the Algorithm

The HAC algorithm first computes the distance between all cluster pairs in the first iteration. These computed distances are used and updated in the subsequent iterations. So the first iteration takes the most amount of time as it computes all the pair-wise distances between the clusters. The algorithm presented here uses the soccer handshake principle presented by Teubner et al. [30] to make the first iteration NUMA-optimized. Essentially, the first iteration constructs a dissimilarity matrix that is split among the various NUMA regions. All the subsequent iterations use this split dissimilarity matrix in a NUMA-aware manner to produce a NUMA-optimized version of the whole algorithm. We first explain how the first iteration works and then give the details of how the subsequent iterations are NUMA-aware.

5.6 Idea for the First Iteration

We will first explain what happens in each individual processor and then we will explain what happens across the different processors. The input to the algorithm is all the clusters for which the pair-wise distances have to be calculated. We make two copies of this input data which will act as two independent

streams R and S. Now the two streams are moved in opposite directions, i.e. stream R is moved from left to right and stream S is moved from right to left. Figure 5.3 gives a diagrammatic representation of this concept. As the streams move against each other, each processor finds the distance between the pairs of clusters which are aligned opposite to each other. These distances are stored in the local memory of the NUMA region that each processor belongs to. This process is repeated until the steams completely pass each other. Once this is completed, the distance between all the pairs of clusters would have been calculated and stored. This process also splits the dissimilarity matrix between the different NUMA regions and is the basic idea for the first iteration of our algorithm.

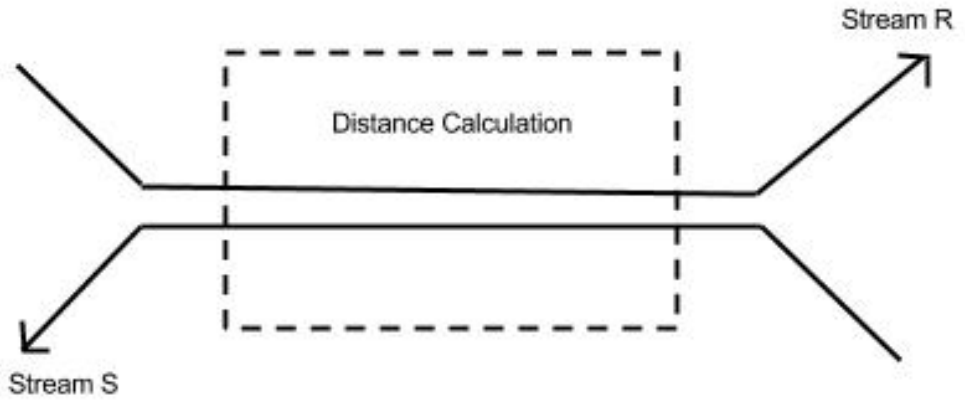


Figure 5.3: *The data is divided into two streams which are lined up against each other and moved in the opposite direction. The distances between the pairs that are aligned opposite to each other are found. Figure taken and redrawn from [30].*

5.6.1 Correctness of the First Iteration

To ensure correctness of the algorithm, the following conditions should be met in each iteration:

- The distance between all pairs should be found.

- The two clusters which have the overall minimum should be identified.

In addition to these two criteria, we also add the following for making sure we don't do any additional work and for ensuring efficiency:

- The distance between a pair of tuples is calculated not more than once.
- The communication between the processors is autonomous and does not depend on a central co-coordinator.

In the following sections, we will see how this algorithm can be parallelized while preserving the criteria mentioned above.

5.7 Parallelization of the First Iteration

5.7.1 Distributing the streams among the different connected processors

To parallelize this algorithm, each processor is equipped with a window for each of the streams R and S. The processors are connected like a chain using message queues. This is shown in Figure 5.4. The two streams are passed through the corresponding windows in each processor. When the streams pass each other, the distances between the clusters in the two windows are found. Whenever a cluster enters the window of a processor, it is stored in the local NUMA region of the processor. Because of this, the distance calculations can always be done locally. Figure 5.5 shows an example for a single processor, with a window for each of the two streams R and S. Any data structure like a hash-table or priority queue can be used to store the computed distance values.

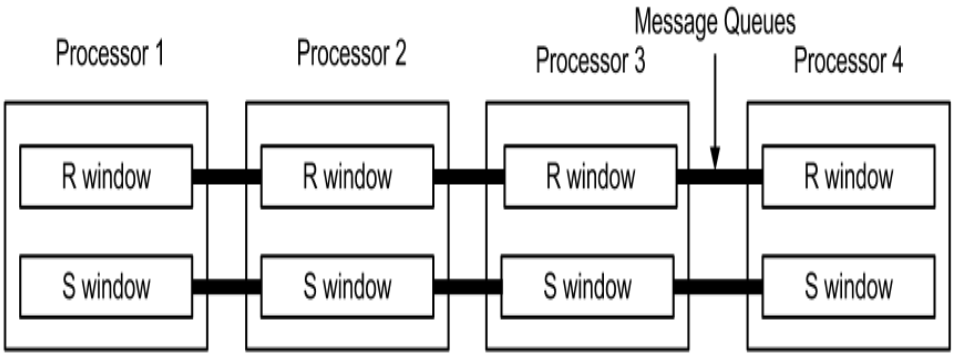


Figure 5.4: Each processor has a window for the two streams. The processors are connected like a chain. Figure taken and redrawn from [30].

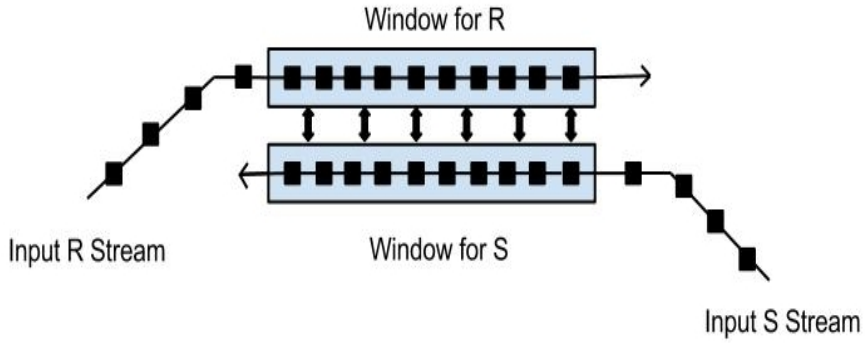


Figure 5.5: Each processor has a window for each of the streams. The streams are moved through the windows. As they move through the windows, the distance between the clusters are calculated. Figure taken and redrawn from [30].

There are two ways to move the streams against each other:

- BiDirectional Handshake Method - The two streams move against each other simultaneously.
- UniDirectional Handshake Method - The two streams are moved one by one.

We will now describe how both these methods are implemented.

5.7.2 BiDirectional Handshake Method

In this version, the two streams are moved towards each other simultaneously. Each processor has its own window for each of the streams of clusters and each window can hold a part of the whole cluster stream. The two streams are simultaneously pushed towards each other through these windows. Figure 5.6 shows an example with four processors each with its own window and the two streams being simultaneously pushed towards each other through the windows.

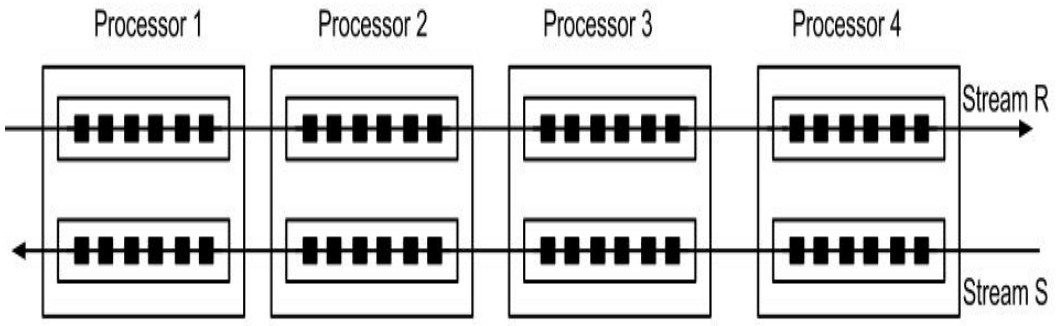


Figure 5.6: Each processor has two windows and the two streams are simultaneously pushed towards each other through the windows. Figure taken and redrawn from [30].

When a new cluster from one of the streams enters a window of a processor, the processor finds the distances between the new cluster and the clusters present in the other window of the same processor. An example of this can be seen in Figure 5.7.

Missed Cluster-pairs

When moving the two streams simultaneously towards each other, two clusters might miss each other because both were in transit. Figure 5.8 explains one example of such a miss. Clusters r_5 and $s4$ from the two opposite streams miss

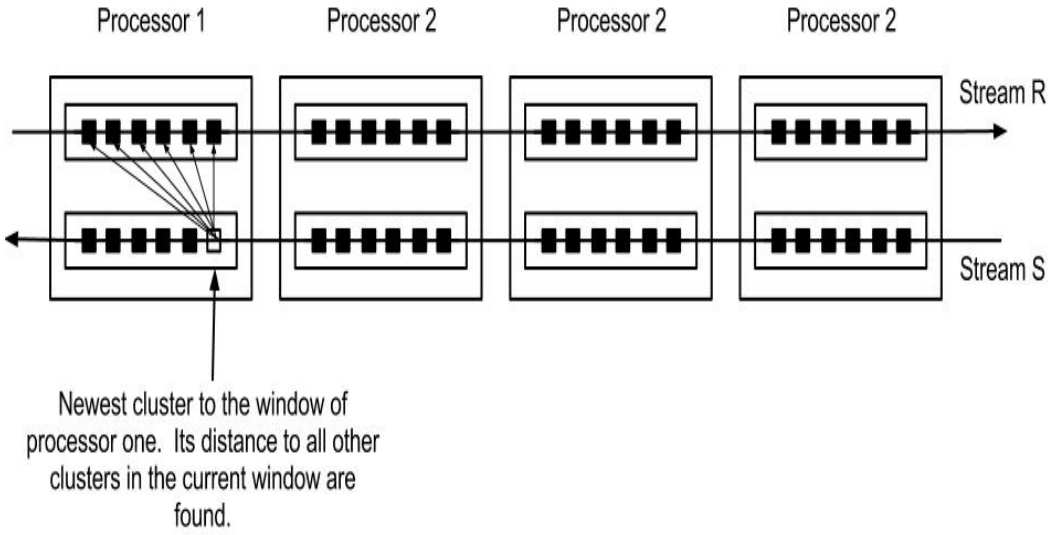


Figure 5.7: The hollow box indicates the new cluster. Its distance is found immediately with the clusters of the other stream present in the current processors window. Figure taken and redrawn from [30].

each other as they both are in the FIFO queues at the same time. So both *processor k* and *processor k+1* do not process the cluster pair *(r_5,s_4)*. This affects the correctness of the algorithm. To avoid this problem, we must make sure that these two clusters meet in one of the processors *k* or *k+1*. This can be done by using acknowledgments and short-term backups of clusters that have been sent to neighboring processors.

The aim is to make the two cluster pairs meet in exactly one of the two processors. Let the processor on the right always process these missed cluster-pairs. In this case it is *processor k+1*. Whenever this processor sends a cluster to its left neighbor, it keeps a copy of this cluster s_i in its local window, but marks it as forwarded. Figure 5.9 illustrates this.

Cluster s_4 is backed up on *processor k+1*. Even though the clusters r_5 and s_4 are in the communication channel at the same time, there is a backed up copy of s_4 in *processor k+1*. So these two clusters meet in *processor k+1*. In this step of the communication process, s_4 was paired with r_6, r_7, r_8... on

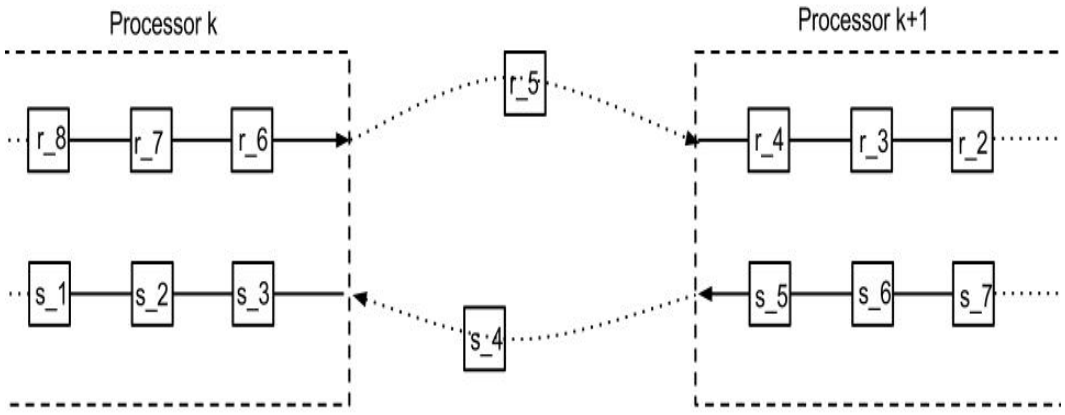


Figure 5.8: An example showing the missed cluster-pair problem. Clusters r_5 and s_4 from the two opposite streams miss each other as they both are in transit at the same time. Figure taken and redrawn from [30].

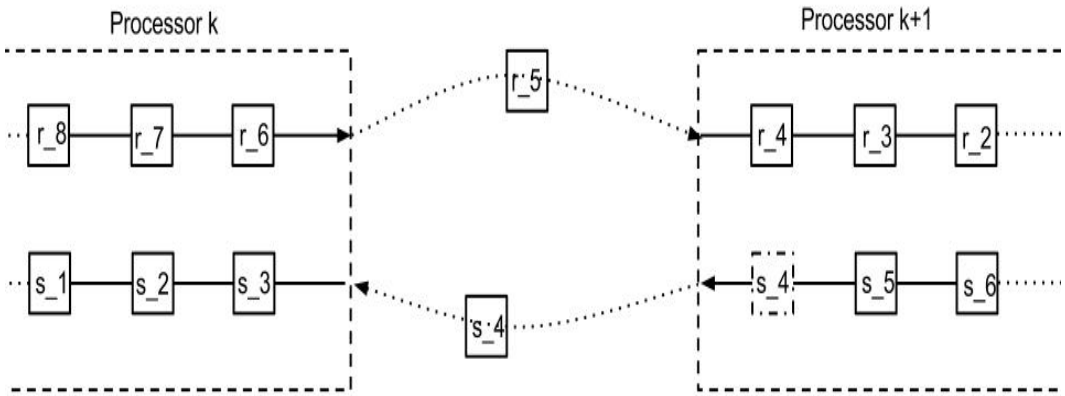


Figure 5.9: Two phase protocol - step one. Cluster s_4 is backed up on *processor k+1*. Figure taken and redrawn from [30].

processor k and r_5 paired with s_4, s_3, s_2... on *processor k+1*. This is illustrated in Figure 5.10

Now we have the second phase, which is the acknowledgement phase. Once *processor k* receives the cluster s_4 it sends an acknowledgement back to *processor k+1* for cluster s_4. This acknowledgement notifies *processor k+1* that any other cluster sent after this acknowledgement will have already seen s_4 in *processor k*. So, once *processor k+1* receives this acknowledgement, it can safely delete cluster s_4 from its window before accepting the next cluster. Fig-

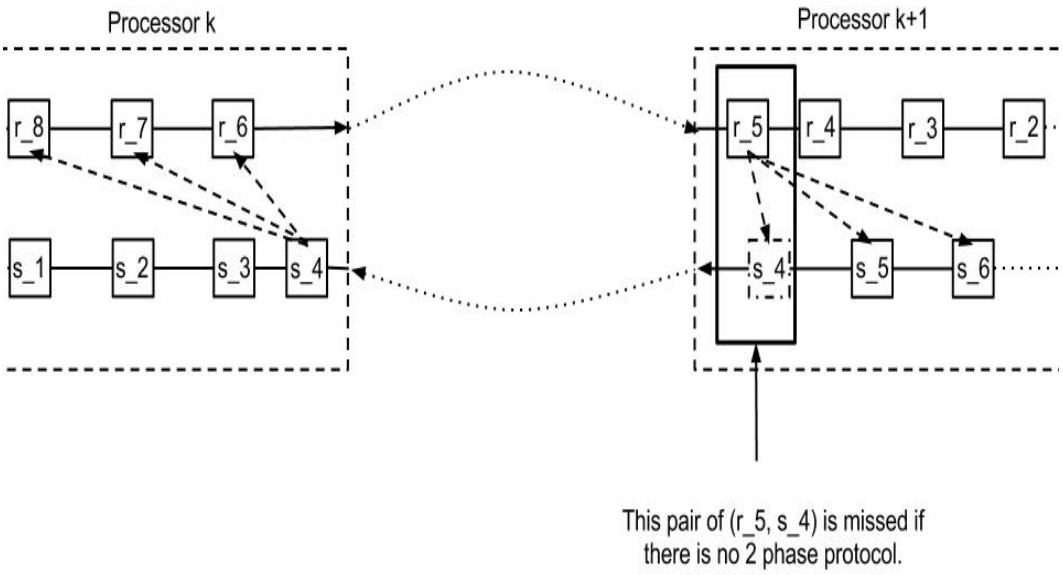


Figure 5.10: s_4 was paired with r_6, r_7, r_8... on *processor k* and r_5 paired with s_4, s_3, s_2... on *processor k+1*. Figure taken and redrawn from [30].

ure 5.11 shows an example of this. This two-phase process ensures that no clusters are missed and also that the distance between every cluster pair is computed only once.

