3-8-2007

SIMD Algorithms for Single Link and Complete Link Pattern Clustering

Shankar Arumugavelu
University of South Florida

SIMD Algorithms for Single Link and Complete Link Pattern Clustering

by

Shankar Arumugavelu

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science & Engineering
College of Engineering
University of South Florida

Major Professor: Nagarajan Ranganathan, Ph.D.
Srinivas Katkoori, Ph.D.
Soontae Kim, Ph.D.

Date of Approval:
March 8, 2007

Keywords: Hierarchical clustering, pattern recognition, parallel algorithms, pattern matrix, proximity matrix

Table of Contents

List of Tables

List of Figures

SIMD Algorithms for Single Link and Complete Link Pattern Clustering

Shankar Arumugavelu

ABSTRACT

Clustering techniques play an important role in exploratory pattern analysis, unsupervised pattern recognition and image segmentation applications. Clustering algorithms are computationally intensive in nature. This thesis proposes new parallel algorithms for Single Link and Complete Link hierarchical clustering. The parallel algorithms have been mapped on a SIMD machine model with a linear interconnection network. The model consists of a linear array of *N* (number of patterns to be clustered) processing elements (PEs), interfaced to a host machine and the interconnection network provides inter-PE and PE-to-host/host-to-PE communication. For single link clustering, each PE maintains a sorted list of its first *logN* nearest neighbors and the host maintains a heap of the root elements of all the PEs. The determination of the smallest entry in the distance matrix and update of the distance matrix is achieved in *O(logN)* time. In the case of complete link clustering, each PE maintains a heap data structure of the inter pattern distances. This significantly reduces the computation time for the determination of the smallest entry in the distance matrix during each iteration, from $O(N^2)$ to *O(N)*, as the root element in each PE gives its nearest neighbor. The proposed algorithms are faster and simpler than previously known algorithms for hierarchical clustering. For clustering

a data set with N patterns, using N PEs, the computation time for the single link clustering algorithm is shown to be $O(NlogN)$ and the time complexity for the complete link clustering algorithm is shown to be $O(N^2)$. The parallel algorithms have been verified through simulations on the Intel iPSC/2 parallel machine.

Chapter One

Introduction

Clustering is the process of classifying objects into subgroups based on certain similarity criteria. The criterion is chosen based on the particular application. Cluster analysis is widely used in many fields such as life sciences, behavioral and social sciences, remote sensing, geography, medicine, information sciences and in engineering applications including exploratory pattern analysis, image segmentation, speech recognition etc. where the goal is to find natural groupings within a given data set [1] [5] [8] [17].

Clustering algorithms are aimed at finding structure in the data. They can be broadly classified as *supervised* and *unsupervised*. In supervised clustering, some form of category labels based on *a priori* partition of the objects is used; whereas, in unsupervised clustering, the proximity matrix is the only input. Further, the unsupervised clustering methods can be sub-divided into two types depending on the resulting structure of the data: *partitional* and *hierarchical*. The partitional clustering methods divide the objects into several clusters according to the selected criteria resulting in a single partition. In hierarchical clustering, a nested sequence of partitions is created [5] [8] [20]. The set of objects which constitute the input to the clustering problem can be best described by two formats: a *pattern matrix* and a *proximity matrix*.

Hierarchical clustering algorithms are very popular because they provide a pictorial representation of the data, known as *dendrograms* which can easily be interpreted by cluster analysts. Dendrograms list the clusterings one after another and cutting a dendrogram at any level provides a clustering and identifies the clusters.

Pattern Matrix

If each object in a set of *N* objects is represented by a set of *M* measurements or features, then each object is said to be represented by a *pattern*. The whole set of such patterns is viewed as an *N* x *M* pattern matrix. The pattern matrix is denoted by $[x_{ij}]$, where x_{ij} denotes the j^{th} feature for the i^{th} pattern. Each row of this matrix defines a pattern and each column denotes a feature. So, the i^{th} pattern, which is i^{th} row of the pattern matrix, can be denoted by the column vector x_i.

$$x_i = (x_{i1}, x_{i2}, \ldots\ldots x_{im})^\top, \; i = 1, 2, \ldots\ldots, n$$

where *m* is the number of features, *n* is the number of patterns and $^\top$ denotes vector transpose. For example, when clustering students in a class, each row in the pattern matrix would represent a student and the features in the pattern matrix could represent the scores in the different subjects, provided the same exams have been administered to all the students in a particular experiment.

Proximity Index

A proximity index refers to either similarity or dissimilarity. A closer resemblance between two objects is indicated by a larger similarity index or a smaller

dissimilarity index. The proximity index between the i^{th} and k^{th} objects is represented by *d(i, k)* and must satisfy the following three properties:

1. (a) For dissimilarity: $d(i, i) = 0$, for all i

   (b) For similarity: $d(i, i) \geq max_k\, d(i, k)$, for all i

2. $d(i, k) = d(k, i)$, for all (i, k)

3. $d(i, k) \geq 0$, for all (i, k)

The most common proximity index is the Minkowski metric, which measures dissimilarity [8]. If m represents the number of features, the Minkowski metric is defined by

$$d(i, k) = \left[\sum_{j=1}^{m} | x_{ij} - x_{kj} |^{r} \right]^{1/r} \qquad where\ r \geq 1$$

Euclidean distance is the most common of the Minkowski metrics. For the Euclidean distance, the value of r is 2 and can therefore be written as

$$d(i, k) = \left[\sum_{j=1}^{m} | x_{ij} - x_{kj} |^{2} \right]^{(1/2)}$$

Proximity Matrix

A proximity matrix is an N x N matrix, where N is the number of objects, which accumulates the pairwise indices of proximity. Each row and column of this matrix represents a pattern. The proximity matrix is denoted by $[d(i, k)]$, where $d(i, k)$ stands for the proximity index between the i^{th} and k^{th} objects determined by using the Minkowski metric described earlier. Also, all proximity matrices are symmetric, so all pairs of

objects have the same proximity index, independent of the order in which they are written. The diagonal entries of the proximity matrix are ignored since all patterns are assumed to have the same degree of proximity with itself.

Hierarchical Clustering

Hierarchical clustering is a technique by which we can obtain a sequence of partitions in which each partition is nested into the next partition in the sequence. It can be broadly classified into two categories: *agglomerative* and *divisive*. The agglomerative algorithm for hierarchical clustering starts by placing each of the objects in the data set in an individual cluster and then gradually merges those individual clusters. The divisive algorithm however, starts with the whole data set as a single cluster and then breaks it down into fewer clusters. *Single Link* and *Complete Link* are two hierarchical agglomerative clustering procedures.

In the Single Link clustering algorithm, clusters are merged at each stage by the single shortest link between them. During each iteration, after the clusters x and y are merged, the distance between the new cluster, say n, and some other cluster, say z, is given by $d_{nz} = min(d_{xz}, d_{yz})$, where d_{nz} denotes the distance between the two closest members of clusters n and z. If the clusters n and z were to be merged, then for any object in the resulting cluster, the distance to its nearest neighbor would be at most d_{nz}.

In the Complete Link clustering algorithm, at each stage, after the clusters x and y are merged, the distance between the new cluster, say n, and some other cluster, say z, is given by $d_{nz} = max(d_{xz}, d_{yz})$, where d_{nz} denotes the distance between the most distant

members of clusters n and z. If n and z were to be merged, then every object in the resulting cluster would be no farther than d_{nz} from every other object in the cluster. All objects in the cluster are thus linked to each other at some maximum distance.

Need for a Parallel Algorithm

Clustering algorithms are computationally intensive in nature. The single link and complete link clustering algorithms exhibit inherent parallelism because of the locality of computations involved. Both these algorithms proceed by determining the smallest entry in the proximity matrix, merging the two clusters associated with it, and updating the whole matrix so that all other patterns reflect the new distances between themselves and the newly merged cluster. This process is repeated N times where N is the number of patterns. As the proximity matrix is a symmetric matrix, the only information that has to be updated in each row of the matrix would be the distance with each of the cluster indices which were merged recently. In the case of single link clustering, each row will retain the minimum of the two values whereas in the case of complete link clustering it will retain the maximum of the two values. It can thus be seen that a fair amount of the processing involved in these algorithms are confined to the rows of the matrix. Also, the sequential algorithms for single link and complete link clustering have a time complexity of $O(N^3)$ and therefore take an excessive amount of time even to cluster moderately sized data sets. The design of efficient parallel algorithms and their implementation on various parallel computational models is of research interest and would be beneficial towards speeding up the clustering process.

Contributions of this Thesis

This thesis proposes efficient parallel algorithms for Single Link and Complete Link clustering techniques, based on a linearly interconnected SIMD parallel computational model. The model consists of a linear array of N processing elements (PEs), where N is the number patterns to be clustered, interfaced to a host machine. By storing only the absolutely required inter-pattern distances and using efficient data structures to store them, the computation time to determine the smallest entry in the distance matrix and the time taken to update the distance matrix is reduced significantly. The algorithms have been verified through simulations on the Intel iPSC/2 parallel machine. It is shown that the proposed algorithms provide considerable speed up over the existing parallel methods in the literature.

Thesis Outline

This thesis is organized as follows. Chapter 2 discusses Single Link and Complete Link hierarchical agglomerative clustering algorithms. A brief literature survey is given in Chapter 3. Chapter 4 describes the proposed parallel algorithms for Single Link and Complete Link clustering. Simulation results and performance comparisons with previous approaches is presented in Chapter 5. Chapter 6 presents the conclusion.

Chapter Two

Hierarchical Agglomerative Clustering

Johnson [9] proposed the first sequential algorithm to solve the hierarchical clustering problem. The algorithm proceeds by determining the smallest entry in the distance matrix during each iteration, merging the two clusters that are separated by that distance and update/delete the rows of the distance matrix according to the selection criteria (*min* for single link and *max* for complete link). This chapter describes the sequential single link and complete link clustering algorithms with an example.

Single Link Clustering

In the single link clustering algorithm, the clusters are merged at each stage by the single shortest link between them. The distance between the new cluster and some other cluster is determined by the distance between the two closest members of the two clusters. The single link clustering algorithm can be described as follows:

Step 1. Construct the distance matrix from the given pattern matrix

Step 2. Assign each pattern to a cluster

Step 3. Determine the smallest entry in the distance matrix D, say $D(c_1, c_2)$ and merge the two clusters c_1 and c_2

Step 4. Update the distance matrix D, by deleting the row and column corresponding to

the cluster c_2, and rename row c_1 and column c_1 to (c_1, c_2).

Assign the $min[D(c_3, c_1), D(c_3, c_2)]$ to $D(c_3, (c_1, c_2))$ and $D((c_1, c_2), c_3)$ for all c_3's

Step 5. If only one cluster is left, stop. Else, go to *Step 3*

Single Link clusters are thus characterized as maximally connected subgraphs as only a single edge between two large clusters is needed to merge the clusters. Figure 1 shows an example of the single link clustering algorithm for a data set of 8 patterns. All seven iterations are shown and the smallest entry of the distance matrix that merges two clusters in each iteration is highlighted. The resulting dendrogram which gives a pictorial representation of the clusters being formed at each level is given in Figure 2.

Complete Link Clustering

In the Complete Link clustering algorithm, at each stage, after the clusters x and y are merged, the distance between the new cluster, say n, and some other cluster, say z, is given by $d_{nz} = max(d_{xz}, d_{yz})$, where d_{nz} denotes the distance between the most distant members of clusters n and z. If n and z were to be merged, then every object in the resulting cluster would be no farther than d_{nz} from every other object in the cluster. All objects in the cluster are thus linked to each other at some maximum distance. The complete link clustering algorithm can be described as follows:

Step 1. Construct the distance matrix from the given pattern matrix

Step 2. Assign each pattern to a cluster

Step 3. Determine the smallest entry in the distance matrix D, say $D(c_1, c_2)$ and merge the two clusters c_1 and c_2

Step 4. Update the distance matrix *D*, by deleting the row and column corresponding to the cluster c_2, and rename row c_1 and column c_1 to (c_1, c_2). Assign the $max[D(c_3, c_1), D(c_3, c_2)]$ to $D(c_3, (c_1, c_2))$ and $D((c_1, c_2), c_3)$ for all c_3's

Step 5. If only one cluster is left, stop. Else, go to *Step 3*

Complete Link clusters are characterized as maximally complete subgraphs. They are conservative in such that all pairs of objects must be related before the objects can form a complete link cluster. An example of the complete link clustering algorithm for the same data set of 8 patterns is shown in Figure 3. Figure 4 represents the resulting dendrogram.

Iteration 1:

	1	2	3	4	5	6	7	8
1		117	14	40	50	45	32	51
2			143	185	259	234	129	146
3				22	28	29	20	33
4					10	7	24	12
5						(5)	42	37
6							35	32
7								13
8								

Iteration 2:

	1	2	3	4	(5, 6)	7	8
1		117	14	40	45	32	51
2			143	185	234	129	146
3				22	28	20	33
4					(7)	24	12
(5, 6)						35	32
7							13
8							

Iteration 3:

	1	2	3	(4, 5, 6)	7	8
1		117	14	40	32	51
2			143	185	129	146
3				22	20	33
(4, 5, 6)					24	(12)
7						13
8						

Iteration 4:

	1	2	3	(4, 5, 6, 8)	7
1		117	14	40	32
2			143	146	129
3				22	20
(4, 5, 6, 8)					(13)
7					

Figure 1. Trace of Single Link Clustering on 8 Patterns

Iteration 5:

	1	2	3	(4, 5, 6, 7, 8)
1		117	14	32
2			143	129
3				20
(4, 5, 6, 7, 8)				

Iteration 6:

	(1, 3)	2	(4, 5, 6, 7, 8)
(1, 3)		117	20
2			129
(4, 5, 6, 7, 8)			

Iteration 7:

	(1, 3, 4, 5, 6, 7, 8)	2
(1, 3, 4, 5, 6, 7, 8)		117
2		

Figure 1. (Continued)