Disadvantage of the Bidirectional Method:

The missing cluster-pair problem requires us to use a two-phase acknowledgement protocol to maintain the correctness of the algorithm. This adds an overhead to the algorithm. Also, in the bidirectional variant, each processor has its own window. Thus if there are 4 NUMA-regions with a total of 32 processors each, then there is considerable overhead as each processor has to go through the two-phase messaging protocol. The experimental results section in this chapter, 5.11, shows results to verify the above fact.

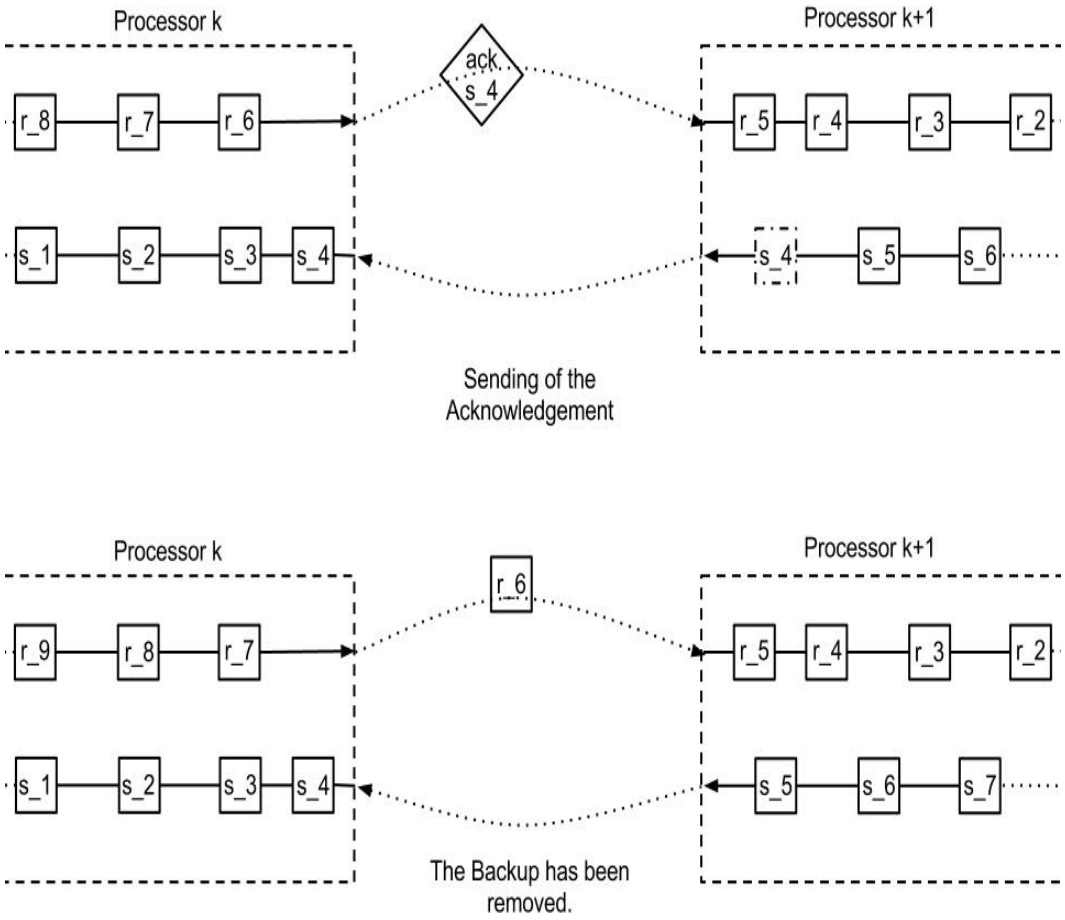


Figure 5.11: Once *processor k* receives the cluster s_4 it sends an acknowledgement back to *processor k+1* for cluster s_4. Once *processor k+1* receives this acknowledgement, it can safely delete cluster s_4 from its window. Figure taken and redrawn from [30].

5.7.3 UniDirectional Handshake Method

The two streams do not move simultaneously in the Unidirectional variant of the algorithm. This avoids the missed cluster-pair problem and the need for acknowledgments. Also the architecture of the unidirectional variant tries to minimize the messaging overhead that occurs in the Bidirectional variant.

Implementation Design

In the unidirectional variant, the stream R is first pushed through all the processors and until all the processors contain the same amount of clusters from stream R. Once this phase is done, the stream S is pushed through. During this phase, the clusters present in streams S and stream R are paired together and the distances between them found.

In the implementation, each NUMA-region has a master thread. The master threads of each NUMA-region are connected to each other through highly-efficient FIFO queues. Each master thread has a window for each of the streams R and S. See Figure 5.12 for the architecture.

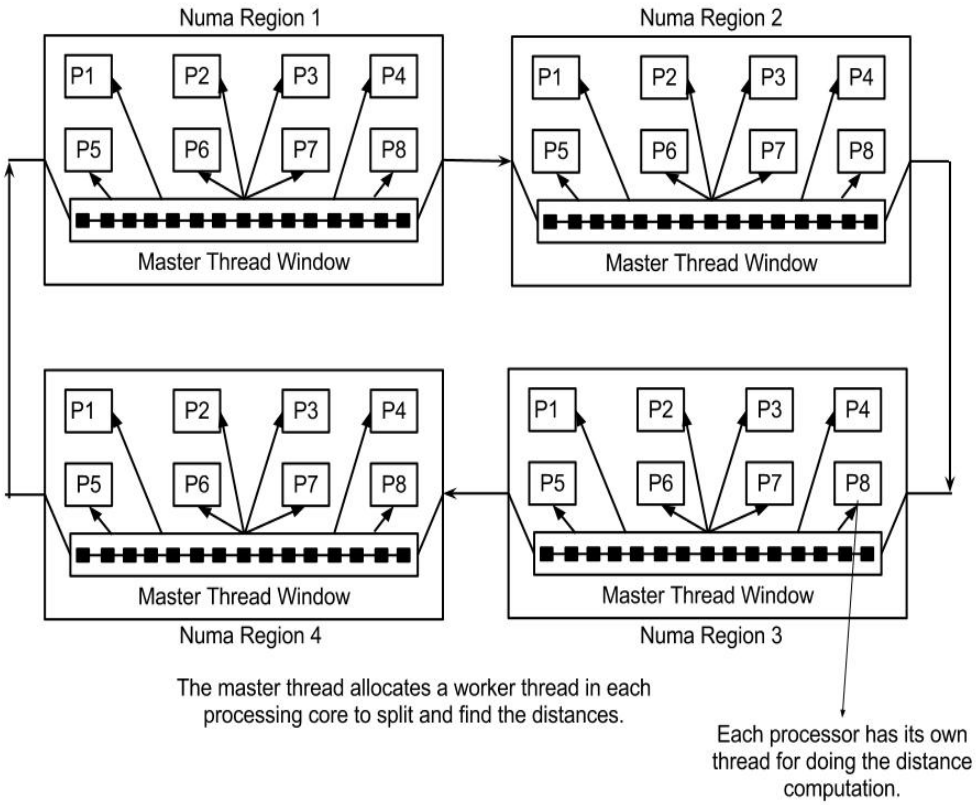


Figure 5.12: Architecture of the unidirectional variant.

First the clusters belonging to stream R are pushed through the NUMA-regions. Figure 5.13 shows this first step with four NUMA regions. Once the stream R has been pushed through all the NUMA regions, the stream S is pushed through from the opposite direction. Instead of pushing the clusters one-by-one they are grouped into chunks of a predefined size and these chunks of clusters are pushed instead. Once a master thread receives a chunk of clusters from stream S, these have to be paired with the clusters from stream R that are stored in the current master thread's window. Figure 5.14 shows the steps when the stream S is pushed through the NUMA regions.

The master thread creates worker threads, one for each processor in the NUMA-region where that master thread runs. The worker threads do the actual work of finding the distances between the cluster pairs. The master thread divides the workload equally among the worker threads and starts the worker threads. It then sends a copy of the newly arrived cluster chunks to the neighboring master thread. The worker threads calculate the distances between the clusters and store them for further lookup. They also keep track of the cluster-pairs with the smallest distance until now.

Once the worker threads complete, the master threads consolidate the closest-pair information from each of its worker threads and finds the closest cluster pair for that NUMA region until now. Then it waits until the next chunk of clusters are received. Algorithm 4 gives the pseudo code for master

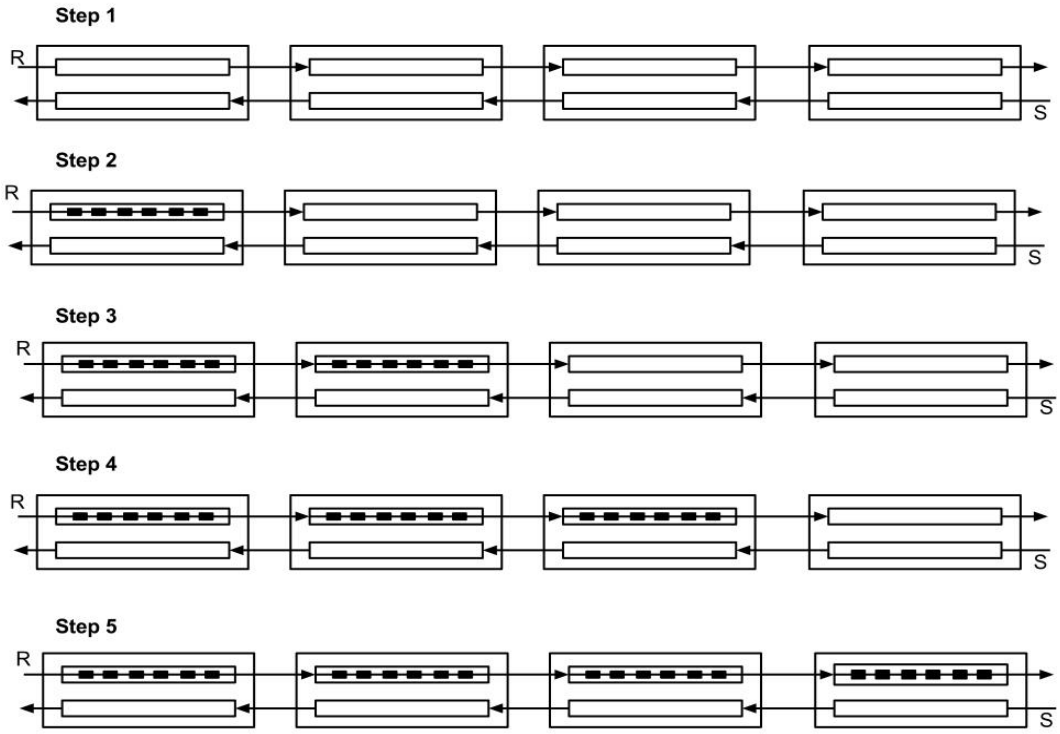


Figure 5.13: The stream R is pushed through the windows of the master threads.

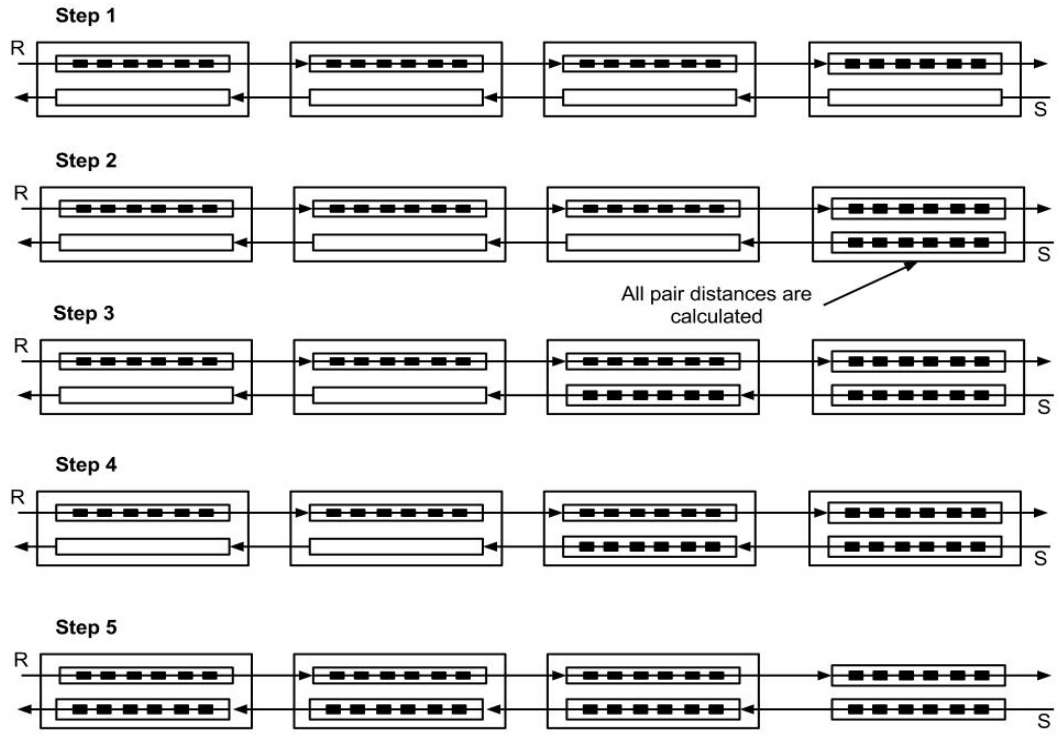


Figure 5.14: The stream S is pushed in chunks through the windows of the NUMA-region after R has already been pushed through. When a NUMA-region gets a new chunk of S it is paired with all the clusters in stream S that are stored in this master thread's window.

thread.

Algorithm 4: Pseudo code for the Master thread

```
1  while true do
2      Wait for new cluster-chunks
3      if New Cluster Chunk C_ri Received then
4          split the new chunk among W worker threads.
5          start the worker threads to process the new data.
6          send a copy of the new cluster chunk to the neighboring master
           thread.
7          while Worker Threads have not completed do
8              wait;
9          end
10         foreach worker thread W_i do
11             if currentMinimumDistance > minimumDistance of W_i then
12                 update minimum distance and mark cluster-pair which
                   has minimum distance.
13             end
14         end
15     end
16 end
```

Communication between Processors

All the processors are connected to two neighboring processors through two FIFO point-to-point links. So every processor, once it receives a new cluster and processes it, sends this new cluster to the neighboring processor and deletes it from its own window. This keeps the communication between the processors asynchronous without any central co-coordinator and results in increased parallelism.

Load Balancing

The asynchronous message passing system that has been implemented can result in an uneven distribution of load among the processors. We ran the algorithm with the *phy_train* dataset and logged the number of cluster pairs processed by each processor. This is shown in Table 5.1 as a percentage (we have rounded it to the nearest whole number). From the table, we can see that some of the processors get a lot of cluster pairs to process. This is because there is no load balancing mechanism. To get the maximum amount of parallelism, the load has to be distributed evenly among the processors. So we use a load balancing scheme that is autonomous and does not need a central co-coordinator. This is what it does - before forwarding the clusters to the neighbor, the current master thread ensures that the neighbor has almost the same number of clusters in its *R and S* windows as itself. This ensures that the neighbor is not too overloaded compared to the current master thread. In case the neighbor's windows have more clusters, the current master thread periodically checks the neighbor's window size until the load on the neighbor decreases before forwarding the clusters. From Table 5.1 we can see that once load balancing is enabled, we get a fairly even distribution of load among the processors.

Processor	*Without Load Balancing*	*With Load Balancing*
*Processor*1	13%	22%
*Processor*2	42%	26%
*Processor*3	39%	27%
*Processor*4	17%	25%

Table 5.1: Number of cluster-pairs processed in each processor with load balancing and without load balancing on the dataset *phy_train*

5.7.4 Optimizations

All the distance metrics that we mentioned in Section 3.2.1 (page 3.2.1) are commutative. That is, if C_A and C_B are two clusters, then

$$d(C_A, C_B) = d(C_B, C_A)$$

So when we calculate the distance matrix, only values above the top diagonal need to be calculated. That is, for $i < j$, if we calculate $d(C_i, C_j)$ we do not calculate $d(C_j, C_i)$. To achieve this in the implementation, we give each cluster an unique numerical id. During distance calculation, any distance $d(C_A, C_B)$ between two clusters C_A and C_B is calculated if and only if

$$\text{Id of } C_A < \text{Id of } C_B$$

5.8 Subsequent iterations

In the subsequent iterations, we reuse the distances computed in the first iteration. Let C_i be a newly created cluster in iteration i. We need to find the distances between this new cluster and all the other clusters. For all the remaining pairs of clusters, we already have computed the distances during the first iteration. Since the dissimilarity matrix was split equally among the NUMA regions, each processor can look up the distances in its local NUMA regions. This avoids any cross-region access and makes this step NUMA optimized. Each processor then sends its locally closest pair to the master thread of the NUMA-region that the processor belongs to. The master threads compute the closest cluster-pair for each NUMA-region. Finally, one of the master threads uses this information to find the globally closest cluster-pair and merges those clusters. We select one master thread randomly during the start of the algorithm and designate it execute the second step. The information about the

merged cluster-pair and the new cluster is then sent to all the processors and then the next iteration is started. So during each iteration, each processor has to send its local closest cluster pair and also receive the newly created cluster pair. As the size of this information is very small we use broadcasting to achieve this.

5.9 Implementation Details

To implement the above methods, we need NUMA-aware memory allocation, NUMA-aware freeing of memory, NUMA-aware thread allocation and so on. We use an API called libnuma [36] which provides methods to do the above. It exposes methods for running a specific thread on a specific node and allocating memory on a specific node. The method *numa_alloc_onnode* is used to allocate memory on a specific NUMA-node and the method *numa_run_on_node* is used to run a particular thread on a specific NUMA-node. But, all the memory allocation methods in this library are in the kernel mode and so are very slow. For our algorithm, we needed to allocate small pieces of memory many times and for this purpose the libnuma API's are a big overkill. The solution is to use a memory manager which is NUMA aware. This memory manager must use the libnuma library to allocate a big chunk of the memory and use this big chunk of memory to satisfy any further memory requests. Patryk Kaminski [14] has written an implementation of one such memory manager. This is essentially a modification of Google's tcmalloc library to create a *NUMA aware malloc function*. We found that this new library resulted in an improved performance over using the default NUMA APIs. So all our experiments were done by using this memory manager.

5.10 Impact of the cache

As explained in 4.2.1, every CPU in modern computers is equipped with a CPU cache which helps reduce the time to access the memory. Usually the caches are layered as a hierarchy and most computers have the following three layers:

- L1 cache.
- L2 cache.
- L3 cache.

When the processor has to do a memory access, it checks the three layers of the cache first. If the data is not present in the cache, the processor executes a memory access. In NUMA systems, the cache is used to hide the cross-region access latency. Whenever the processor executes a cross-region access, it reads a chunk of the memory and keeps it in its cache. If the next request is for data that is present in the cache, then there is no need for the cross-region access and so it increases performance. This works well if the amount of cross-region access is small. But if there are a large number of cross-region accesses, or the amount of data which is accessed across regions is larger than the cache size, then the efficiency of the cache is reduced. In those cases, the cross-region access latency is not hidden anymore and results in decreased performance if the program is not NUMA-aware.

5.11 Experimental Results

As explained earlier, the bidirectional variant was slower than the unidirectional variant. Table 5.2 gives a comparison of the bidirectional variant and

the unidirectional variant. So all the results reported here are using the unidirectional variant of the algorithm.

Dataset	*Bidirectional Variant*	*Unidirectional Variant*
Bio_test	0.94	1.32
Phy_train	0.96	1.34
Bio_train	0.97	1.3

Table 5.2: Speedup obtained on some datasets using the Bidirectional variant and the Unidirectional variant. It can be seen that the Bidirectional variant is slower than the naive Multithreaded HAC algorithm.

As explained in Section 4.2.3 (page 31), the experiments were run on an AMD Opteron machine with four NUMA-regions of eight processors each. We could not run tests on datasets with more than 60,000 data points because of memory limitations and so for bigger datasets we used a prefix of the dataset that would fit into the available memory.

Graph 5.15 gives the speedup obtained on a number of datasets by the unidirectional variant. The datasets have been taken from the UCI machine learning repository [40] and the KDD repository [35]. We also used the Mnist handwritten digit recognition dataset [39].

It can be seen that the speedup is in the range of *1.3* to *1.5* on our machine. It can also be observed that the Mnist dataset has a slightly higher speedup when compared to the other datasets. We suspect that this is because of the large size of the Mnist dataset and so it might overflow the cache. We also ran the two algorithms on datasets obtained as a prefix of the Mnist dataset. Graph 5.16 gives the speedup obtained on prefixes of various sizes. It can be seen that the speedup is lower for smaller sizes but hits a high of about *1.45* times. It then remains constant even after the dataset size is increased. This is probably because of the cache impact outlined in Section 5.10 (page 54). Until

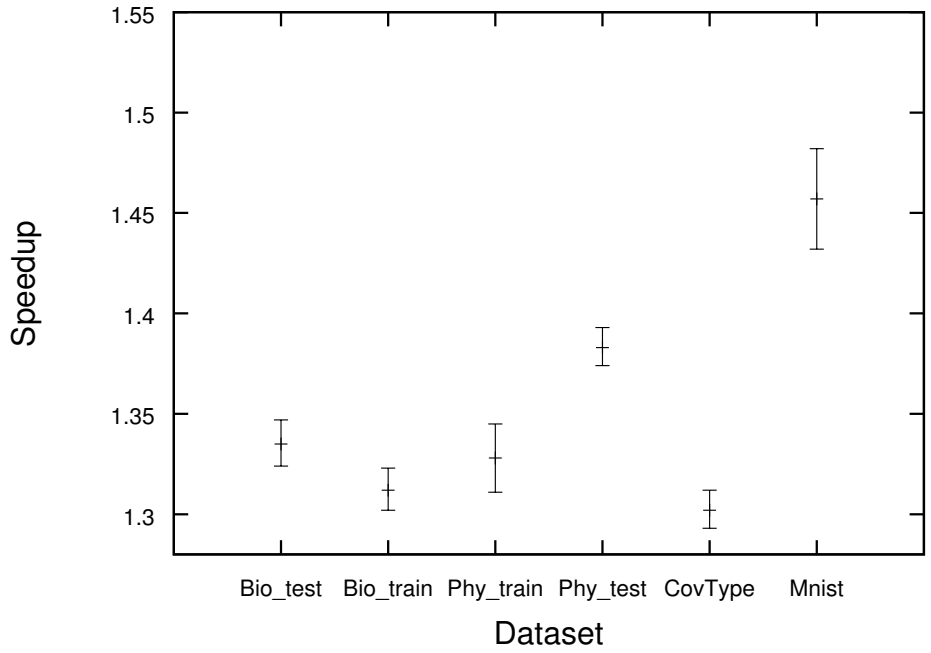


Figure 5.15: Speedup obtained by the multi threaded unidirectional variant of NUMA aware HAC over the naive multi threaded HAC with 32 processors.

a particular size, the caches remain effective and the speedup is not great. But once the size is greater than the cache size, the naive version becomes slower than the NUMA aware version. After a particular size, saturation is reached and the speedup remains constant.

Graph 5.17 gives the scalability of the algorithm when compared to the ideal scale up for the multi threaded unidirectional variant of NUMA aware HAC algorithm.

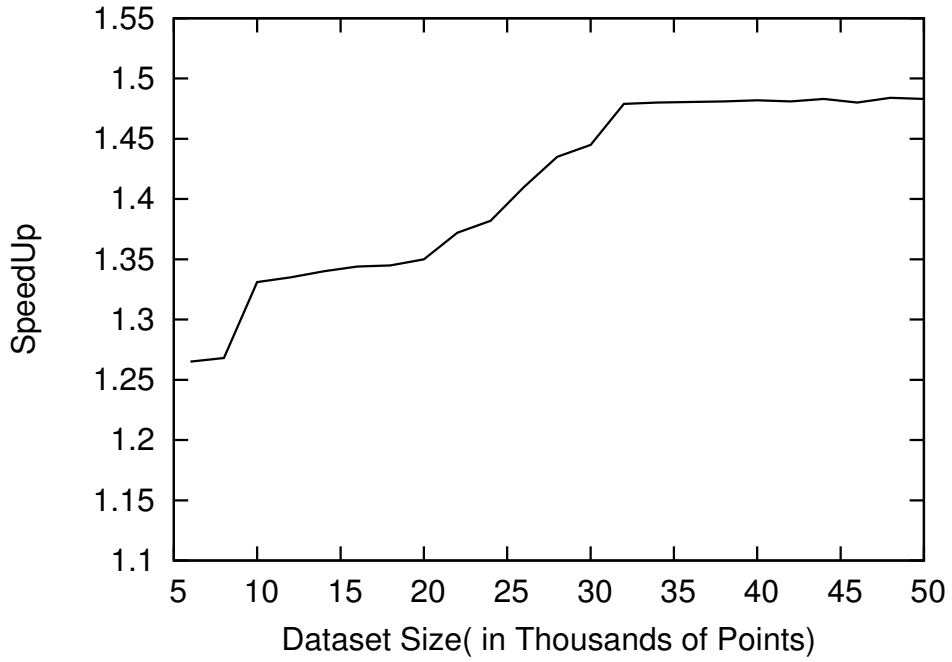


Figure 5.16: Speedup obtained by the multi threaded unidirectional variant of NUMA aware HAC over the naive multi threaded HAC on different-sized prefixes of the Mnist datasets.

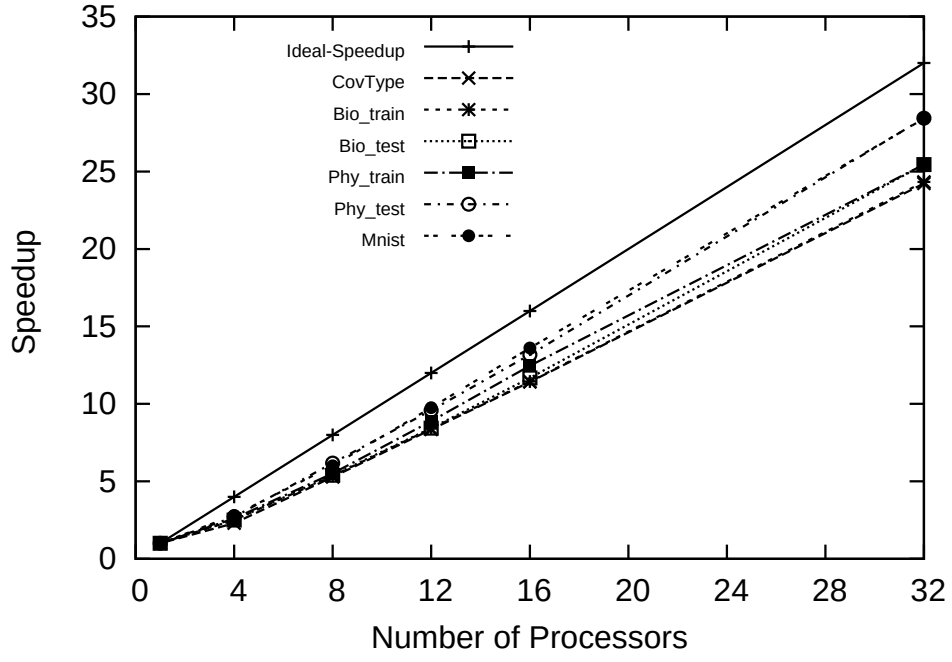


Figure 5.17: The speedup obtained when the number of processors are increased compared to the ideal speedup for the multi threaded unidirectional variant of NUMA aware HAC algorithm.

Chapter 6

Nearest Neighbor Based Hierarchical Clustering Algorithm

In this chapter we first outline a HAC algorithm using Nearest Neighbor search. Secondly we present the research done on speeding up the Nearest Neighbor search and the different data structures that can be used to speed up the Nearest Neighbor search. Thirdly we present the different requirements that have to be met by any data structure used to speed up Nearest Neighbor search in this clustering algorithm and show that a data structure named *Cover Tree* satisfies all those requirements. Then we describe the Cover Tree data structure and the way it will be used to speed up the HAC algorithm. Finally, we present the results of the experiments that we conducted when using the Cover Tree for HAC.

In most cases it is impractical to use an HAC algorithm that uses $O(N^2)$ space. In these cases, we are interested in HAC algorithms that uses the stored data methods mentioned in Chapter 3. One way to implement this is by using Nearest Neighbors. In this chapter, we explain the naive brute force way to implement the Nearest Neighbor based HAC algorithm. We then analyze the

time and space complexities of this algorithm for different linkage criteria. Then we use a new data structure called the Cover Tree [1] which speeds up the Nearest Neighbor based HAC algorithm.

6.1 Nearest Neighbor Search

Nearest Neighbor Search is the problem of preprocessing a set of points S in a metric space (M, d) so that given a query point $p \in M$, we can find the point that is closest to p in set S. This is a well studied and classical problem that has numerous practical applications. These include compressing data, querying databases, machine learning, biological computations and reducing dimensionality.

6.2 Nearest Neighbor based HAC

The Nearest Neighbor based HAC algorithm uses the Nearest Neighbors of every cluster to obtain the same greedy clustering as the traditional HAC algorithm. The algorithm creates and maintains a Nearest Neighbor array. A Nearest Neighbor array is a N-tuple $P = (P_1, ..., P_N)$ that identifies for each cluster i, a nearest-neighboring cluster P_i: each P_i for $1 \leq i \leq N$ satisfies the condition that $d(i, P_i) = min(d(i, j) : 1 \leq j \leq N, i! = j)$ [23]. In the Nearest Neighbor search domain, this is called the 2-Nearest Neighbor search (The first Nearest Neighbor of a point p in dataset S if $p \in S$ is always p). Algorithm 5

gives the pseudo code for the naive Nearest Neighbor based HAC algorithm.

Algorithm 5: Nearest Neighbor Based HAC

Input: Data(N,D)

Output: A Hierarchy of Clusters.

1 Compute Nearest Neighbor Array NN_Array

2 **for** $i = 1$ *to* $N - 1$ **do**

3 Find closest pair of clusters (C_p, C_q) from the NN_Array

4 Merge the closest pair of clusters

5 Add a new entry in the NN_Array for the new cluster and find the Nearest Neighbor of the new cluster.

6 Update all the entries in the NN_Array which had C_p or C_q as their Nearest Neighbor

7 Output the pair of clusters that have been merged.

8 **end**

6.3 Time Complexity

The time complexity of Algorithm 5 is dependent on the time complexity to find the inter-cluster distances. The different linkage criteria have different time complexities for finding the distances between clusters. For example, centroid linkage has constant time complexity for finding the inter-cluster distance and other linkage criteria like single link, complete link and average link do not have constant time complexity for finding the inter-cluster distances. So we first define the time complexity of the algorithm outside of the time complexity for finding inter-cluster distances. Then in the following subsections, we explain the time complexities with the different linkage criteria.

In Algorithm 5, step 1 performs a Nearest Neighbor search to initially populate the NN_Array. If we use a brute force method for Nearest Neighbor search, then for each cluster, we have to find the distance with all other clusters

to find the Nearest Neighbor. This has a time complexity of $O(N^2)$. An efficient way to implement the NN_Array is by using a priority queue. In that case, step 3 takes a time complexity of $O(1)$. Step 5 of the algorithm has a time complexity of $O(N)$ if we use the brute force Nearest Neighbor algorithm. In Step 6, we update the NN_Array for those clusters which are affected by the merger of C_p or C_q. Let the number of clusters affected be α. Then the time taken for this step, using a brute force Nearest Neighbor search is $\alpha O(N)$. According to Anderberg [23], the number of clusters for which the NN_Array needs to be updated averages to a constant for each iteration. In that case, the complexity of this step is $O(N)$. So the time complexity of the Nearest Neighbor based HAC algorithm outside of the time complexity for finding inter-cluster distances is $O(N^2)$. In the subsequent subsections, we will see the time complexity of the Nearest Neighbor based HAC specifically for the different linkage criteria.

6.3.1 Centroid Linkage Criteria

In the centroid linkage criteria, every cluster is represented only by its centroid. Calculating the distance between clusters using the centroid linkage criteria has constant time complexity. But the centroid linkage criteria does not satisfy the reducibility property.