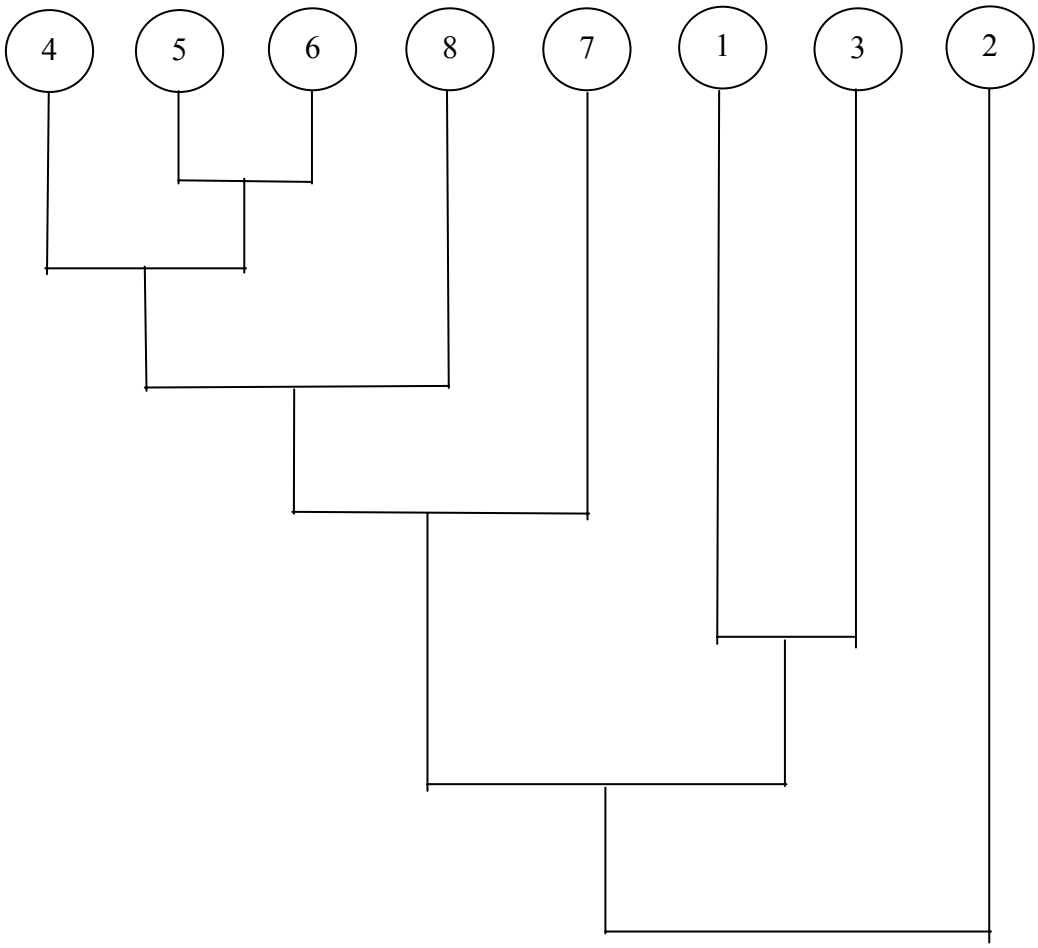

Figure 2. Result Dendrogram of Single Link Clustering on 8 Patterns

Iteration 1:

	1	2	3	4	5	6	7	8
1		117	14	40	50	45	32	51
2			143	185	259	234	129	146
3				22	28	29	20	33
4					10	7	24	12
5						5	42	37
6							35	32
7								13
8								

Iteration 2:

	1	2	3	4	(5, 6)	7	8
1		117	14	40	50	32	51
2			143	185	259	129	146
3				22	29	20	33
4					10	24	12
(5, 6)						42	37
7							13
8							

Iteration 3:

	1	2	3	(4, 5, 6)	7	8
1		117	14	50	32	51
2			143	259	129	146
3				29	20	33
(4, 5, 6)					42	37
7						13
8						

Iteration 4:

	1	2	3	(4, 5, 6)	(7, 8)
1		117	14	50	51
2			143	259	146
3				29	33
(4, 5, 6)					42
(7, 8)					

Figure 3. Trace of Complete Link Clustering on 8 Patterns

Iteration 5:

	(1, 3)	2	(4, 5, 6)	(7, 8)
(1, 3)		143	50	51
2			259	146
(4, 5, 6)				42
(7, 8)				

Iteration 6:

	(1, 3)	2	(4, 5, 6, 7, 8)
(1, 3)		143	51
2			259
(4, 5, 6, 7, 8)			

Iteration 7:

	(1, 3, 4, 5, 6, 7, 8)	2
(1, 3, 4, 5, 6, 7, 8))		259
2		

Figure 3. (Continued)

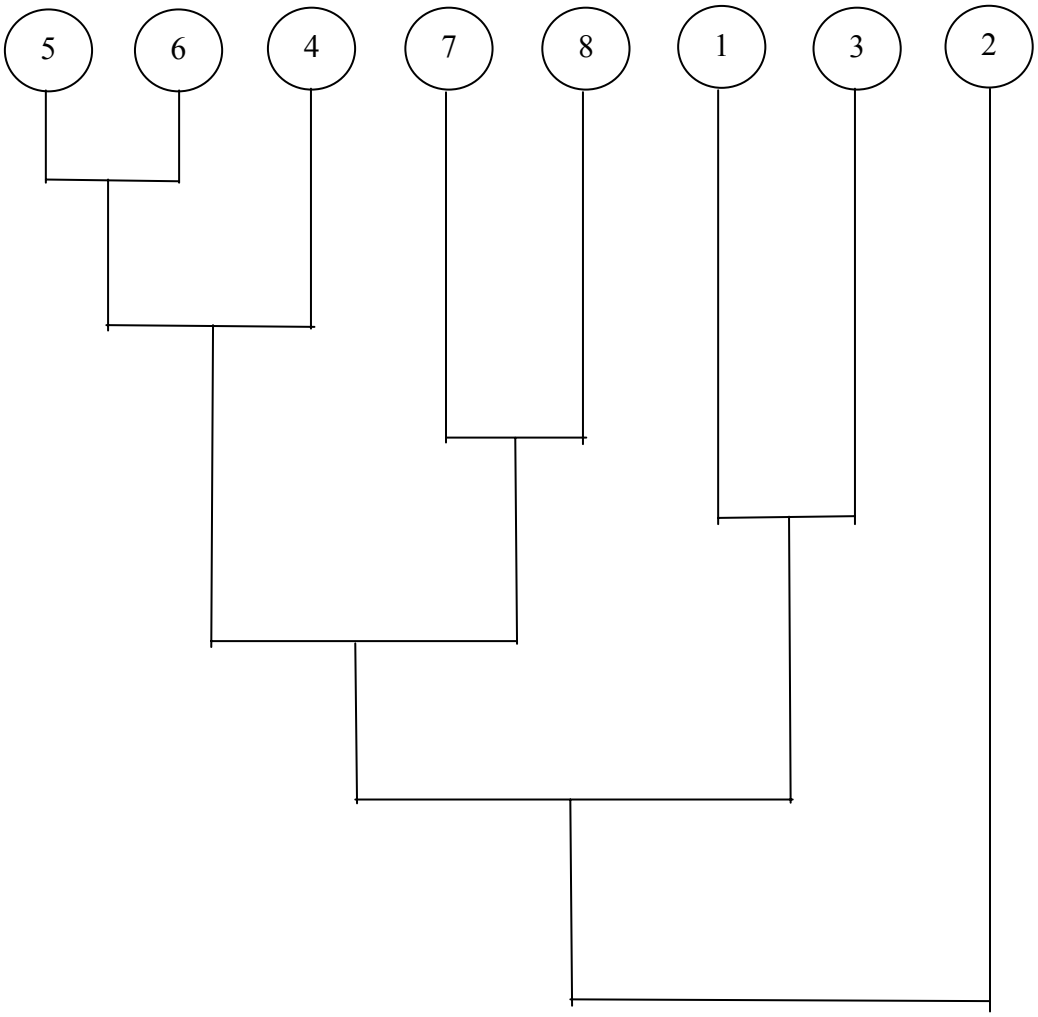

Figure 4. Result Dendrogram of Complete Link Clustering on 8 Patterns

Chapter Three

Related Work

The importance of pattern clustering is evidenced by the existence of a large number of sequential and parallel algorithms in the literature. Several sequential and parallel clustering algorithms have been proposed to speed up the clustering process. Johnson [9] proposed the first sequential algorithm to solve the hierarchical clustering problem. The time complexity of this algorithm is $O(N^2M + N^3)$ including the distance matrix computation (N is the number of patterns and M is the number of features). Hattori and Torii [6] presented two effective algorithms for the nearest neighbor method in hierarchical agglomerative clustering.