Reducibility Property

Let us consider three clusters C_i, C_j and C_k and a distance function d. Let C_i and C_j be mutual Nearest Neighbors and let $C_i \cup C_j$ be a cluster formed by merging C_i and C_j. The distance function d is said to be reducible if it satisfies the following inequality:

$$d(C_i \cup C_j, C_k) \geq min(d(C_i, C_k), d(C_j, C_k))$$

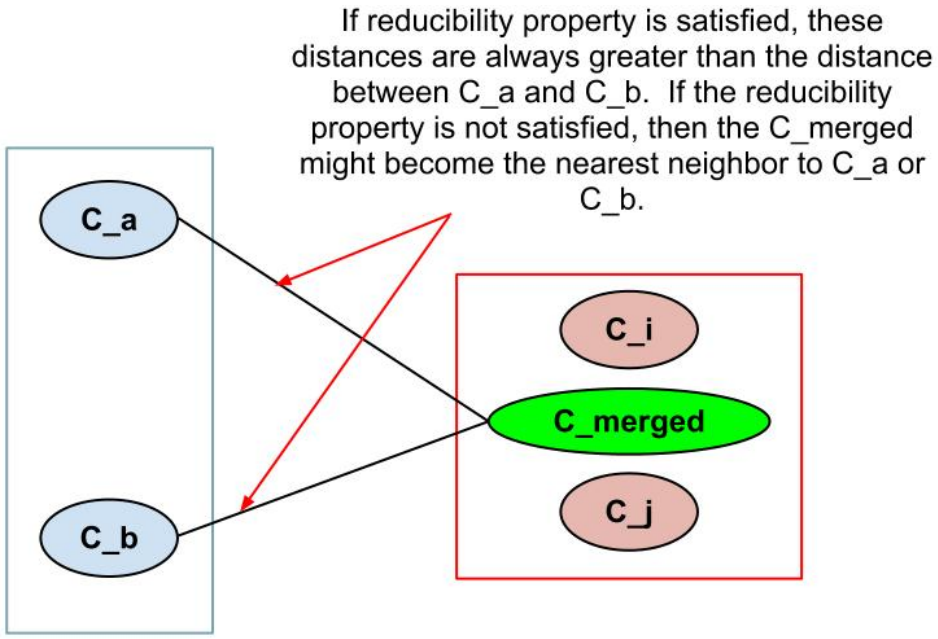


Figure 6.1: An illustration of the reducibility property.

Other linkage criteria like single link, complete link and average link satisfy the reducibility criteria. Since the centroid linkage criteria does not satisfy the reducibility property, we have the following problem. Consider a cluster C_a which has C_b as its Nearest Neighbor. Let C_i and C_j be two clusters that are being merged. If the linkage criteria satisfies the reducibility criteria, then the merged cluster C_{merged} would not be a Nearest Neighbor of either C_a or C_b. But if the reducibility property is not satisfied, this can happen. Figure 6.1 illustrates this. So if the reducibility criteria is not satisfied, in Algorithm 5, we have to also check whether the newly merged cluster becomes the closest neighbor of any other existing cluster. This has time complexity $O(N)$. The total time complexity of this algorithm is $O(N^2)$ [3].

6.3.2 Complete Linkage Criteria and Average Linkage Criteria

HAC with complete linkage criteria uses the maximum distance between two points from each cluster as the distance between the clusters. Similarly, HAC

with average linkage criteria uses the average distance between points as the distance between clusters. Unlike centroid linkage based clustering, these two do not have a constant-time method for computing the distance between the clusters. If we store and reuse the distances between the points using a dissimilarity matrix, then this algorithm has $O(N^2)$ space and time complexity. Hence in this case, it becomes a stored matrix algorithm.

6.3.3 Single Linkage Criteria

The single linkage criteria uses the minimum distance between the points as the distance between the clusters. Like complete link and average link, there is no constant-time method to find the distance between clusters using single linkage criteria. If we store and reuse the distances between points, then the space and time complexity is $O(N^2)$ and this also becomes a stored matrix method. However, HAC using single linkage criteria can be effectively done by another algorithm that computes the minimum spanning tree of the distances using Prim's Algorithm [25]. This method is called SLINK [27] and has a time complexity of $O(N^2)$ and space complexity of $O(N)$. That method would be preferable to the Nearest Neighbor method when single linkage is used.

So the Nearest Neighbor based HAC is most useful when the centroid linkage is used as it is a stored data algorithm. The remaining part of this chapter is focused on using a data structure called Cover Tree for speeding up this algorithm.

6.4 Intrinsic Dimensionality and Expansion Constant

The naive Nearest Neighbor search can be used when the structure of the dataset is unknown. But datasets that arise in practice usually have a nicer structure which can be exploited to get much better bounds on the Nearest Neighbor search. In a high-dimensional dataset, some of the dimensions of the dataset might be redundant: they can be calculated from the other dimensions. During clustering, these redundant dimensions do not have a significant impact when calculating the dissimilarities. The *Intrinsic Dimensionality* of a dataset is the number of dimensions of the dataset that are non-redundant and have a significant impact for calculating dissimilarities. One notion of Intrinsic Dimensionality is called the *Expansion Constant* or the *Doubling Constant* [1]. Given a set S of N points, let p be a point belonging to S. We denote the closed sphere of radius r around p in S by $B_S(p, r) = q \in S : d(p, q) \leq r$. When the context is clear, we just write $B(p, r)$. The Expansion Constant of S is defined as the smallest value $c \geq 2$ such that $|B_S(p, 2r)| \leq c|B_S(p, r)|$ for every p and $r > 0$. In other words, if we have a sphere of radius r surrounding p, then the Expansion Constant is the ratio of the number of points in a sphere of radius $2r$ centered at p to the number of points in the sphere of radius r centered at p. The Expansion Constant of a dataset is important as the Nearest Neighbor search algorithms have theoretical bounds that depend on the Expansion Constant. This will be described in the following sections. The

algorithm to find the Expansion Constant is given in Algorithm 6.

Algorithm 6: Algorithm for finding the Expansion Constant c

Input: Data(N,D)

Output: Expansion Constant c

```
set c = 0
for Every point p ∈ Data do
    Find the distance to every other point and sort the distances into
    array sortedDistArray.
    for Every Distance d ∈ sortedDistArray do
        N1 = Number of points that are within a d distance away from p
        N2 = Number of points that are within a 2 * d distance away
        from p
        c = max(c, N2/N1);
    end
end
return c
```

We can optimize the Nearest Neighbor based HAC by speeding up the Nearest Neighbor search. The next section gives the different research that has been done to speed up the Nearest Neighbor search.

6.5 Related Work in speeding up Nearest Neighbor Search

There are different variations of Nearest Neighbor Search problems. The metric space (M, d) may be application specific (weighted edit distance between strings) or infinite (Euclidean space). Also there may be special considerations, where the pre-processing stage needs to be condensed or the metric space might be unknown. Most research is focused on the case where the space is Euclidean.

Several algorithms and metric structures have been proposed to exploit the case where the data has high dimensionality with a low intrinsic dimensionality. An approach when the dimension is small is to use KD-trees [9]. Karger and Ruhl [15] proposed a randomized algorithm called the *Metric skip list* for metric spaces in which the intrinsic dimensionality is small. To construct this data structure, they introduce a random ordering of the points in the sample space S. Then based on this random order, the data structure is constructed in a deterministic way. Even though this algorithm uses randomization, they prove that it will always return the correct results. The randomization enables the algorithm to reach the nearest neighboring point in an optimized way when the intrinsic dimensionality is small. Krauthgamer and Lee have proposed a data structure called Navigating Nets [18]. The Navigating Nets is a directed acyclic graph and is much simpler than the *metric skip list* while giving the same speed up as the *metric skip list*. Beygelzimer et.al [1] have proposed a simple tree based-data structure called the Cover Tree which is very similar to the Navigating nets while having better space bounds than Navigating Nets. We discuss the space of and time complexities of these algorithms in the section below.

6.6 Comparison of the different Nearest Neighbor Algorithms

In this section, we present a comparison of the different approaches used to speed up Nearest Neighbor search that were outlined in the previous section. Table 6.1 presents the case where there is no assumption about the Expansion Constant of the dataset.

Now we focus on the case, where we know the Expansion Constant of the dataset. Including the Expansion constant in the analysis of the algorithms helps us compare the algorithms much more exactly when the datasets have

	Cover Tree	Navigating Net	Metric Skip List
Construction Space	$O(N)$	$O(N^2)$	$O(N^2)$
Construction Time	$O(N^2)$	$O(N^2)$	$O(N^2)$

Table 6.1: Nearest Neighbor methods comparison without assumptions about the Expansion Constant [1]

varying Expansion Constants. Let c be the expansion constant of a dataset. Table 6.2 gives the comparison of the different data structures taking into account the Expansion Constant c and the size of the dataset N.

	Cover Tree	Navigating Net	Metric Skip List
Construction Space	$O(N)$	$c^{O(1)}N$	$c^{O(1)}NlogN$
Construction Time	$O(c^6nlogN)$	$c^{O(1)}NlogN$	$c^{O(1)}NlogN$
Insertion/ Removal	$O(c^6logN)$	$c^{O(1)}logN$	$c^{O(1)}logN$
QueryTime	$O(c^{12}logN)$	$c^{O(1)}logN$	$c^{O(1)}logN$

Table 6.2: Nearest Neighbor methods comparison with the assumption that the Expansion Constant is c [1]

6.7 Data Structure Needed for HAC

Any data structure that can be used to improve the Nearest Neighbor search in the Nearest Neighbor based HAC algorithm should satisfy the following requirements:

- Space Requirement - The space required by the data structure should be less than $O(N^2)$. Ideally, the space requirement should be $O(N)$.
- Simplicity - The data structure should be relatively simple to understand

and implement.

- Practical Speed - In addition to having strong theoretical bounds, it should be fast in practice.

From Table 6.2, it can be seen that the Navigating Nets and the Cover tree have lower bounds than the Metric Skip List. The main advantage of the Cover Tree over the Navigating Nets is that, the Cover Tree has a space bound that is independent of the Expansion Constant. It takes $O(N)$ space irrespective of the Expansion Constant of the dataset. Thus, the Cover Tree should perform better than the Navigating net for larger datasets. The Cover Tree is also a simple tree-based structure that is easy to implement and manipulate. For our purpose here, a tree is a connected di-graph with N vertices and N-1 edges, and a distinguished vertex called the root, with no incoming edges and only outgoing edges. The distance between any two nodes U and V is the number of edges between them. All nodes at distance d from the root are said to be at *level d*. Because of its simplicity and efficient space and time bounds, we have chosen the Cover Tree as the data structure to speed up then Nearest Neighbor based HAC.

6.8 Cover Tree

A *Cover Tree* T on a dataset S is a tree where every node in the tree contains a data point from the dataset S and each level of the tree acts as a cover for the descendents below it. Each level is indexed by an integer i which decreases from the root as the tree is descended. The data points from dataset S are distributed among the nodes of the tree in such a way that a number of invariants are satisfied. These invariants are discussed below.

An example Cover Tree is shown in Figure 6.2

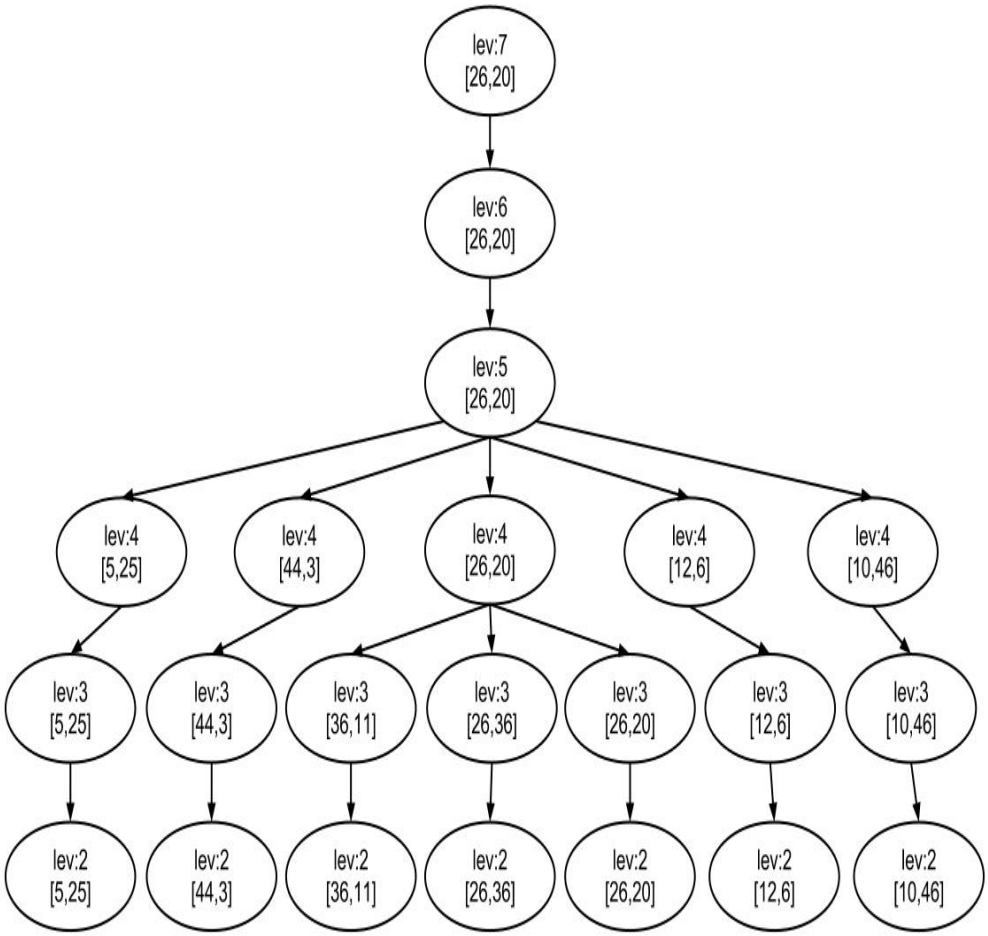


Figure 6.2: An example Cover Tree with seven elements. Figure taken from [17]

6.8.1 Cover Tree Invariants

Let C_i be the set of nodes at level i. Any Cover Tree T on dataset S satisfies the following invariants:

- **Nesting** - $C_i \subset C_{i-1}$. A diagrammatic representation is shown in Figure 6.3.
- **Covering Tree** - For every $p \in C_i$, there exists a $q \in C_{i+1}$ satisfying

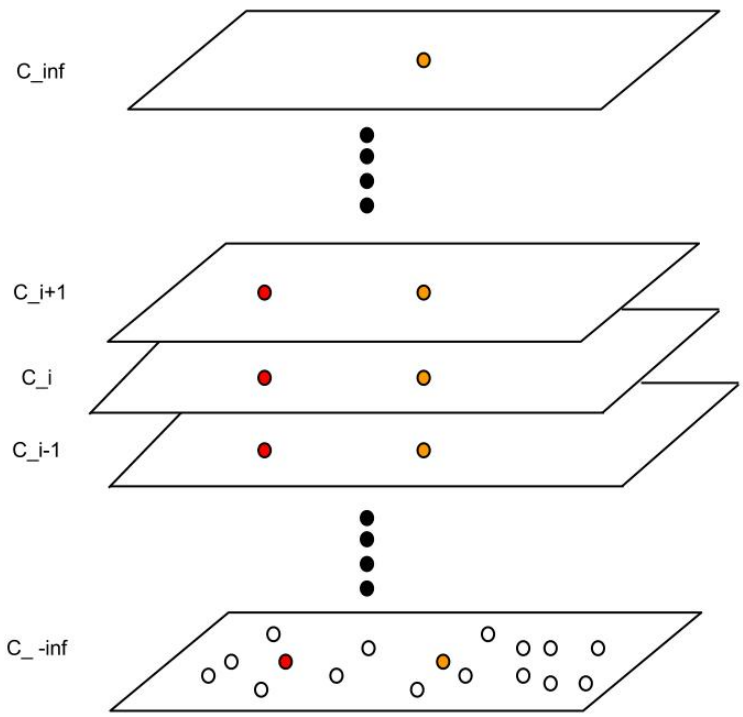


Figure 6.3: A diagrammatic representation of nesting. The orange node occurs first in level C_{inf} and is present in all levels below it. The red node occurs first in level C_{i+1} and is present in all levels below it. So every level C_{i-1} is a subset of the level C_i. Figure taken from [37]

$d(p, q) \leq 2^{i+1}$. A diagrammatic representation is shown in Figure 6.4.

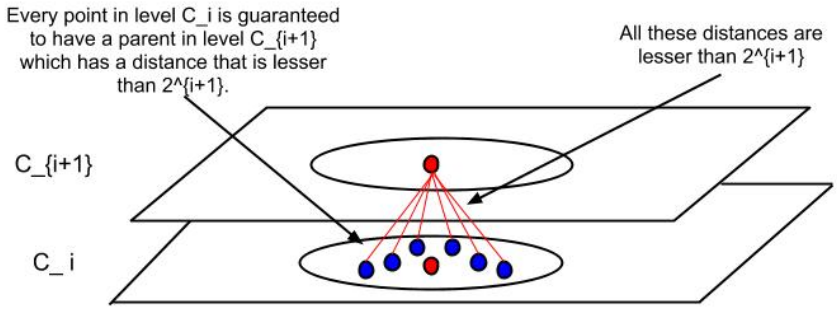


Figure 6.4: A diagrammatic representation of covering. The blue nodes occur first in the level C_i. It will be guaranteed that for every node p in level C_i there will be a node q (the red node) in level C_{i+1} which satisfies $d(p, q) \leq 2^{i+1}$. Figure taken from [37]

- **Separation** For all $p, q \in C_i$, $d(p, q) > 2^i$. A diagrammatic representation is shown in Figure 6.5.