Many attempts have been made in the recent years to devise parallel algorithms and also develop special purpose hardware chips to solve the clustering problem. A two-level pipelined VLSI systolic array for Squared Error partitional clustering was proposed by Ni and Jain [13]. Sahni and Ranka [16] proposed efficient parallel algorithms for Squared Error partitional clustering on a SIMD machine model with NM PEs (N being the number of patterns and M being the number of features) interconnected using a hypercube interconnection network. Olson [14] has presented $O(nlogn)$-time $n/logn$-processor algorithms on the Parallel Random Access Machine (PRAM), butterfly, and tree models. Tsai et al. [18] have proposed an $O(log^2n)$-time algorithm that uses

n^3-processor array with a reconfigurable bus system (PARBS) processors. Wu et al. [19] have presented an *O(logn)*-cycles algorithm that uses n^3 processors on the Arrays with Reconfigurable Optical Buses (AROB) model. Rajasekaran [15] presents a PRAM algorithm that runs in *O(logn)*-time using $n^2/logn$ Concurrent-Read-Concurrent-Write (CRCW) PRAM processors and an AROB algorithm that runs in $O(log^2 n)$-cycles using n^2 processors. A taxonomy diagram of related work in the area of hierarchical clustering is shown in Figure 5. The two parallel algorithms of direct relevance to this work are [11] and [12].

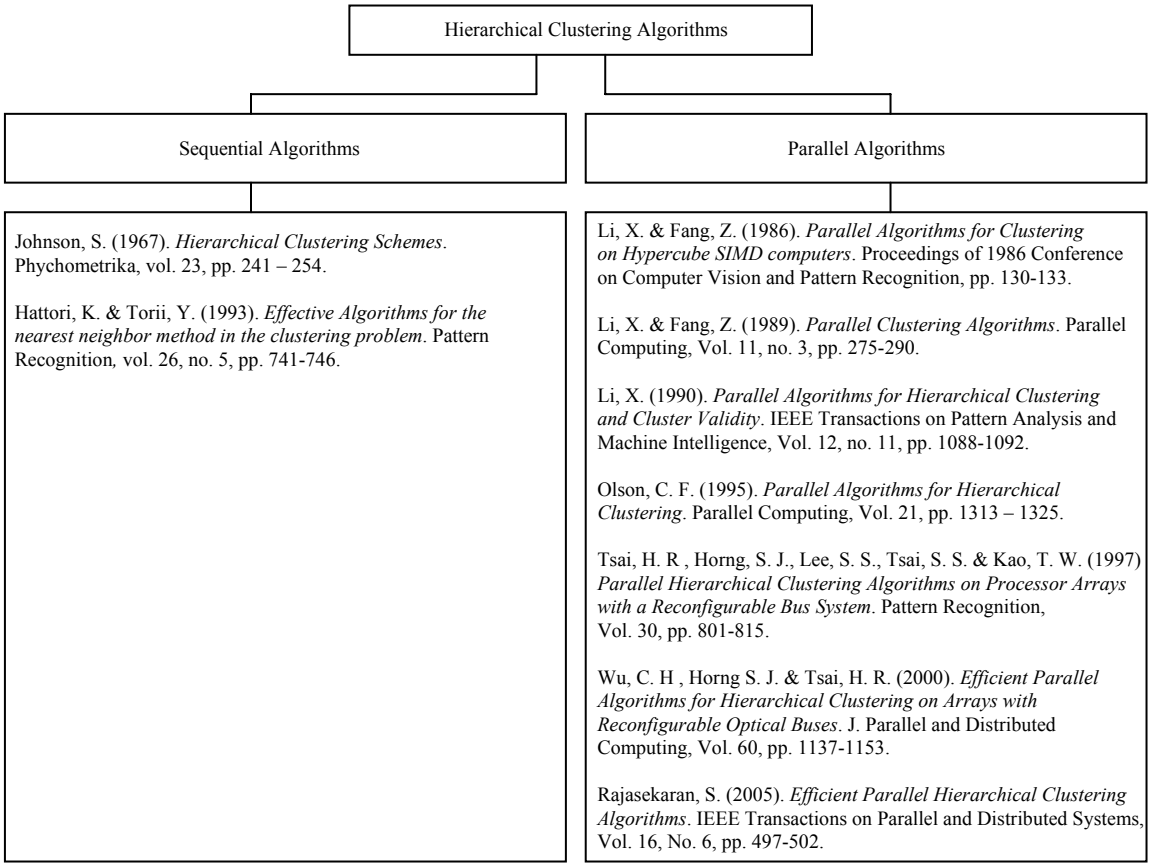

Figure 5. Taxonomy Diagram of Related Work

SIMD Hypercube Computational Model

Li and Fang [11] proposed a parallel algorithm for the single link hierarchical clustering problem, on a SIMD hypercube computer with *NM* processors for *N* patterns and *M* features. The PEs are arranged as a 2-dimensional array. Each PE had an *O(N)* memory and the nearest neighbor distance for each pattern was computed and stored before beginning the clustering. This way, the process of finding the global minimum entry of the distance matrix during each iteration involved only *N* entries instead of N^2 entries. During the course of the iteration, only the cluster which is affected has to update its nearest neighbor distance. The parallel algorithm proceeds as follows:

Step 1. Spread the pattern matrix to all the PEs so that each PE holds the corresponding feature values of all the patterns.

Step 2. Compute the inter-pattern distances (distance matrix *D*) and compare these distances in parallel to find the nearest neighbor of each pattern.

Step 3. Repeat the following N times

a. Find the minimum of the nearest neighbors for each pattern
b. Output the cluster pair and update the distance matrix *D*
c. Update the local minimum for the cluster which is retained of the two that were clustered in *Step 3a*

In the above algorithm, *Step 1* takes *O(NM)* time. Using *NM* PEs and the hypercube interconnection network, *Step 2* is achieved in *O(NlogM)* time. *Step 3a* takes *O(logN)* time. Steps *3b* and *3c* take constant time respectively. *Step 3* as a whole is thus an *O(NlogN)* procedure.

The overall time complexity of this parallel algorithm including the distance matrix computation is *O(NM + NlogM + NlogN)*.

SIMD Shuffle-Exchange Computational Model

Li [12] proposed parallel algorithms for single and complete link hierarchical clustering on a SIMD machine model with a shuffle-exchange network interconnecting an array of *N* processing elements and a shared parallel memory system. The interconnection network is used for both memory-PE communication and PE-PE communication. The distance matrix is stored in *N* memory modules in such a way that all the elements in each row of the distance matrix are in different memory modules. The algorithm thus permits parallel accessibility of the distance matrix. As the distance matrix is symmetric, the algorithms modify only the upper right triangle of the distance matrix. The distance matrix update is done in *O(logN)* time. To compute the minimum among the elements in the distance matrix, an *O(NlogN)* procedure is proposed using the shuffle-exchange network. The parallel algorithm for single link hierarchical clustering proceeds as follows:

Step 1. Determine the smallest entry in the distance matrix. Let the clusters associated with that entry be x_1 and x_2.

Step 2. Repeat the following *N-2* times:

a. Update the distance matrix after merging the two clusters x_1 and x_2
b. Find the new nearest neighbor for cluster x_1
c. Find the global minimum

Step 3. Merge the two clusters left behind.