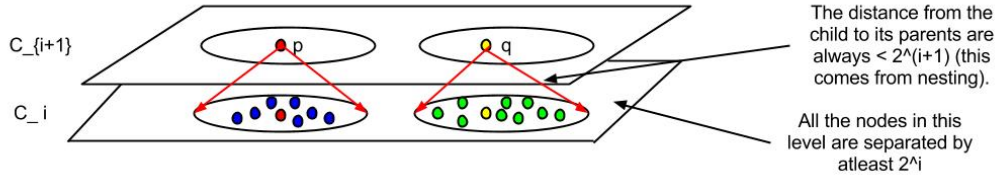


Figure 6.5: A diagrammatic representation of separation. Figure taken from [37]

6.8.2 Cover Tree Representation

We use two kinds of representation for the Cover Tree:

- Implicit Representation.
- Explicit Representation.

The *Implicit* representation, consists of infinitely many levels C_i. The level C_∞ is the root of the tree. The Covering Tree invariant ensures that if a node is present in level i, then it should be present in all levels below i. Also, each node becomes its own child in the subsequent levels. If this representation is used in the implementation, we cannot meet the space bound of $O(N)$ as we will be storing the same node over and over again at the different levels. So we use this representation mainly for understanding the algorithms of the Cover Tree. For example, consider the two dimensional points given in graph 6.6. Figure 6.7 gives a Cover Tree constructed with these points in implicit representation.

The *Explicit* representation coalesces all the nodes in which the only child is the self child. Thus for any given node n_i, which appears first in level i it

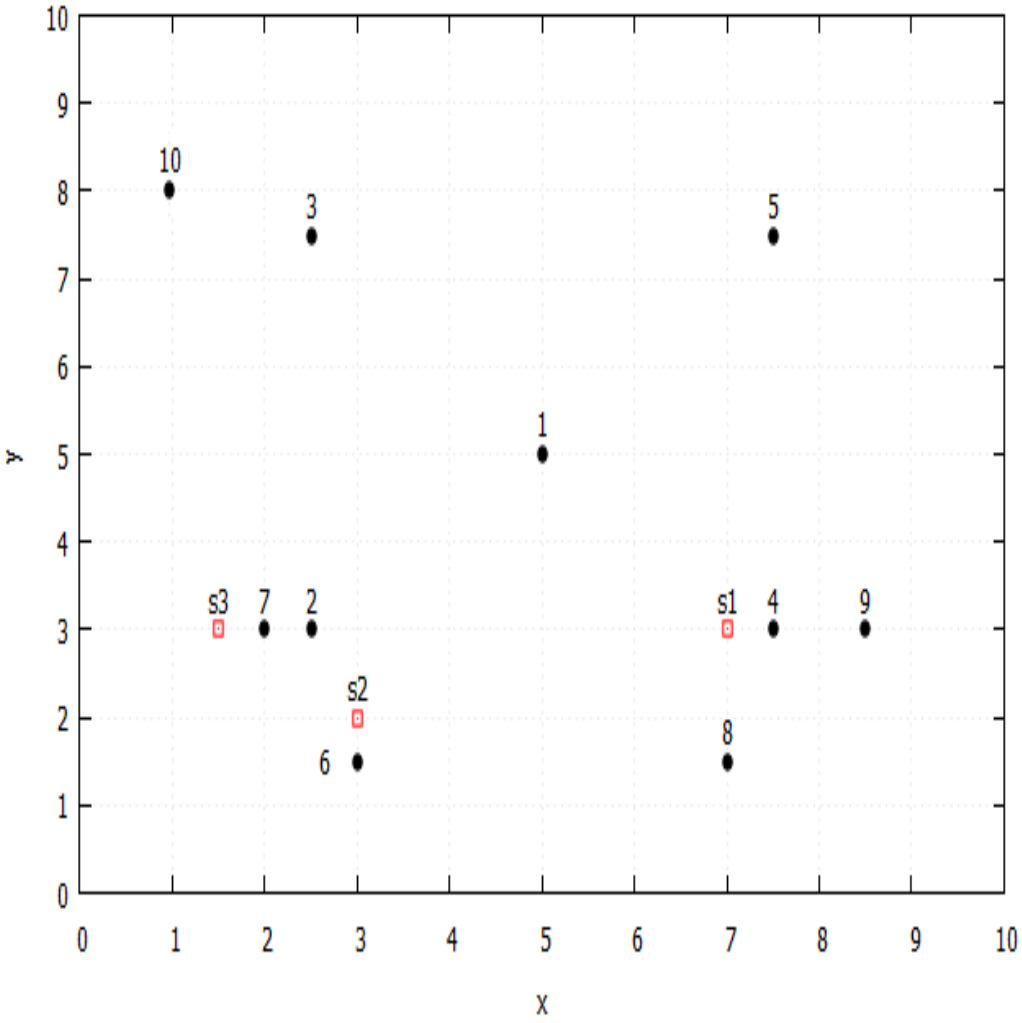


Figure 6.6: Graph of points used in the Cover Tree. Figure taken from [37]

is assumed that it is its own child in every level below i. We only store the non-self children of n_i in each level. This saves a lot of space. Figure 6.8 gives the explicit representation of the same Cover Tree given in Figure 6.7.

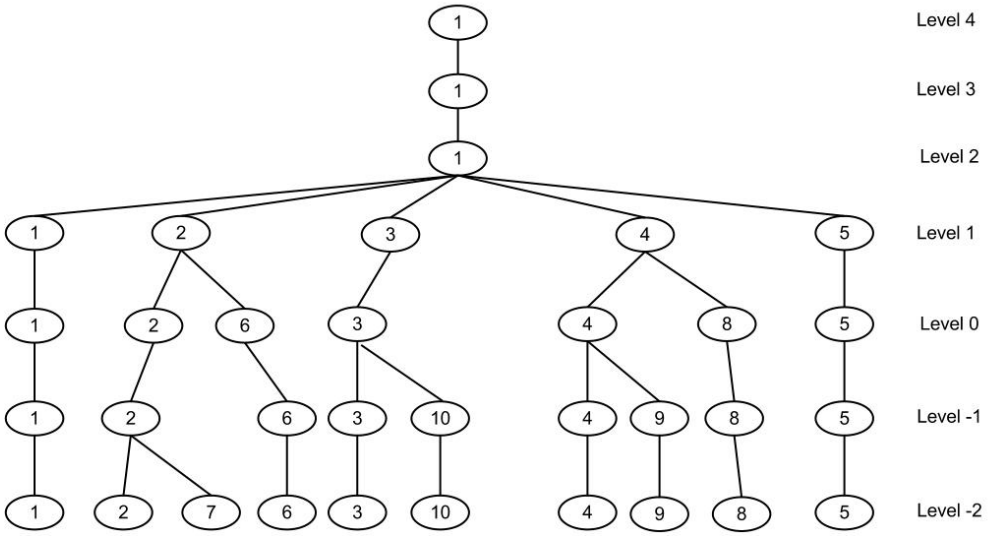


Figure 6.7: An example of an Implicit representation of a Cover Tree constructed with the points given in the graph 6.6.

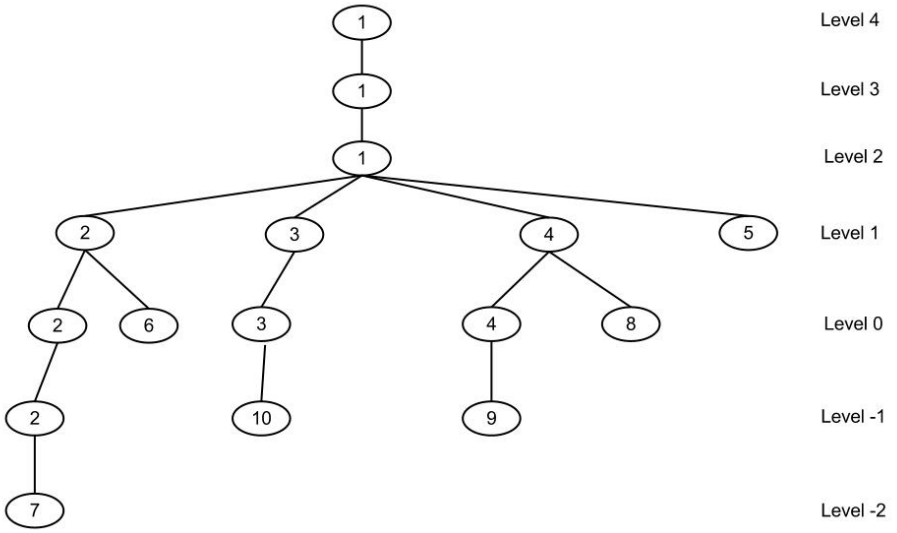


Figure 6.8: An example of an Explicit representation of a Cover Tree constructed with the points given in the graph 6.6.
This d

It should be noted that Figure 6.8 shows the same node repeatedly as its child only for demonstration purposes. In the implementation, the Cover Tree is stored using hash tables for good space utilization. For each level i of the Cover Tree there is a hash table H_i. If a node $Node_k$ has children at level i, then an entry is made in the hash table H_i. The key is $Node_k$ and all the

children of that node at that level are the values. No entry is made in any hash table for self-chilren. The algorithms that deal with this representation are aware that in a Cover Tree, if a node occurs in level i then it occurs in every level $i-1$. This saves space by removing redundancy. Figure 6.9 gives an example storage of the Cover Tree given in Figure 6.8 using hash tables.

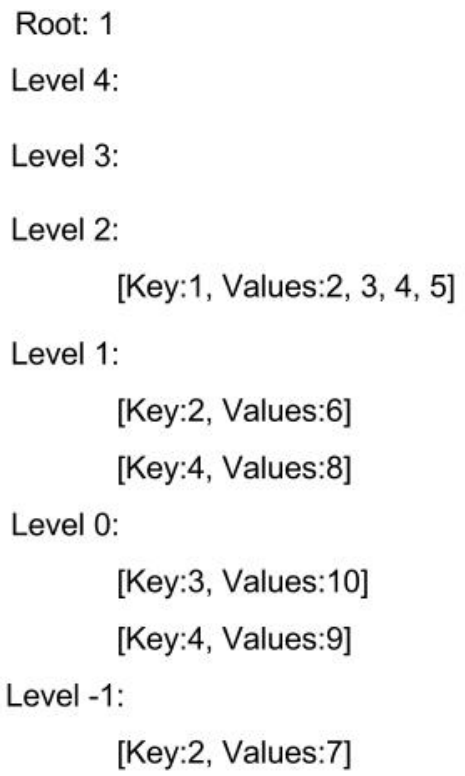


Figure 6.9: The explicit representation of the Cover Tree given in Figure 6.8 stored using hash tables in the implementation.

6.8.3 Space Requirement

If we use the Explicit representation of the Cover Tree, for a dataset containing N points, the space requirement becomes $O(N)$. This can be proved as follows:

In the Cover Tree, each node is connected to at most one parent. Also in a Cover Tree, if a node first occurs in level i, then it is present in all levels below i. The Explicit representation does not store this node repetitively for

every level. Instead the node is stored on the level i that it occurs first. So each node in the Cover Tree has a parent other than itself and also its children cannot be itself. Thus, there are N-1 unique nodes in the representation with N unique links and so the space bound is $O(N)$.

6.8.4 Finding the Nearest Neighbor

To find the Nearest Neighbor of a point p, we descend the tree keeping track of a subset $Q_i \subset C_i$ of nodes whose descendents might be Nearest Neighbors of p. The algorithm iteratively constructs the next subset Q_{i-1} of points by expanding the points in Q_i to its children in C_{i-1}, and then throwing away the points that cannot lead to the Nearest Neighbor of p. This set of points that are obtained at each level is called the *cover set*.

First we outline the algorithm to find the Nearest Neighbor and then we prove its correctness. For the sake of understanding, let us consider an implicit tree with infinite levels. Let $Children(p)$ denote the set of children of node p. Let $d(p, Q)$ denote the distance of p to the nearest point in a set Q. That is,

$$d(p, Q) = min_{q \in Q} d(p, q)$$

Algorithm 7 gives the algorithm for finding the Nearest Neighbors of a

point.

Algorithm 7: Algorithm to find the Nearest Neighbors in the Cover Tree

Input: Cover Tree T, Query Point p

Output: Nearest Neighbor of p

1 set $Q_\infty = C_\infty$

2 **for** $i = \infty \rightarrow -\infty$ **do**

3 consider the set of children of Q_i: $Q = Children(q) : q \in Q_i$

4 form next cover set: $Q_{i-1} = q \in Q : d(p,q) \leq d(p,Q) + 2^i$

5 **end**

6 return $argmin_{q \in Q_{-\infty}} d(p,q)$

Proof of Correctness: For any point q in C_{i-1}, the distance between q and any descendant of q, namely q' can be given by

$$d(q,q') \leq \Sigma_{j=i-1}^{-\infty} 2^j = 2^i$$

We set the distance bound of the search to $d(p,Q) + 2^i$. So we ensure that step 4 of the Nearest Neighbor algorithm never throws out a grandparent of the Nearest Neighbor of the point p. Eventually, we are assured that the Nearest Neighbor of the point p will be in the final cover set. The brute force search in line 6 gives us the Nearest Neighbor. Figure 6.10 gives a diagrammatic representation of this concept. The hollow point is the query point. First, in level C_i, all the red points indicate the points which are within the distance bound to the query point. Then in level C_{i-1} all the red points are present (because of the nesting property) and in addition we also have new points, the blue points, which are children to the points in level C_i. In a similar way, in level C_{i-2}, we have more points (yellow points), in addition to the already existing points.

As an example let us consider the Cover Tree in Figure 6.7 constructed using the points given in 6.6. Figures 6.11, 6.12 and 6.13 give an example of

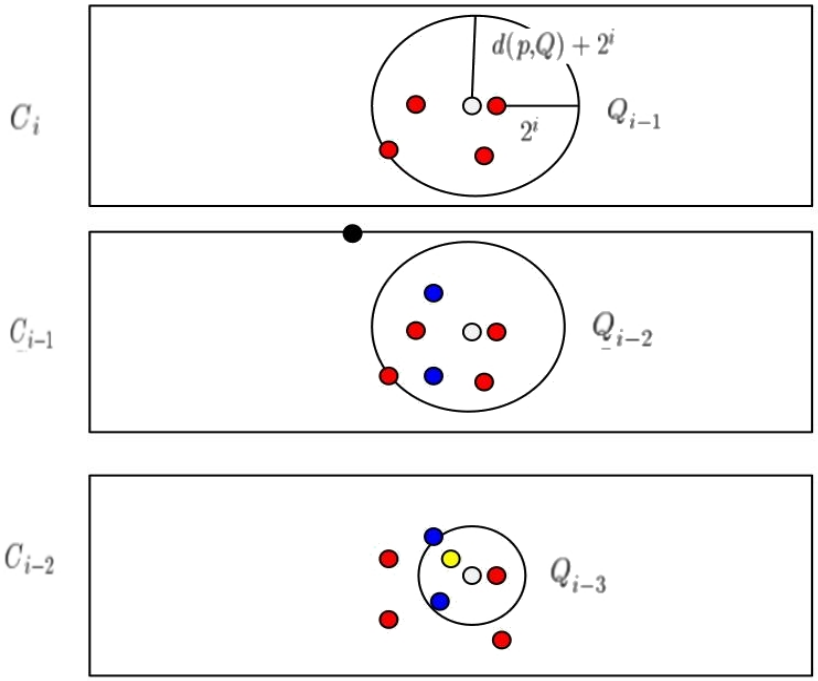


Figure 6.10: A representation of the Nearest Neighbor Search. The hollow point is the query point. Figure taken from [1]

the search process for points $s1$, $s2$ and $s3$ respectively in the graph 6.6.

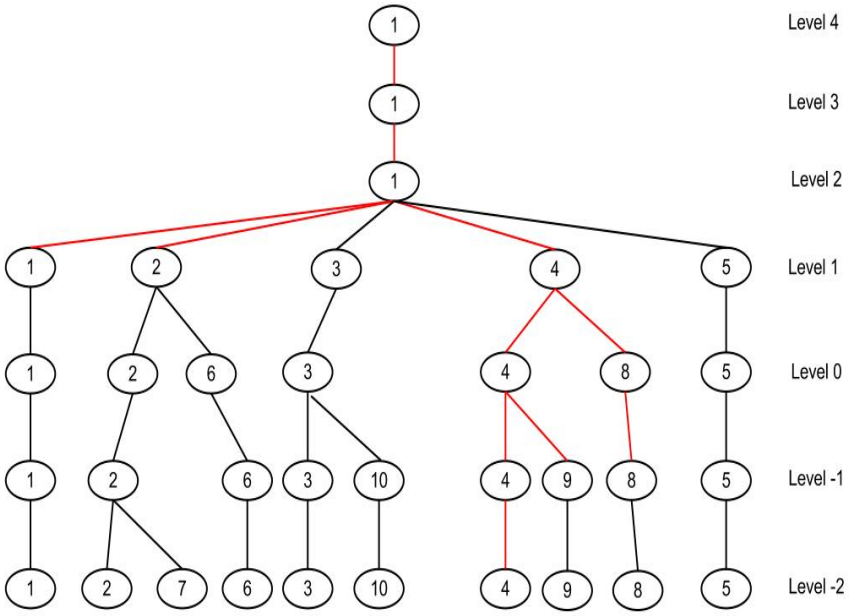


Figure 6.11: A Nearest Neighbor search for point $s1$ in the graph 6.6. The red lines indicate the nodes that were added to the different cover sets.

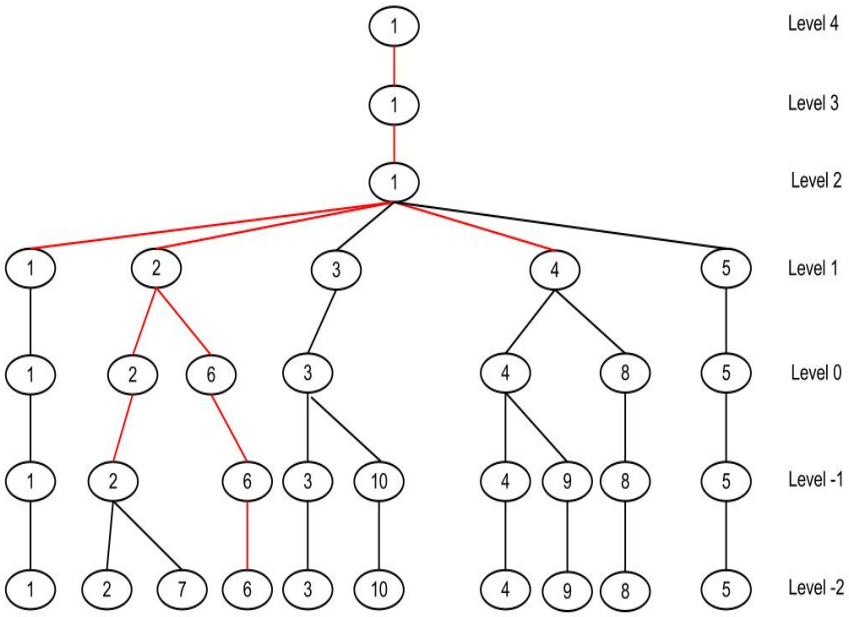


Figure 6.12: A Nearest Neighbor search for point $s2$ in the graph 6.6. The red lines indicate the nodes that were added to the different cover sets.

The search method returns the first Nearest Neighbor of a node. So if the query point p is already in the dataset, it will always be returned as the Nearest Neighbor. To find the next closest point, we need to do a 2-Nearest Neighbor

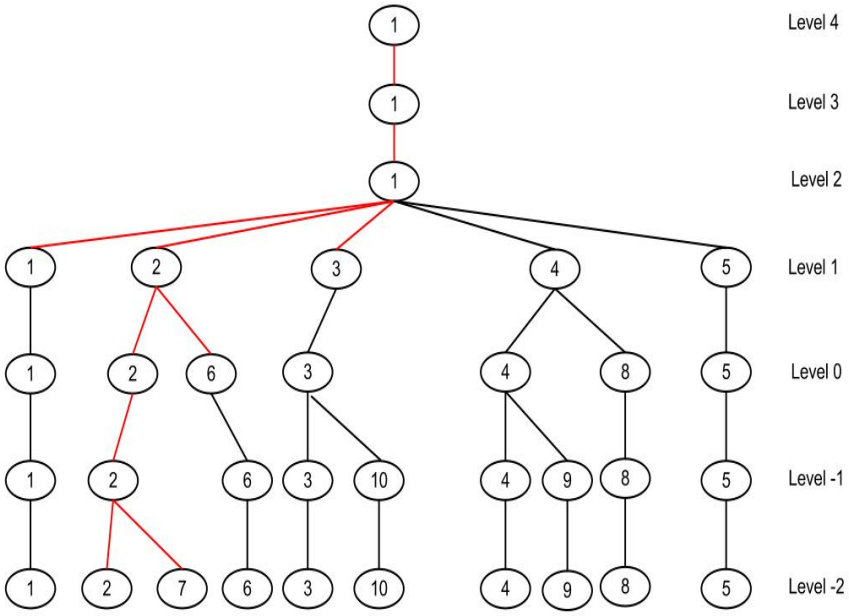


Figure 6.13: A Nearest Neighbor search for point $s3$ in the graph 6.6. The red lines indicate the nodes that were added to the different cover sets. Here there are two nodes that are present in the final cover set. In that case, step 6 of the algorithm finds the Nearest Neighbor by brute force.

search. The above algorithm can be easily modified to return the k-Nearest Neighbors of a node. We keep track of a set of k points, called the $minSet$ which are the k-closest neighbors as seen so far. So with that definition of the $minSet$, we can define $maxDist$ as follows:

$$maxDist = max_{x \in minSet} d(p, x).$$

Now, the algorithm can be modified as follows:

Algorithm 8: Algorithm to find the k-Nearest Neighbors in the Cover Tree

Input: Cover Tree T, Query Point p

Output: k-Nearest Neighbors of p

set $Q_\infty = C_\infty$
add root to minSet
maxDist = d(p, root)
for $i = \infty \rightarrow -\infty$ **do**
    consider the set of children of Q_i: $Q = Children(q) : q \in Q_i$
    form next cover set: $Q_{i-1} = q \in Q : d(p, q) \leq maxDist + 2^i$
    if $(q \in Q : d(p, q) < maxDist)$ **then**
        Update minset to add q
        Update maxDist
    end
end
return minset

Complexity The time complexity of the Search algorithm is $O(c^{12}logN)$. Refer to the paper by Beygelzimer et al. [1] for a mathematical proof of the complexity.

6.9 Inserting and Removing nodes from the Cover Tree

The algorithms for inserting and removing nodes Cover Tree are given in the Appendix A and B.

6.10 Cover Tree Based HAC

We replace the brute-force Nearest Neighbor search in the Nearest Neighbor based HAC by a cover Tree based Nearest Neighbor search. First we create the Cover Tree using all the points in the input dataset. Then the NN_Array is populated by executing a Nearest Neighbor search for all the points in the dataset. The closest neighbors for each iteration are merged into a new cluster and this new cluster is inserted into the Cover Tree. The two old clusters are removed from the Cover Tree. This process is repeated until all the iterations are completed. Algorithm 9 gives the pseudo code for this procedure.

Algorithm 9: Cover Tree Based HAC

Input: Data(N,D)

1 Construct the Cover Tree using the initial Data - Data(N,D).
2 Compute Nearest Neighbor Array NN_Array using the Cover Tree.
3 **for** $i = 1$ *to* $N - 1$ **do**
4 Find closest pair of clusters (p, q) from the NN_Array
5 Merge the closest pair of clusters
6 Add the newly created cluster into the Cover Tree.
7 Remove the two merged clusters from the Cover Tree.
8 Add a new entry in the NN_Array for the new cluster and find the Nearest Neighbor of the new cluster using the Cover Tree.
9 Update all the entries in the NN_Array which are affected by the removal of (p, q) by executing searches in the Cover Tree.
10 **end**

6.10.1 Implementation Details

We implemented the Cover Tree based HAC as a single threaded sequential algorithm using C++. Whenever there is enough space available, the algorithm is confined to one NUMA region. In case that NUMA region runs out

of memory, the algorithm allocates memory in the nearest NUMA region that has available memory. These NUMA aware memory allocations were done using the memory manager written by Patryk Kaminski [14].

6.11 Experimental Results

The Cover Tree based HAC and the naive nearest Neighbor based HAC were tested on multiple datasets that have been taken from the UCI database and the KDD 2004 championship. As explained in Section 4.2.3 (page 31), the experiments were run on an AMD Opteron machine with four NUMA-regions of eight processors each. Table 6.3 gives the sizes and the dimensions of each of these datasets. Since we dont store the dissimilarity matrix for the Cover Tree based HAC and the naive nearest Neighbor based HAC (they are stored data algorithms), we can cluster bigger datasets using these algorithms compared to the NUMA-aware HAC algorithm defined in Chapter 5.11 (page 56) and this can be seen from the sizes of the datasets given in Table 6.3.

We used centroid linkage as the linkage criteria in conjunction with Euclidean distance. Centroid linkage was chosen because it can be implemented as a stored data algorithm and so can be run on bigger datasets. Table 6.4 gives the comparison of the times taken by the two algorithms.

It can be observed that the Cover Tree based HAC does better than the naive search in most of the datasets while taking the same amount of space. In other datasets, it does no worse than the naive Nearest neighbor version. Table 6.5 gives the total number of distance calculations that have been done by the two algorithms for each of the datasets.

From Table 6.5 it can be seen that the Cover Tree based HAC does far fewer distance calculations than the naive version.

Dataset	Number of Rows	Dimensions
bio_test.data	139657	74
bio_train.data	145750	75
corel.data	37748	32
letter.data	19999	17
phy_test.data	99999	78
phy_train.data	49999	80
mnist.data	59999	785
covtype.data	581011	55

Table 6.3: Datasets used for the experiments on Cover Tree based HAC

The two algorithms were also executed using complete linkage as the linkage criteria and Euclidean distance as the distance metric. Two versions of the algorithm were implemented

- A *Stored Matrix Algorithm* using both a dissimilarity matrix and a Nearest Neighbor array.
- A *Stored Data Algorithm* using only a Nearest Neighbor array.

In the stored data version, there was no dissimilarity matrix and the distances between points were calculated when needed. The stored matrix version uses a dissimilarity matrix to store the distances between points and so it is faster. But it requires lot more space for storing the dissimilarity matrix. For bigger datasets, the store matrix version required a very large amount of space and the store data version required a very large amount of time. So smaller datasets which required a reasonable amount of time and space were chosen for this

Dataset	NNHAC-Time(s)	CoverHAC-Time(s)	Speedup
bio_test.data	15940	3697	4.31
bio_train.data	21965	5014	4.38
corel.data	607	187	3.25
letter.data	114	38	3
phy_test.data	6427	470	13.67
phy_train.data	1429	136	10.51
mnist.data	22488	12706	1.77
covtype.data	225368	5054	44.59

Table 6.4: Comparison of Time taken when using Cover Trees and when not using Cover Trees with the centroid linkage and Euclidean distance

experiment. Table 6.6 presents the results for the stored data version of the algorithms and Table 6.7 presents the results for the stored matrix version of the algorithms.

Once again it can be seen that the Cover Tree based HAC is faster when compared to the naive version for both the stored data and stored matrix versions. Similar times were seen for both average linkage and single linkage.

6.11.1 Impact of the Expansion Constant

We analyze why the Cover Tree based HAC performs better for some datasets and not so well for other datasets. As explained in the definition of the Cover Tree, the complexities of the Insert Algorithm A and the Search Algorithm 6.8.4 depend on the expansion constant c. Thus the Expansion con-

Dataset	NNHAC-Num	CoverHAC-Num
bio_test.data	42,288,088,582	3,104,167,770
bio_train.data	59,972,679,705	3,289,354,766
corel.data	2,759,940,387	190,824,750
letter.data	928,734,931	45,093,957
phy_test.data	22,828,425,799	248,463,832
phy_train.data	5,083,867,739	92,461,542
mnist.data	9,563,199,588	3,937,459,084
covtype.data	638,077,109,310	576,817,606

Table 6.5: Comparison of Distances Computed when using Cover Trees and when not using Cover Trees with the centroid linkage and Euclidean distance

stant plays a major role in the time taken by the Cover Tree based HAC. To illustrate this, we calculated the Expansion constant of few of the datasets mentioned in Section 6.11 (page 82). The algorithm for the Expansion Constant given in Section 5 (page 60) has a time complexity of $O(N^2 log(N))$. So we can run it only on smaller datasets. We take the prefix of every dataset consisting of 5000 points and run the Expansion Constant algorithm (Algorithm 6) on it. The algorithm enables us to find the worst-case expansion constant of every point in the dataset and the overall worst-case expansion constant of the whole dataset. Graph 6.14 gives the cumulative distribution of expansion constants across points for all the datasets. In datasets *Mnist and Enron*, the expansion constant of all the points is high. Consequently, the Cover Tree does not perform well in these datasets. But in the datasets *Covtype, bio_test, Phy_test and Phy_train*, only a few points have a high expansion constant and all the other points have small expansion constants. In such a case, the Cover Tree performs well. Thus the speedups obtained are not based on the worst-case expansion constant of a dataset. Instead they are based on

Dataset	NNHAC-Time(s)	CoverHAC-Time(s)	Speedup
bio_test.data	80,761	18,584	4.35
corel.data	3,519	999	3.52
covtype.data	790,630	62,664	12.62
letter.data	699	206	3.39
mnist.data	77,143	39,623	1.95
optdigits.tra	33	29	1.14
phy_test.data	49,518	3,961	12.50
phy_train.data	12,028	1,073	11.21

Table 6.6: Comparison of Time Taken when using Cover Trees and when not using Cover Trees with the Complete linkage and Euclidean distance - Stored Data Version

Dataset	NNHAC-Time(s)	CoverHAC-Time(s)	Speedup
bio_test.data	15,603	3715	4.2
corel.data	552	160	3.45
letter.data	99.9	30	3.33
mnist.data	23,009	12,110	1.9
phy_test.data	4,840	400	12.1
phy_train.data	1,332	120	11.1

Table 6.7: Comparison of Time Taken when using Cover Trees and when not using Cover Trees with the Complete linkage and Euclidean distance - Stored Matrix version

the average expansion constant of the whole dataset.

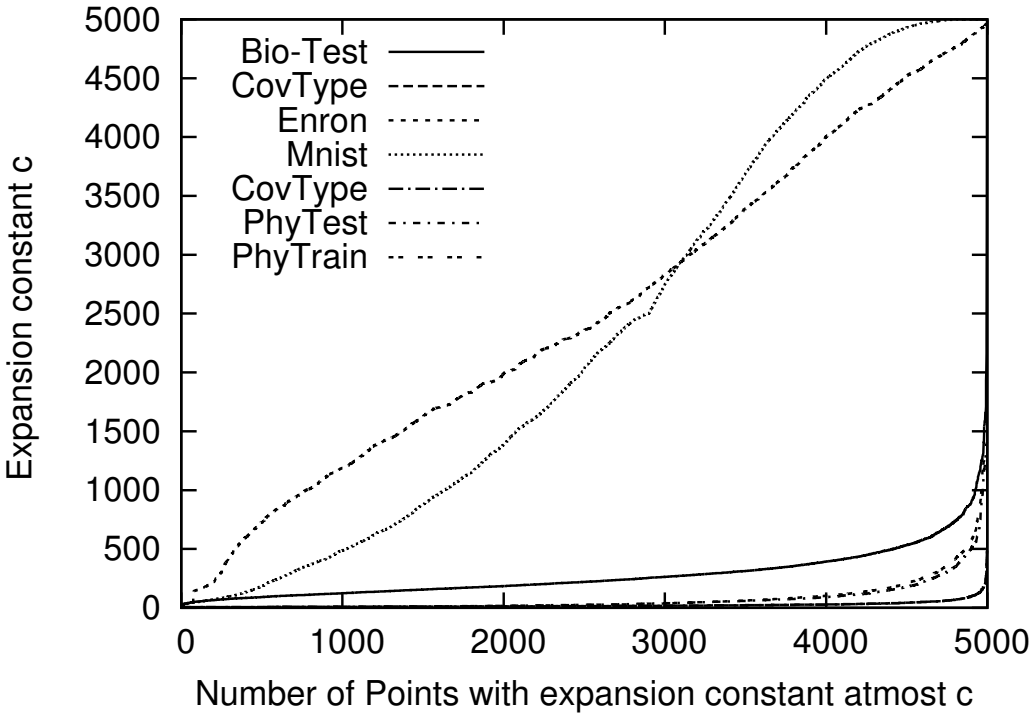


Figure 6.14: Distribution of the expansion constant c over the prefixes of the different datasets.

Chapter 7

Parallel Nearest Neighbor Based Hierarchical Clustering Algorithm

In this chapter we present parallel versions of the Nearest Neighbor based HAC algorithms that were discussed in the previous chapter (page 59). We first outline the parallel version of the Naive Nearest Neighbor based HAC algorithm and then present a parallel version of the Cover Tree based HAC algorithm. We then compare the performances of these two algorithms and also compare the performance of the Cover Tree based HAC Algorithm to the NUMA aware HAC algorithm that we presented in Chapter 5.11 (page 56).