Step 1 is achieved by first iterating across N patterns and finding the nearest neighbor of each pattern and then determining the minimum of the N nearest neighbor distances. This is an *O(NlogN)* procedure. *Step 2a* is performed to eliminate the cluster x_2 and update the distance matrix with the new distances corresponding to cluster x_1. This is achieved in *O(logN)* time. *Step 2b* fetches the row corresponding to cluster x_1 and determines the minimum among the N elements in the row. This is an *O(logN)* procedure. *Step 2c* and *Step 3* each take an *O(logN)* time. The overall time complexity of the single link parallel algorithm is *O(NlogN)*.

In the case of complete link clustering, the nearest neighbors for each cluster cannot be stored before beginning the clustering because the distance between an existing cluster and a newly created cluster can get large. The parallel algorithm for complete link hierarchical clustering is as follows:

Step 1. Repeat the following $N - 1$ times:

a. Determine the smallest entry in the distance matrix. Let the associated cluster pair be x_1, x_2
b. Eliminate the cluster x_2 and update the distance matrix with the new distances corresponding to the cluster x_1

Steps *1a* and *1b* are *O(NlogN)* and *O(logN)* procedures respectively.

The overall time complexity of the complete link parallel algorithm is $O(N^2 logN)$.

Significance of the Proposed Work

In this work, new parallel algorithms for single link and complete link hierarchical clustering are proposed using a SIMD linear array with *N* PEs interfaced to a host machine. For the single link clustering, each PE maintains a sorted list of its first *logN* nearest neighbors before beginning the clustering. At any point during the clustering, the host maintains a heap of all the nearest neighbors of all the PEs. The determination of the smallest entry in the distance matrix and updates to the distance matrix is achieved in *O(logN)* time. In the case of complete link clustering, a heap data structure is used to store the inter-pattern distances in the PEs, thereby speeding up the minimum determination during each iteration. The time complexity of the proposed parallel algorithms for single link and complete link clustering are shown to be *O(NlogN)* and $O(N^2)$ respectively.

Chapter Four

Proposed Parallel Algorithms

The parallel algorithms for single link and complete link clustering proposed in this work are based on a SIMD computational model with a linear interconnection network. The model consists of a linear array of $N (= 2^n)$ PEs, where N is the number of patterns to be clustered, interfaced to a host machine. The PEs are indexed *1* through *N*. Figure 6 gives an illustration of the SIMD computational model.

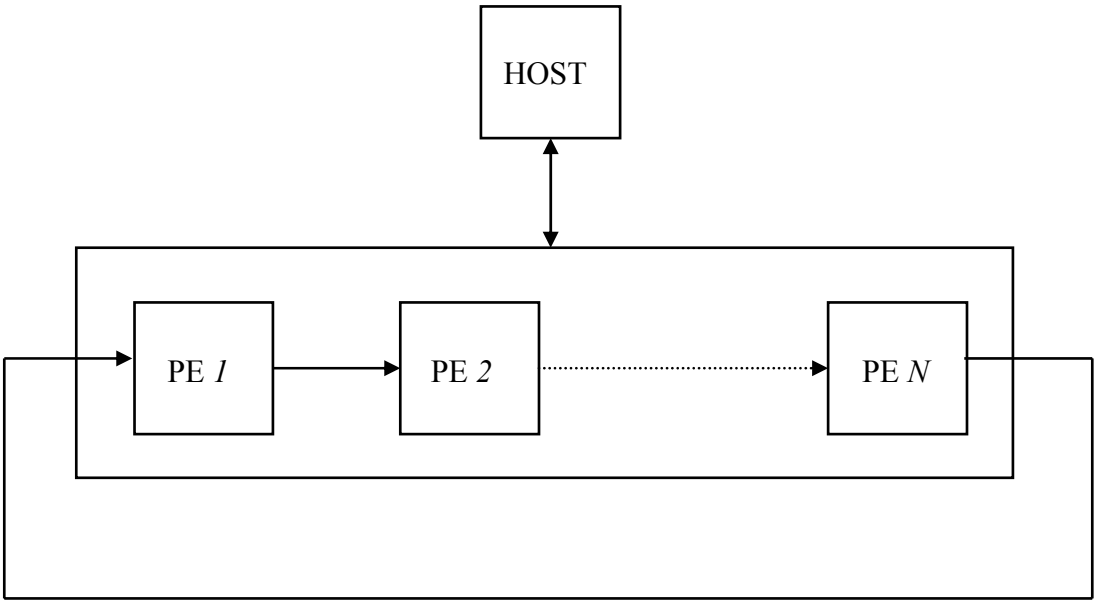

Figure 6. Proposed SIMD Computational Model

The interconnection network provides inter-PE and PE-to-host/host-to-PE communication. Each PE has an *O(N)* memory. For discussion purposes, let us consider *r* to be the PE index. The routing function is defined as: *linear(r) = (r + 1) mod N.*
As the clustering algorithm starts by treating each pattern as a unique cluster, in the proposed model, each PE can initially be visualized as a cluster. The host machine has an *O(N)* memory to store the distance values, the associated cluster pairs, the revised distances during each iteration and the cluster indices which are merged at each stage. The input (pattern matrix) to the proposed algorithm is stored in main memory on the host. The following operations are considered to be unit step operations:

1. a step of data transfer from one PE to another PE which is directly connected
2. a step of arithmetic or logic operation performed either on the host or the PE
3. a step of data transfer from one PE to the host or vice versa
4. a broadcast operation from the host to all PEs

Computation of the Distance Matrix

The input to the proposed algorithm is the pattern matrix. The distance matrix is first computed before beginning the clustering. Two procedures are described: one to load the pattern matrix onto the *N* PEs and the other to compute the inter-pattern distances. These two procedures (Appendix A) are common to both the single link and complete link clustering algorithms.

The procedure *Load Pattern* loads the pattern vectors onto the PEs, from the main memory on the host in such a way that all the features of a particular pattern, say *x*, is in

PE *x*. In other words each PE now represents a pattern. The time complexity of this procedure is *O(NM)*, where *N* is the number of patterns and *M* is the number of features.

The procedure *Compute Distance* calculates and stores the distance matrix in such a way that each PE stores the distance between itself and all other patterns. The criterion chosen is the Euclidean distance which is a commonly used measure. The time complexity of this procedure is *O(NM)*, where *N* is the number of patterns and *M* is the number of features.

Heap Procedures Used

To effectively construct the heap, update an element, delete an element and insert an element into the heap, standard heap procedures available in the literature are used [2]. The pseudo code for these procedures can be seen in Appendix B. The procedure *ReestablishHeap* is used to exchange the elements of the array such that it satisfies the condition of a heap. Procedure *ShiftUp* is used to update an element and maintain the heap property if the updated value is lesser than the original value. If the updated value is larger than the original value, then procedure *ReestablishHeap* can be used to properly shift down the element so that the heap property is satisfied. The procedure *DeleteHeap* which is used to remove an element from the heap is also easily implemented using the procedures *ReestablishHeap* and *ShiftUp*. Procedure *InsertHeap* is used to insert an element into the heap and rearrange the elements to satisfy the heap property. The time complexity for all these procedures is shown in Table 1.