7.1 Parallel Nearest Neighbor based HAC

In the Nearest Neighbor based HAC algorithm which was outlined in Section 6.2 (page 59), the biggest chunk of work is to find the Nearest Neighbor of the clusters. This part of the work readily lends itself to parallelization and so we have implemented a parallel version of this algorithm which consists of a master thread and a number of workers threads. Each of the worker

threads runs in a different processor and before the start of the algorithm, the clusters present are divided equally among the worker threads. Let set W_i be the subset of clusters allocated to worker thread W_i. Each worker thread also maintains a Nearest Neighbor array for the subset of the clusters that it owns. As explained in Section 6.2 (page 59), a Nearest Neighbor array is a N-tuple $P = (P_1, ..., P_N)$ that identifies for each cluster i, a nearest-neighboring cluster P_i: each P_i for $1 \leq i \leq N$ satisfies the condition that $d(i, P_i) = min(d(i, j) : 1 \leq j \leq N, i! = j)$ [23]. This array is populated before the start of the algorithm and is continuously updated during the execution of the algorithm. The pseudocode for the master thread of the parallel Nearest Neighbor based HAC algorithm is given by Algorithm 10 and the pseudocode for the worker threads of the same algorithm is given by Algorithm 11.

In each iteration, the Master thread merges the two closest clusters C_p and C_q to create a new cluster C_r. (Algorithm 10, line 6). The Master thread then removes the two merged clusters from the Nearest Neighbor arrays and point-subsets maintained by the worker threads. The next steps for the Master thread is to find the Nearest Neighbor of the newly created cluster and the Nearest Neighbor of any cluster that had the two merged clusters C_p and C_r as its Nearest Neighbor. (Algorithm 10, lines 9, 12). To find the Nearest Neighbor of any cluster C_i, the Master threads sends the information of cluster C_i to all the worker threads. Each worker thread finds the Nearest Neighbor of C_i among its local subset of clusters and sends this local Nearest Neighbor to the Master. The Master finds the global Nearest Neighbor of cluster C_i from the local results that the workers returned and sends them to the worker threads. Then the worker thread that owns the cluster C_i updates its Nearest Neighbor array with the updated value.

Algorithm 10: A Parallel version of the Nearest Neighbor based HAC - The Master Thread

Input: A set of N clusters to be clustered. Initially each cluster is its own cluster. Let the number of threads be P.

1 Allocate (N/P) clusters to each worker thread.
2 Start worker threads.
3 Signal worker threads to compute Nearest Neighbor Array NN_Array
4 Let $TotalNumberOfClusters$ = N
5 **while** $TotalNumberOfClusters > 1$ **do**
6 Find closest pair of clusters (C_p, C_q) from the NN_Array.
7 Merge the closest pair of clusters and create new merged cluster C_r.
8 Update the NN_Array of the workers to remove the cluster C_p and C_r.
9 Signal worker threads to find local Nearest Neighbor of C_r and wait for their completion.
10 Use the local Nearest Neighbor results to find the global Nearest Neighbor of C_r
11 Add C_r to one of the worker threads and signal workers to update the NN_Array to add C_r and its closest neighbor.
12 **foreach** *Cluster* $C_{affected}$ *that had either* C_p *or* C_q *as its Nearest Neighbor* **do**
13 Signal worker threads to find local Nearest Neighbor of $C_{affected}$ and wait for their completion.
14 Use the local Nearest Neighbor results to find the global Nearest Neighbor of $C_{affected}$.
15 Signal the worker thread that owns $C_{affected}$ to update the NN_Array to reflect the new Nearest Neighbor.
16 **end**
17 Output the pair of clusters that have been merged.
18 $TotalNumberOfClusters = TotalNumberOfClusters - 1$.
19 **end**

Algorithm 11: A Parallel version of the Nearest Neighbor based HAC - The Worker Thread W_i

```
1  Let Set W_i be the subset of clusters allocated to this worker thread.
2  Wait for signal from Master thread.
3  if Master thread signals to find local Nearest Neighbor of cluster C_i then
4      Do a brute force Nearest Neighbor search among the subset of
       clusters W_i allocated to this worker thread.
5      Send this local Nearest Neighbor C_LocalNearestNeighbor of cluster C_i to
       Master.
6  end
7  else if Master thread signals to update the NN_Array for Cluster C_i
   then
8      Get the global Nearest Neighbor C_GlobalNearestNeighbor of C_i from
       Master thread.
9      if Current worker thread has entry for C_i in its NN_Array then
10         Set the Nearest Neighbor of C_i to C_NearestNeighbor
11     end
12 end
13 else if Master thread signals to remove Cluster C_i from NN_Array
   then
14     Remove Cluster C_i from the NN_Array.
15 end
```

7.2 Parallel Cover Tree based HAC

In this section, we present a parallelized version of the Cover Tree based HAC algorithm. It is to be noted that the Cover trees themselves have not been parallelized and so the main idea of this algorithm is to have multiple Cover trees - one for each processor. The algorithm is very similar to the parallel Nearest Neighbor based HAC algorithm presented in the previous section and is composed of one master thread and a number of worker threads. Similar to the algorithm explained in the previous section, each of the worker threads runs in a different processor and initially the clusters present are divided equally among the worker threads. Let set W_i be the subset of clusters

allocated to worker thread W_i. Each worker thread also maintains a Cover tree $CoverTree_i$ in addition to the cluster subset W_i and the Nearest Neighbor array NN_Array. The $CoverTree_i$ for that worker thread is constructed from the subset of clusters W_i owned by that worker thread. The pseudocode for the master thread of the parallel Cover Tree based HAC algorithm is given by Algorithm 12 and the pseudocode for the worker threads of the same is given by Algorithm 13. It can be seen that line 5 in worker thread Algorithm 13, that we do a Cover Tree search to find the local Nearest Neighbor in the worker thread. This is the main difference between the parallel Nearest Neighbor based HAC algorithm outlined in the previous section and this algorithm. We also have to maintain the Cover Trees by removing the clusters that have been merged and by adding the clusters that have been created by the merging.

7.3 NUMA-aware optimizations

Both algorithms defined in Section 7.1 and Section 7.2 were implemented in C++. Similar to the optimizations that were outlined for the Cover Tree based HAC algorithm (explained in Section 6.10.1, page 81), we enabled NUMA-aware memory allocation for these two parallel algorithms. We use the *pthread* API to create the threads and the threads are forced to run on a specific NUMA region using the libnuma library [36]. To make sure that the memory allocated by the threads are in the same NUMA region that the thread is running, we used the memory manager written by Patrick Kaminski [14]. Thus, the data structures used by every thread (the cluster list, the Nearest Neighbor array and the Cover Tree) are present in the same NUMA region as the thread itself and so there are no cross-region memory accesses as long as that NUMA region has enough memory. If the NUMA region runs out of memory, then the memory manager will allocate memory from a neighboring NUMA region and there will be some cross-region memory access. But since these are stored data algorithms, this will happen only for huge datasets (none of the datasets

Algorithm 12: A Parallel version of the Cover Tree based HAC - The Master Thread

Input: A set of N clusters to be clustered. Initially each cluster is its own cluster. Let the number of threads be P.

```
1  Allocate (N/P) clusters to each worker thread.
2  Start worker threads.
3  Compute Nearest Neighbor Array NN_Array
4  Let TotalNumberOfClusters = N
5  while TotalNumberOfClusters > 1 do
6      Find closest pair of clusters (C_p, C_q) from the NN_Array.
7      Merge the closest pair of clusters and create new merged cluster C_r.
8      Signal worker threads to remove the clusters C_p and C_r from their
       Cover Trees.
9      Signal worker threads to find local Nearest Neighbor of C_r and wait
       for their completion.
10     Use the local Nearest Neighbor results to find the global Nearest
       Neighbor of C_r
11     Add C_r to one of the worker threads and update the NN_Array to
       add C_r and its closest neighbor.
12     foreach Cluster C_affected that had either C_p or C_q as its Nearest
       Neighbor do
13         Signal worker threads to find local Nearest Neighbor of C_affected
           and wait for their completion.
14         Use the local Nearest Neighbor results to find the global Nearest
           Neighbor of C_affected.
15         Signal worker thread that owns C_affected to update the
           NN_Array to reflect the new Nearest Neighbor.
16     end
17     Output the pair of clusters that have been merged.
18     TotalNumberOfClusters = TotalNumberOfClusters − 1.
19 end
```

that we used resulted in a NUMA region running out of memory).

Algorithm 13: A Parallel version of the Cover Tree based HAC - The Worker Thread W_i

```
1  Let Set W_i be the subset of clusters allocated to this worker thread.
2  Construct Cover tree CoverTree_i from the set of clusters W_i.
3  Wait for signal from Master thread.
4  if Master thread signals to find local Nearest Neighbor of cluster C_i then
5  |  Do a search in CoverTree_i to find the local Nearest Neighbor for
   |  point C_i among the subset of clusters W_i allocated to this worker
   |  thread.
6  |  Send this local Nearest Neighbor C_LocalNearestNeighbor of cluster C_i to
   |  Master.
7  end
8  else if Master thread signals to update the NN_Array for Cluster C_i
   then
9  |  Get the global Nearest Neighbor C_GlobalNearestNeighbor of C_i from
   |  Master thread.
10 |  if Current worker thread has entry for C_i in its NN_Array then
11 |  |  Set the Nearest Neighbor of C_i to C_NearestNeighbor
12 |  end
13 end
14 else if Master thread signals to delete cluster C_i from CoverTree_i then
15 |  Remove C_i from CoverTree_i.
16 end
17 else if Master thread signals to remove Cluster C_i from NN_Array
   then
18 |  Remove Cluster C_i from the NN_Array.
19 end
```

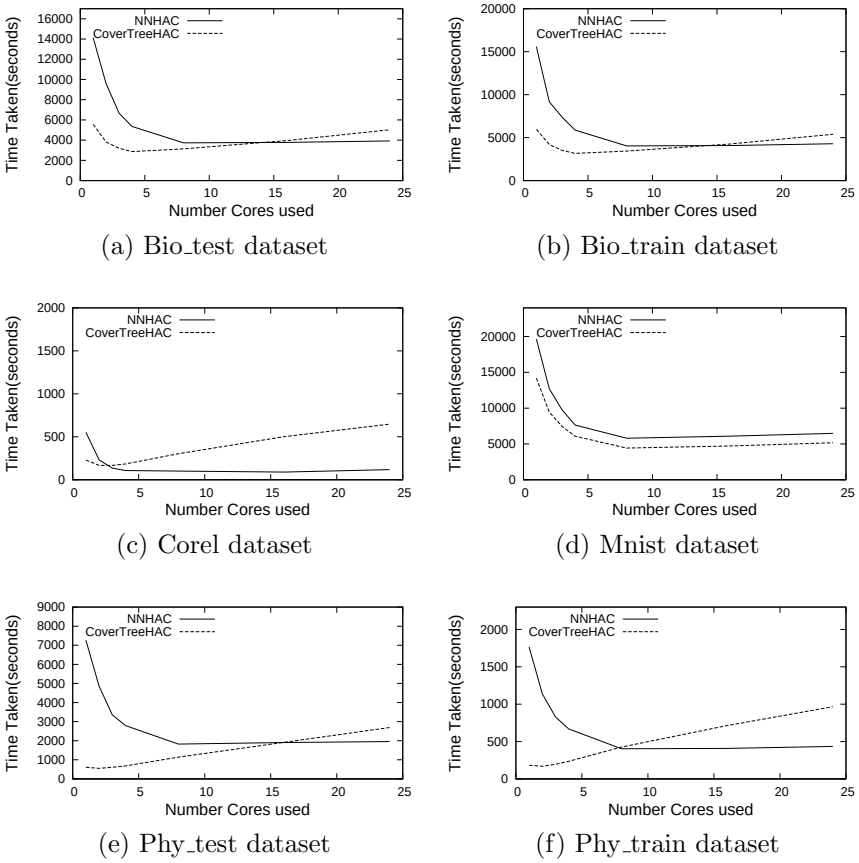
7.4 Experimental Results

Both parallel algorithms mentioned in this chapter were tested on the same datasets that the Nearest Neighbor based HAC algorithm and the Cover Tree based HAC algorithm were tested. For more information on the datasets used and their dimensions, please refer to Table 6.3 (page 83). As explained in Section 4.2.3 (page 31), the experiments were run on an AMD Opteron

machine with four NUMA-regions of eight processors each.

Graph 7.1 plots the time taken (in seconds) to the number of cores used for the Cover tree based HAC algorithm and the Nearest Neighbor based HAC algorithm for various datasets. It can seen from the graphs that for the Cover tree based HAC algorithm, the time taken first drops as the cores are increased, but beyond a point, the time taken increases. On the other hand, for the Nearest Neighbor based HAC algorithm, increasing the number of cores in very effective until about ten cores and after that, the time taken remains nearly constant. It is also interesting to note that for the Mnist dataset, the graphs of both these algorithms looks similar. As pointed out in Section 6.11.1 (page 84), the Mnist dataset has a high Expansion Constant for all the points and it was shown that for such datasets the sequential Cover tree based HAC algorithm performs similar to the sequential Nearest Neighbor based HAC algorithm and it does not have a big speedup. So, it might be that for datasets where these two algorithms perform similar in the sequential version, they also tend to perform in similar fashion in the parallel versions.

Graph 7.2 shows the number of distance computations made by both these algorithms. It can be seen that as the number of cores increases, the number of distance computations made by the Cover tree based HAC algorithm increases while for the Nearest Neighbor based HAC algorithm it remains constant. It should be noted that the number of distance computations made by the Cover tree based version is smaller than the Nearest Neighbor based version even when the number of cores are increased. This should mean that the Cover tree based version should take lesser time than the Nearest Neighbor based version even when the number of cores are increased. But as shown in graph 7.1, increasing the number of cores makes the Cover tree based version slower than the Nearest Neighbor based version except in the Mnist case. One of the reasons for this could be that the overhead involved in maintaining multiple Cover trees is very high and this leads to a degradation in performance even though the number of distance computations is still smaller than the Nearest



(a) Bio_test dataset (b) Bio_train dataset

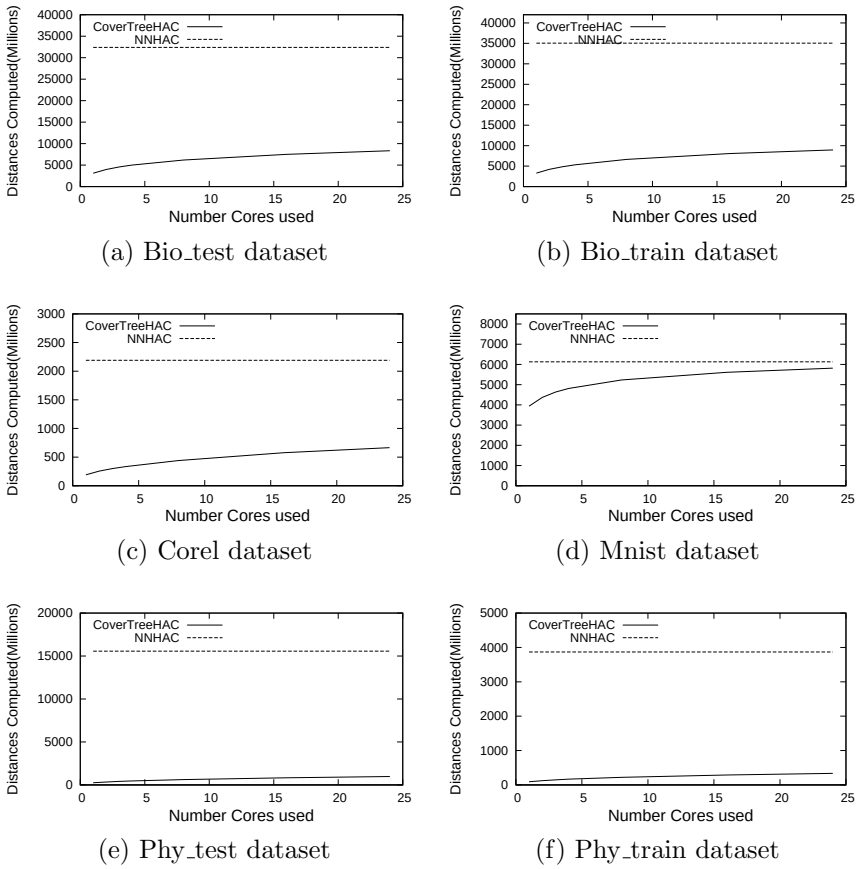
(c) Corel dataset (d) Mnist dataset

(e) Phy_test dataset (f) Phy_train dataset

Figure 7.1: Graph for the amount of time taken in seconds vs the number of cores used for both the CovertreeHAC and the NNHAC on various datasets.