Table 1. Time Complexity of Heap Procedures Used

Procedure Name	Time Complexity
ReestablishHeap	$7.5logN + 4$
HeapConstruct	$19N - 7.5logN - 15$
ShiftUp	$5logN + 4$
UpdateHeap	$7.5logN + 5$
DeleteHeap	$7.5logN + 8$
InsertHeap	$5logN + 5$

Parallel Algorithm for Single Link Clustering

In the case of single link clustering, storing the nearest neighbor distance of each cluster speeds up the algorithm by *O(N)* as the minimum determination at each stage only involves N elements as opposed to N^2 elements as in the Johnson's scheme. In the proposed algorithm for single link clustering, this technique is used. Although the distance matrix is symmetric, in this approach, the PEs store the distances between themselves and all other patterns. If we consider the full distance matrix, it can be seen that the nearest neighbor distances of all the clusters will come into the picture at some point or other during the process of single link clustering. The PEs maintain a register *CLUSTER_FLAG* to indicate if that particular pattern is clustered or not. The host machine maintains a heap of the current nearest neighbors of all the PEs and also keeps track of which patterns have been clustered at any given point during the clustering. For

the single link clustering case, it is enough if each PE stores the first *logN* minimums, in a sorted order. *N – 1* minimum values are required to complete the single link clustering algorithm. In this case, it is possible that two PEs may have the same nearest neighbor distance with the partners being each other. For example, PE *x* may have the nearest neighbor to be *y* and PE *y* may have the nearest neighbor to be *x*. Thus the worst case scenario will be where we have only N/2 minimum values at the first stage that would participate in the clustering. Storing the first *logN* minimum values in each PE will be sufficient as all the minimum values which should be taking part in the clustering will be spread across all the PEs. The proposed parallel algorithm for single link clustering proceeds as follows:

Step 1. Once the inter-pattern distances are computed, each PE constructs a heap out of the *N – 1* distance values stored.

Step 2. Each PE determines its first *logN* minimum values and stores it in an array along with the associated index of the pair.

Step 3. Each PE sends the first element of its sorted list to the host.

Step 4. The host constructs a heap out of the *N* values it receives. The host also sets up an array which is used to identify at any point, if a pattern is clustered or not.

Step 5. The host checks its root element to see if either of the pattern indices associated with that distance is clustered or not. If either of them is clustered, it deletes that element from the heap; else, it broadcasts that element to all the PEs.

Step 6. The PEs receive the broadcasted pair and store it in registers P_1 and P_2.

Step 7. If the index of the PE is not equal to P_1 or P_2, then it updates the distance value

between itself and P_1. If the index of the PE is equal to P_2, then it sets the *CLUSTER_FLAG* to TRUE and sends all the elements it has, excluding the first element, to the host. PE with index equal to P_1 deletes its first element and rearranges the sorted list.

Step 8. PE with index equal to P_1 receives the P_2 related distances from the host and builds a new sorted list with the revised distances.

Step 9. If there are only two clusters left, merge them and stop. Else, go to *Step 5*.

The time complexity of the proposed single link clustering algorithm (Appendix C) is $20NlogN + 61N - 43logN + 7.5(logN)^2 - 94$.

Parallel Algorithm for Complete Link Clustering

In the case of complete link clustering, we cannot store the nearest neighbor distances between the patterns as we were able to do with single link clustering, because, the inter-cluster distances can get larger. However, by using an efficient heap data structure to store the inter-cluster distances, the determination of the smallest entry during each iteration can be speeded up. In the proposed algorithm, each PE maintains a heap of all the distances it has and the nearest neighbor is given by the root element of the heap. Each PE maintains a register *CLUSTER_FLAG* which indicates if it has been clustered or not.

The proposed parallel algorithm for complete link clustering proceeds as follows:

Step 1. Once the inter-pattern distances are computed, each PE constructs a heap out of the $N - 1$ distance values stored. The root element of the heap in each PE is the nearest neighbor of the cluster that corresponds to that PE.

Step 2. Determine the minimum of all the root elements in the PEs and send the cluster pair associated with that minimum distance to the host.

Step 3. The host broadcasts the cluster pair to all the PEs and the PEs store those indices in two registers P_1 and P_2.

Step 4. The PE with index equal to P_1 removes the root element in its heap. The PE with index equal to P_2 sets its *CLUSTER_FLAG* to TRUE. The PEs with index not equal to P_1 or P_2 and do not have their *CLUSTER_FLAG* set to TRUE, will be involved in updating the distance between themselves and the cluster with index equal to P_1, based on the following criteria: if the distance between themselves and the cluster with index equal to P_1 is greater than the distance between themselves and the cluster with index equal to P_2, then they remove the latter distance value from their heap; if the distance between themselves and the cluster with index equal to P_1 is lesser than the distance between themselves and the cluster with index equal to P_2, then they update the former distance value with the latter one and remove the latter distance value from their heap.

Step 5. Each PE sends the updated distance between itself and the cluster with index equal to P_1 to the host.

Step 6. The PE with index equal to P_1 receives the revised distance values from the host and reconstructs the heap with those elements.

Step 7. If only two clusters are left, merge those two and stop. Else, go to *Step 2*.

The time complexity of the proposed complete link clustering algorithm (Appendix D) is $15N^2 - 64.5N - 37.5logN + 74$.

Partitionability of the Proposed Algorithms

The proposed algorithms can be mapped onto a fixed array of Q processors. When the number of patterns to be clustered, N, is greater than Q, the pattern assignments to the PEs will be wrapped around instead of each pattern being assigned to a PE as was the case when the number of processors was equal to the number of patterns. So, each PE would hold $\lceil N/Q \rceil$ rows of the distance matrix. The time complexity of the proposed single link clustering algorithm then becomes $O(N \lceil N/Q \rceil logN)$ and that for the complete link clustering algorithm becomes $O(N2 \lceil N/Q \rceil)$.

Chapter Five

Simulation Results and Performance Comparison

To demonstrate and verify the parallel algorithms proposed, these algorithms were implemented on the Intel iPSC/2 concurrent computer. The distance matrix used is the same as shown in Figure 1. Table 2 shows the performance comparison of the proposed single link clustering algorithm with Li's single link clustering algorithm. The performance comparison of the proposed complete link clustering algorithm with Li's complete link clustering algorithm is shown in Table 3. A numerical comparison between the proposed algorithms and Li's parallel algorithms, for clustering a data set with 64, 128, 256, 512 and 1024 patterns is shown in Table 4. All the numbers in this table are in unit steps. The figures given do not take the distance matrix computation into account.

Table 2. Performance Comparison of Single Link Clustering Algorithms

	No.of PEs	Memory per PE	Interconnection Network	Time Complexity
Li	N	$O(N)$	Shuffle-Exchange	$39N\log N + 22N - 39.5\log N - 20$
Proposed	N	$O(N)$	Linear	$20N\log N + 61N + 7.5(\log N)^2 - 43\log N - 94$

Table 3. Performance Comparison of Complete Link Clustering Algorithms

	No.of PEs	Memory per PE	Interconnection Network	Time Complexity
Li	N	$O(N)$	Shuffle-Exchange	$8N^2 logN + 4N^2 + 14.5NlogN + 11N - 22.5logN - 14$
Proposed	N	$O(N)$	Linear	$15N^2 - 64.5N - 37.5logN + 74$

Table 4. Numerical Comparison

N	Number of unit steps			
	Li's algorithm		Proposed algorithm	
	Single Link	Complete Link	Single Link	Complete Link
64	16127	219115	11502	57161
128	37464	997269	25701	237316
256	85168	4488766	56618	966302
512	190600	19995176	123519	3898873
1024	421473	88239888	267490	15662291

Chapter Six

Conclusion

Hierarchical clustering algorithms are computationally intensive in nature. This thesis proposes new parallel algorithms for Single Link and Complete Link hierarchical clustering on a SIMD machine model. The model consists of *N* PEs connected by a linear interconnection network and interfaced to a host machine. In the case of single link clustering, by having the PEs maintain a sorted list of only their first *logN* nearest neighbors and the host maintain a heap of the nearest neighbors of all the PEs, the time complexity is shown to be *O(NlogN)*. For the complete link clustering problem, even though the nearest neighbors of the PEs cannot be stored because of the fact that the inter-pattern distances can get larger, by having each PE maintain a heap of all its inter-pattern distances, determining the smallest entry in the distance matrix during each iteration is achieved in *O(N)* time. Thus the overall time complexity of the complete link clustering algorithm is shown to be $O(N^2)$. The proposed algorithms have been shown to achieve reasonable speed up over the previous approaches.