Neighbor based version.

We also compared the Cover tree based HAC algorithm to the Uni directional handshake method of the NUMA aware HAC algorithm which was explained in Section 5.7.3 (page 46). The NUMA aware HAC algorithm is a stored matrix algorithm and as explained in Section 5.11 (page 54) it could not be run on datasets with more than 60,000 data points because of memory limitations and so for bigger datasets we used a prefix of the dataset that would fit into the available memory. Each algorithm's performance is optimum for a certain number of cores for a given dataset and so we have presented the results of both the algorithms on the optimal number of cores for that algorithm for that particular dataset (The optimal number of cores was found by



(a) Bio_test dataset (b) Bio_train dataset

(c) Corel dataset (d) Mnist dataset

(e) Phy_test dataset (f) Phy_train dataset

Figure 7.2: Graph for the number of distance computations made vs the number of cores used for both the CovertreeHAC and the NNHAC on various datasets.

running the algorithms with varying number of cores for each dataset and finding the number of cores which gives the best performance). We observed that the Cover tree based HAC algorithm was faster than the NUMA aware HAC algorithm in all the datasets. Graph 7.3 shows the speedup obtained by the Cover tree based HAC algorithm over the NUMA aware HAC algorithm. It can be seen from the Graph 7.3 that the Cover tree based HAC has a greater speed up on some datasets like *Covtype*, *phy_test* and *phy_train*, while on other datasets the speedup obtained is smaller. But it is always faster than the NUMA aware HAC in all the datasets tested.

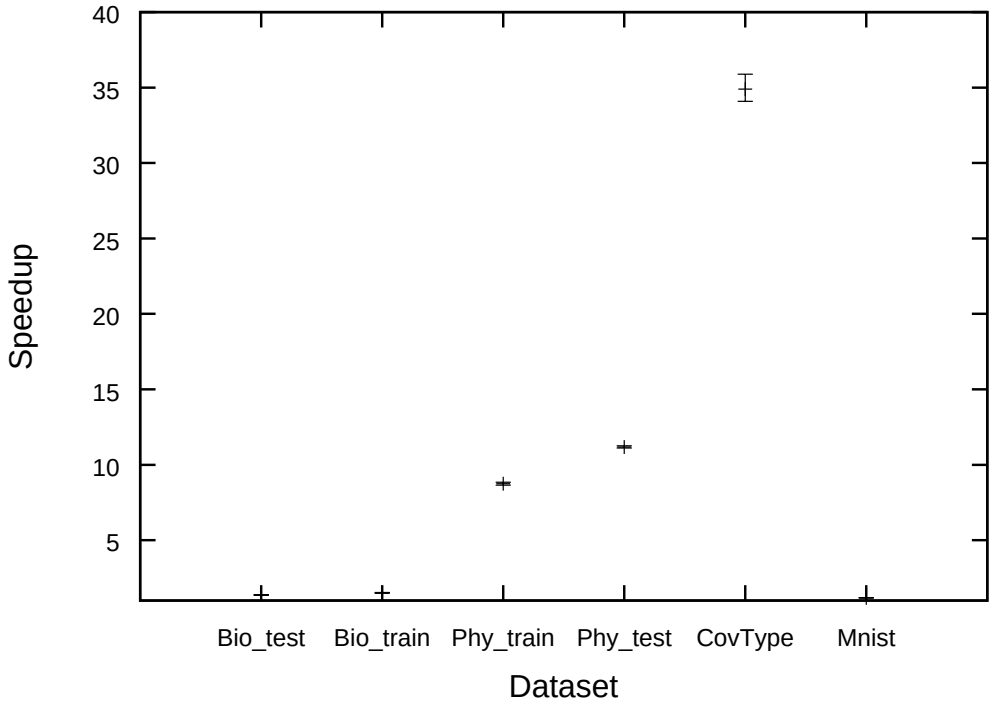


Figure 7.3: Speedup obtained by the Cover tree based HAC algorithm over the NUMA aware HAC algorithm (For bigger datasets, a prefix of the dataset that fits in the memory was used)

Chapter 8

Conclusion and Future Work

In this thesis, we have provided methods to speed up two different variations of HAC Algorithms. These new methods will increase the scalability of the traditional HAC algorithm without using any approximations. For each of the new solutions, we have contributed a survey, a new algorithm and an evaluation.

First we outlined the basic HAC algorithm and then outlined the different multi-processor architectures - Non-Uniform Memory Access Architectures (NUMA) and Uniform Memory Access Architectures (UMA). Then we have proposed a NUMA-aware HAC Algorithm which takes advantage of the specific structure of the Non-Uniform Memory Access architecture. The algorithm streamlines the flow of data among the nodes and reduces the bottle-neck in the Point-to-Point transport. We have conducted experiments that show that this method is faster than the naive HAC Algorithm. This algorithm stores the whole distance matrix in memory ($O(N^2)$ space requirement) and so is not suitable for big datasets.

So we outlined the Nearest Neighbor based HAC Algorithm which has a space requirement of $O(N)$ at the expense of the increased number of distance

computations. We presented the Cover Tree data structure and how it can be used to speed up the Nearest Neighbor Search. Then the cover-tree based HAC Algorithm was presented and the different optimizations that were done to make it faster were outlined. We have conducted experiments that show that the Cover Tree is faster for many datasets which intrinsically have a low dimensional structure. We have also outlined the Expansion Constant and presented an algorithm to find the Expansion Constant. Then we show how the average Expansion Constant affects the performance of the Cover Tree algorithm for the Nearest Neighbor search.

We also implemented a parallel version of the Cover Tree based HAC algorithm and compared it to a parallel version of the Nearest Neighbor based HAC algorithm and the NUMA aware HAC algorithm. In the parallel version of the Cover Tree based HAC, we have not parallelized the Cover Tree datastructure itself - instead we have used multiple Cover Trees - one for each core. We observed that the parallel version of the Cover Tree based HAC algorithm is slower than the Nearest Neighbor based HAC algorithm when the number of cores are increased which might be due to the overhead of maintaining multiple Cover Trees.

There are some areas of improvement in the algorithms that have been proposed in this thesis. First the NUMA-aware HAC Algorithm requires the whole distance matrix to be stored in memory. It would be worthwhile to devise an algorithm which does not need the whole distance matrix in memory (a stored data algorithm) while at the same time being NUMA-aware. Such an algorithm would make all data flow streamlined for the NUMA architecture and would be very practical on many datasets due to a reduced memory requirement.

There are also a number of improvements that can be made to the Cover Tree based HAC Algorithm. One area where research needs to be done is to parallelize the Cover Tree data structure itself. As explained above, in

the parallel Cover Tree based HAC that we have implemented (Section 7.2, page 91), we have not parallelized the Cover Tree datastructure itself and instead, we have used multiple Cover Trees- one for each thread. We have shown that this algorithm results in slower execution times than the sequential Cover Tree based HAC algorithm. It would be interesting to see if the Cover Tree data structure itself can be parallelized so that multiple threads can simultaneously use a single Cover Tree. One other improvement would be to implement the lazy construction presented by Beygelzimer et al. [1]. The lazy construction amortizes the construction cost into the query cost and hence might prove to be faster in practice.

Appendices

Appendix A

Insert Algorithm for Cover Tree

The insert algorithm (Algorithm 14) is similar to the Nearest Neighbor search algorithm. This algorithm starts at the root node of the tree $Q_\infty = C_\infty$. It recurses down the tree until it finds a level to put p such that all the three Covering Tree invariants hold (nesting, covering and separation).

Proof of Correctness: First we show that the algorithm completes. To prove this, we need to show that for each point, we enter the if statement in line 2. With each recursion, we are decreasing the level i. So the cover set of each point at that level becomes 2^i. If the dataset S is discrete, then for some i, the new point p will fall outside the cover set of all the other points. Since the if statement is invoked, there will be a level at which every point will be inserted.

We now show that, all the covering tree properties hold when the algorithm completes.

- Covering: Step 9 of the insert algorithm ensures that $d(p, Q_i) < 2^i$. So there exists at least one parent for p and we pick exactly one. This ensures the covering tree invariant.

Algorithm 14: Insert Algorithm for the Cover Tree

Input: Cover Set Q_i, Point p, Level i

1 set $Q = Children(q) : q \in Q_i$
2 **if** $d(p, Q) > 2^i$ **then**
3 | return 'Parent Found' - true.
4 **end**
5 **else**
6 | $Q_{i-1} = q \in Q : d(p, q) \leq 2^i$
7 | found = Insert(p, Q_{i-1}, i-1)
8 | **if** *found and* $d(p, Q_i) \leq 2^i$ **then**
9 | | pick a single $q \in Q$, such that $d(p, q) \leq 2^i$
10 | | insert p into Children(q);
11 | | return 'Finished' - false
12 | **end**
13 | **else**
14 | | return found
15 | **end**
16 **end**

- Nesting: When we insert p implicitly, we insert p into all the levels. Thus $C_i \subset C_{i-1}$.
- Separation: The if statement in step 2 of the insert algorithm ensures that all the points we added are separated.

Complexity The time complexity of the Insert algorithm is $O(c^6 logN)$. Refer to the paper by Beygelzimer et al. [1] for a mathematical proof of the complexity.

Appendix B

Remove Algorithm for Cover Tree

The remove algorithm (Algorithm 15) is similar to the insert algorithm, but in addition, it has to cope with the children of removed nodes.

Proof of Correctness: As in the insert, sets Q_i maintain points in level i closest to p as we descend through the tree. The recursion stops as soon as we reach a level below with p is always implicit.

For each level i explicitly containing p, we remove p from C_i and from the list of children of its parent in C_{i+1}. This does not affect the nesting and the separation invariants. For each child q of p other than itself, we go up the tree looking for a new parent. If we find a node $q' \in C_i$ such that $d(q, q') \leq 2^i$ we make q' a parent of q; Else, we insert q in level i and repeat, propagating this node up the tree until a parent is found. Insertion does not violate the nesting and separation constraints, since $d(q, C_i) > 2^i$. This process always terminates because the covering tree invariant is enforced for all children of p.

Complexity The time complexity of the Remove algorithm is $O(c^6 logN)$. Refer to the paper by Beygelzimer et al. [1] for a mathematical proof of the

Algorithm 15: Remove(p, cover sets $Q_{i-1}, Q_i, ...Q_\infty, level i$)

```
1  set Q = Children(q) : q ∈ Q_i
2  set Q_{i-1} = q ∈ Q : d(p,q) ≤ 2^i
3  Remove(T, p, Q_{i-1}, Q_i, ...Q_∞, i − 1)
4  if d(p,Q) = 0 then
5      remove p from C_{i-1}
6      Remove p from Children(Parent(p))
7      foreach q ∈ Children(p) do
8          set i' = i-1
9          while d(p,Q) => 2^{i'} do
10             insert q into C_{i'} and Q_{i'}
11             set i' = i'+ 1
12         end
13         pick q' ∈ Q_{i'} satisfying d(q,q') <= 2^{i'}
14         make q' point to q
15     end
16 end
```

complexity.

Bibliography

[1] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, International Conference on Machine Learning, pages 97–104, New York, NY, USA, 2006. Association of Computing Machinery.

[2] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd Association of Computing Machinery Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. Association of Computing Machinery.

[3] William H.E. Day and Herbert Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1(1):7–24, 1984. Springer-Verlag.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications, Association of Computing Machinery*, 51(1):107–113, January 2008. Association of Computing Machinery.

[5] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. Springer.

[6] James R Driscoll, Harold N Gabow, Ruth Shrairman, and Robert E Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the Association of Computing*

Machinery, 31(11):1343–1354, 1988. Association of Computing Machinery.

[7] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th Association of Computing Machinery International Symposium on High Performance Distributed Computing*, High Performance Distributed Computing '10, pages 810–818, New York, NY, USA, 2010. Association of Computing Machinery.

[8] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, Institute of Electrical and Electronics Engineers Transactions*, C-21(9):948–960, 1972. Institute of Electrical and Electronics Engineers.

[9] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *Association of Computing Machinery Transactions on Mathematical Software*, 3(3):209–226, september 1977. New York, NY, USA.

[10] H. Gao, J. Jiang, L. She, and Y. Fu. A new agglomerative hierarchical clustering algorithm implementation based on the map reduce framework. *International Journal of Digital Content Technology and its Applications*, 4(3):95–100, 2010. Elsevier Netherlands.

[11] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: An efficient clustering algorithm for large databases. *Association of Computing Machinery Special Interest Group on Management of Data*, 27(2):73–84, 1998. Association of Computing Machinery.

[12] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Rock: A robust clustering algorithm for categorical attributes. *Information Systems*, 25(5):345 – 366, 2000.

[13] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall Inc., Upper Saddle River, NJ, USA, 1988.

[14] Patryk Kaminski. Numa aware heap memory manager - white paper. `http://developer.amd.com/Assets/NUMA_aware_heap_memory_manager_article_final.pdf`, February 2009.

[15] David R. Karger and Matthias Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proceedings of the thiry-fourth annual Association of Computing Machinery symposium on Theory of computing*, pages 741–750, New York, NY, USA, 2002. Association of Computing Machinery.

[16] G. Karypis, Eui-Hong Han, and V. Kumar. Chameleon: hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999. Institute of Electrical and Electronics Engineers.

[17] Thomas Kollar. Fast nearest neighbors - white paper, 2006.

[18] Robert Krauthgamer and James R. Lee. Navigating nets: simple algorithms for proximity search. In *Proceedings of the fifteenth annual Association of Computing Machinery-SIAM symposium on Discrete algorithms*, SODA '04, pages 798–807, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[19] G. N. Lance and W. T. Williams. A general theory of classificatory sorting strategies: 1. hierarchical systems. *The Computer Journal*, 9(4):373–380, 1967. British Computer Society.

[20] Cen Li and G. Biswas. Unsupervised learning with mixed numeric and nominal data. *Institute of Electrical and Electronics Engineers Transactions on Knowledge and Data Engineering*, 14(4):673–690, 2002. Institute of Electrical and Electronics Engineers.

[21] Ramn A. Mollineda and Enrique Vidal. A relative approach to hierarchical clustering. In *Pattern Recognition and Applications, Frontiers in Artificial Intelligence and Applications*, pages 314–329. IOS Press, 2000.

[22] P. Mowforth and B. Shepherd. Vehicle silhouette dataset. Turing Institute, Glasgow, Scotland, Publicly disposable at UCI Repository Machine Learning: `http://www.ics.uci.edu/AI/ML/MLDBRepository.html`.

[23] Anderberg M.R. Cluster analysis for applications. Technical report, Defense Technical Information Center, 1973.

[24] C.F. Olson. Parallel algorithms for hierarchical clustering. *Parallel computing*, 21(8):1313–1325, 1995. Elsevier.

[25] R.C. Prim. Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6):1389–1401, 1957. Bell Labs.

[26] Edie M Rasmussen and Peter Willet. Efficiency of hierarchic agglomerative clustering using the icl distributed array processor. *Journal of Documentation*, 45(1):1–24, 1989. MCB UP Ltd.

[27] Robin Sibson. Slink: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973. The British Computer Society.

[28] Peter HA Sneath. The application of computers to taxonomy. *Journal of General Microbiology*, 17(1):201–226, 1957. Society for General Microbiology.

[29] Thorvald Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons. *Biologiske Skrifter. K. Danske videnskabernes Selskab*, 5:1–34, 1948.

[30] Jens Teubner and Rene Mueller. How soccer players would do stream joins. In *Proceedings of the 2011 Association of Computing Machinery Special Interest Group on Management of Data*, pages 625–636, New York, NY, USA, 2011. Association of Computing Machinery.

[31] Rui Xu and II Wunsch, D. Survey of clustering algorithms. *Institute of Electrical and Electronics Engineers Transactions on Neural Networks*, 16(3):645–678, 2005. Institute of Electrical and Electronics Engineers.

[32] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. *Association of Computing Machinery Special Interest Group on Management of Data*, 25(2):103–114, June 1996. Association of Computing Machinery.

[33] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. imapreduce: A distributed computing framework for iterative computation. *Journal of Grid Computing*, 10(1):47–68, 2012. Springer Netherlands.

[34] `http://dali.feld.cvut.cz/ucebna/matlab/toolbox/stats/dendrogram.html`.

[35] KDD Data Repository, `http://kdd.ics.uci.edu/`.

[36] Libnuma api. `http://linux.die.net/man/3/numa`.

[37] Power point on Cover Trees, `http://www.lems.brown.edu/vision/independentStudy/Voctoria_covertree.ppt`.

[38] SGI Systems Manual, `http://techpubs.sgi.com/library/manuals/5000/007-5683-003/pdf/007-5683-003.pdf`.

[39] The Mnist Dataset, `http://yann.lecun.com/exdb/mnist/`.

[40] UCI Data Repository, `http://www.ics.uci.edu/~mlearn/`.