References

[1] Anderberg, M. R. (1973). *Cluster Analysis for Applications*. Academic Press.

[2] Baase, S. (1988). *Computer Algorithms*. Addison-Wesley Publishing Company.

[3] Fu, K. S. (1984). *VLSI for Pattern Recognition and Image Processing*. Springer-Verlag.

[4] Fu, K. S. & Ichikawa, T. (1982). *Special Computer Architectures for Pattern Processing*. CRC Press, Florida.

[5] Hartigan, J. A. (1975). *Clustering Algorithms*. Wiley Series in probability and mathematical statistics.

[6] Hattori, K. & Torii, Y. (1993). *Effective Algorithms for the nearest neighbor method in the clustering problem*. Pattern Recognition, vol. 26, no. 5, pp. 741-746.

[7] Hwang, K. & Fu, K. S. (1983). *Integrated Computer Architectures for Image Processing and database management*. Computer, pp 51-60.

[8] Jain, A. K. & Dubes, R. C. (1988). *Algorithms for Clustering Data*. Prentice Hall Advanced Reference Series.

[9] Johnson, S. (1967). *Hierarchical clustering schemes*. Phychometrika, vol. 23, pp 241-254.

[10] Li, X. & Fang, Z. (1986). *Parallel Algorithms for Clustering on Hypercube SIMD computers*. Proceedings of 1986 Conference on Computer Vision and Pattern Recognition, pp. 130-133.

[11] Li, X. & Fang, Z. (1989). *Parallel Clustering Algorithms*. Parallel Computing, vol. 11, no. 3, pp. 275-290.

[12] Li, X. (1990). *Parallel Algorithms for Hierarchical Clustering and Cluster Validity*. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 12, no. 11, pp. 1088-1092.

[13] Ni, L. M. & Jain, A. K. (1985). *A VLSI Systolic Architecture for pattern clustering*. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 7, no. 1, pp. 80-89.

[14] Olson, C. F. (1995). *Parallel Algorithms for Hierarchical Clustering*. Parallel Computing, vol. 21, pp. 1313 – 1325.

[15] Rajasekaran, S. (2005). Efficient Parallel Hierarchical Clustering Algorithms. IEEE Transactions on Parallel and Distributed Systems, vol. 16, No. 6, pp. 497-502.

[16] Ranka, S. & Sahni, S. (1990). *Clustering on a Hypercube Multicomputer*. Proceedings of 1990 Conference on Computer Vision and Pattern Recognition, pp. 532-536.

[17] Spath, H. (1980). *Cluster Analysis Algorithms for data reduction and classification of objects*. Ellis Horwood Publishers.

[18] Tsai, H. R., Horng, S. J., Lee, S. S., Tsai, S. S. & Kao, T. W. (1997). *Parallel Hierarchical Clustering Algorithms on Processor Arrays with a Reconfigurable Bus System*. Pattern Recognition, vol. 30, pp. 801-815.

[19] Wu, C. H., Horng S. J. & Tsai, H. R. (2000). *Efficient Parallel Algorithms for Hierarchical Clustering on Arrays with Reconfigurable Optical Buses*. Journal on Parallel and Distributed Computing, vol. 60, pp. 1137-1153.

[20] Zupan, J. (1982). *Clustering of Large Data Sets*. Research Studies Press.

Appendices

Appendix A: Procedure Load Pattern & Compute Distance

PROCEDURE Load Pattern:

Input: Pattern matrix P of size N x M (N is the number of patterns and M is the number of features), stored in the host.

Output: Store P[i, j] for j = 0, 1, 2, … M-1, in PE with index i+1 for i = 0, 1, 2, … N-1

```
BEGIN
    // Initialization //
    for j = 0 to M-1
        m[j] = -1;
    end for
    // Load the feature values of all the patterns onto the respective PEs //
    for j = 0 to M-1
        for i = N-1 to 0 step -1
                // PE with index 1 gets P(i, j) from the host //
                if (index == 1) then m[j] ← P[i, j] ((host));
                // Propogate the j'th feature of pattern i to PE with index i+1 //
                if (m[j] <> -1 && i <> 0) then m[j]((index)) → m[j]((linear(index)));
        end for
    end for
END
```

Appendix A: (Continued)

PROCEDURE Compute Distance:

Input: The feature values m[j] for j = 0, 1, 2, … M-1, stored in each PE

Output: The populated arrays *dis* (where each PE stores the distance between itself and all other clusters), *pair* (used to store the indices of clusters which are associated with each distance value in the array *dis*) and *location* (used by each PE to store the location of the distance values between itself and other clusters)

```
BEGIN
    // Initialization
    pair_index = index;
    for j = 0 to M-1
        temp[j] = m[j];
    end for
    // Compute the inter-pattern distances and store them
    for i = 1 to N-1
        sum = 0;
        pair_index ((index)) → pair_index ((linear(index)));
        for j = 0 to M-1
                temp[j] ((index)) → temp[j] ((linear(index)));
                sum = sum + ((m[j] – temp[j]) * (m[j] – temp[j]));
        end for
        dis[i] = sum;
```

Appendix A: (Continued)

```
        pair[i] = pair_index;
        location[pair_index] = i;
    end for
END
```

Appendix B: Standard Heap Procedures

PROCEDURE ReestablishHeap (loc, key, partner, bound):

Input: The distance value given by *key*

Output: The heap with the keys properly arranged

```
BEGIN
    empty = loc;
    while (2 * empty <= bound) do
        smallerchild = 2 * empty;
        if (smallerchild < bound) && (dis[smallerchild + 1] < dis[smallerchild]) then
                smallerchild = smallerchild + 1;
        if (key > dis[smallerchild]) then
                dis[empty] = dis[smallerchild];
                pair[empty] = pair[smallerchild];
                location[pair[empty]] = empty;
                empty = smallerchild;
        else exitloop
        end if
    end while
    dis[empty] = key;
    pair[empty] = partner;
    location[pair[empty]] = empty;
END
```

Appendix B: (Continued)

PROCEDURE HeapConstruct:

Input: The array *dis* with the distances in arbitrary positions

Output: The same array satisfying the heap property

```
BEGIN
    for i = floor(num_dis/2) to 1 step -1 do
        ReestablishHeap(i, dis[i], pair[i], location[pair[i]], num_dis);
    end for
END
```

PROCEDURE ShiftUp(loc, key, partner):

Input: The updated element

Output: The heap with the keys rearranged to satisfy the heap property

```
BEGIN
    empty = loc;
    while (empty/2 >= 1) do
        largerparent = floor(empty/2);
        if (dis[largerparent] > key) then
                dis[empty] = dis[largerparent];
                pair[empty] = pair[largerparent];
                location[pair[empty]] = empty;
                empty = largerparent;
```

Appendix B: (Continued)

```
        else exitloop
        end if
    end while
    dis[empty] = key;
    pair[empty] = partner;
    location[pair[empty]] = empty;
END

PROCEDURE UpdateHeap (index, value):
Input: The index of the element to be updated and the value to be updated with
Output: The heap with all the keys rearranged to satisfy the heap property
BEGIN
    if (value > dis[index]) then
        dis[index] = value;
        ReestablishHeap (index, dis[index], pair[index], num_dis);
    else if (value < dis[index]) then
        dis[index] = value;
        ShiftUp(index, dis[index], pair[index]);
    end if
END
```

Appendix B: (Continued)

```
PROCEDURE DeleteHeap:
Input: The index of the element to be removed
Output: The heap with the keys rearranged to satisfy the heap property
BEGIN
    x = dis[index];
    dis[index] = dis[num_dis];
    pair[index] = pair[num_dis];
    location[pair[index]] = index;
    if (x < dis[num_dis]) then
        ReestablishHeap(index, dis[index], pair[index], num_dis-1);
    else if (x > dis[num_dis]) then
        ShiftUp(index, dis[index], pair[index]);
    end if
END

PROCEDURE InsertHeap (key, bound):
Input: The key to be inserted
Output: The heap with the keys rearranged to satisfy the heap property
BEGIN
    bound = bound + 1;
    i = bound;
```

Appendix B: (Continued)

```
    while (i > 1 && heap[parent[i]] < key) do
        heap[i] = heap[parent[i]];
        i = parent[i];
    end while
    heap[i] = key;
END
```

Appendix C: Procedure SINGLELINK

```
BEGIN
    // Load the pattern vectors onto the PEs
    Load Pattern;
    // Compute the inter-pattern distances
    Compute Distance:
    // Initialization
    num_dis = N – 1;
    CLUSTER_FLAG = FALSE;
    // Each PE constructs a heap out of the N-1 distance values it has
    for i = floor(num_dis/2) to 1 step -1
        ReestablishHeap(i, dis[i], pair[i], num_dis);
    end for
    // Each PE determines the first floor(logN) minimum values and stores them in the
    array local_dis. local_pair stores the associated pattern index and local_location is a
    pointer to the distance value
    temp = 1;
    for i = N to (N – logN – 1) step -1 do
        min_dis = dis[1];
        min_pair = pair[1];
        ReestablishHeap(1, dis[i], pair[i], i-1);
        local_dis[temp] = min_dis;
```

Appendix C: (Continued)

```
        local_pair[temp] = min_pair ;
        local_location[min_pair] = temp ;
        temp = temp + 1 ;
    end for
    // Each PE sends the first element in its sorted array local_dis to the host
    host_index((host)) = 1;
    for i = 1 to N
        if (index == i) then
                index → pattern1[host_index]((host));
                local_dis[1] → dis_value[host_index]((host));
                local_pair[1] → pattern2[host_index]((host));
                host_index((host)) ++;
        end if
    end for
    // The host now constructs a heap of the elements it received
    HeapConstruct();
    // Start the clustering
    for level = 1 to N-2
        if (pattern1[1]((host)) && pattern2[1]((host)) not in same cluster) then
                pattern1[1] ((host)) => P1;
                pattern2[1] ((host)) => P2;
```

Appendix C: (Continued)

```
else
        DeleteHeap(1) ((host));
end if
// Update the distances
if (index <> P1 && index <> P2) then
        if (local_dis[local_location[P1]] < local_dis[local_location[P2]]) then
                remove data related to P2;
        else
                update distances related to P2 as P1;
                remove data related to P1;
        end if
end if
CLUSTER_FLAG = TRUE (( PE with index P2));
// The host collects the distance values from PE P1 and P2
collect_index ((host)) = 1;
for h = 1 to floor(logN)
        local_dis[h] ((PE with index P1)) → p1collect[collect_index]((host));
        local_dis[h] ((PE with index P2)) → p2collect[collect_index]((host));
        collect_index((host)) ++;
end for
// The host merges at most 2*floor(logN) values to determine the first floor(logN)
```

Appendix C: (Continued)

```
        minimum distance values
        host_merge();
        // The revised distances are in the array merged_dis on the host
        // Insert merged_dis[i] into the heap in the host
        InsertHeap(merged_dis[i], collect_index);
        // The merged distances are transferred to PE with index P1
        for k = 1 to collect_index
                merged_dis[k] ((host)) → local_dis[k] ((PE with index P1));
                merged_pair[k] ((host)) → local_pair[k] ((PE with index P1));
        end for
    end for
END
```

Appendix D: Procedure COMPLETELINK

```
BEGIN
    // Load the pattern vectors onto the PEs
    Load Pattern();
    // Compute the inter-pattern distances
    Compute Distance();
    // Initialization
    num_dis = N-1;
    CLUSTER_FLAG = FALSE;
    // Each PE constructs a heap out of the N-1 distance values it has
    for i = floor(num_dis/2) to 1 step -1
        ReestablishHeap(i, dis[i], pair[i], num_dis);
    end for
    // Start the clustering
    for level = 1 to N-2 do
        // Determine the minimum of the N root elements in the PEs
        index_t = index;
        pair_index_t = pair[1];
        dis_value_t = dis_value[1];
        for j = 1 to N-1
                index_t ((index)) → index_t ((linear(index)));
```

Appendix D: (Continued)

```
            pair_index_t ((index)) → pair_index_t((linear(index)));
            dis_value_t ((index)) → dis_value_t((linear(index)));
            if (index <> 0 && index <> N-1 && CLUSTER_FLAG <> TRUE) then
                  if (dis[1] < dis_value_t) then
                        index_t = index;
                        pair_index_t = pair[1];
                        dis_value_t = dis[1];
                  end if
            end if
      end for
      // PE N sends to the host the indices of the cluster pair with the minimum distance
      if (index == N) then
            index_t → pattern1[level] ((host));
            pair_index_t → pattern2[level] ((host));
      end if
      // The host sends the two indices to all the PEs
      pattern1[level] ((host)) => P1;
      pattern2[level] ((host)) => P2;
      // Update the distance values
      if (index <> P1 or index <> P2 && CLUSTER_FLAG <> TRUE) then
            if (dis[location[P1]] >= dis[location[P2]]) then
```

Appendix D: (Continued)

```
                DeleteHeap(location[P2]);
        else
                UpdateHeap(location[P1], dis[location[P2]]);
                DeleteHeap(location[P2]) ;
        end if
else
        if (index == P1) then DeleteHeap(location[P2]) ;
        else if (index == P2) then CLUSTER_FLAG = TRUE;
        end if
end if
// Update the distances in the PE with index = P1
count ((host)) = 1;
for i = 1 to N
        if (index <> P1) then
                dis[location[P1]] ((index)) → rev_dis[count] ((host));
                index ((index)) → pattern_index[count] ((host));
                count ((host)) ++;
        end if
end for
for h = 1 to count
        rev_dis[h] ((host)) → dis[h] ((PE with index P1));
```

Appendix D: (Continued)

```
                pattern_index[h] ((host)) → pair[h] ((PE with index P1));
        end for
        // Rearrange the heap of elements in PE with index P1
        if (index == P1) then
                for m = floor(num_dis/2) to 1 step -1
                        ReestablishHeap(m, dis[m], pair[m], location[pair[m]], num_dis);
                end for
        end if
    end for
    // Merge the last two clusters left behind
    for d = 1 to N
        if (index == d && CLUSTER_FLAG == FALSE) then
                index ((index)) → pattern1[N-1] ((host));
                pair[1] ((index)) → pattern2[N-1] ((host);
        end if
    end for
END
```