Parallel Algorithms for Hierarchical Clustering

Clark F. Olson
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720
clarko@robotics.eecs.berkeley.edu

December 28, 1993

Abstract

Hierarchical clustering is a common method used to determine clusters of similar data points in multi-dimensional spaces. $O(n^2)$ algorithms, where n is the number of points to cluster, have long been known for this problem [24, 7, 6]. This paper discusses parallel algorithms to perform hierarchical clustering using various distance metrics. I describe $O(n)$ time algorithms for clustering using the single link, average link, complete link, centroid, median, and minimum variance metrics on an n node CRCW PRAM and $O(n \log n)$ algorithms for these metrics (except average link and complete link) on $\frac{n}{\log n}$ node butterfly networks or trees. Thus, optimal efficiency is achieved for a significant number of processors using these distance metrics. A general algorithm is given that can be used to perform clustering with the complete link and average link metrics on a butterfly. While this algorithm achieves optimal efficiency for the general class of metrics, it is not optimal for the specific cases of complete link and average link clustering.

1 Introduction

Clustering of multi-dimensional data is required in many fields. For example, in model-based object recognition, pose clustering is used to determine possible locations of an object in an image [25, 26, 19]. Some possible methods of clustering data are:

1. Hierarchical Clustering: These methods start with each point being considered a cluster and recursively combine pairs of clusters (subsequently updating the intercluster distances) until all points are part of one hierarchically constructed cluster.

2. Partitional Clustering: The methods start with each point as part of a random or guessed cluster and iteratively move points between clusters until some local minimum is found with respect to some distance metric between each point and the center of the cluster it belongs to.

3. Binning: These methods partition the space into many (possibly overlapping) bins and determine clusters by finding bins that contain many data points.

For many applications partitional clustering is not ideal since the approximate number of clusters must be known in advance and each data point must be place in some cluster. Binning has low time requirement with respect to the number of data points, but suffers from the problems that multi-dimensional spaces require a large number of bins and accuracy declines due to the separations imposed by the bin boundaries. If a reasonable distance metric can be defined on the multi-dimensional space, hierarchical clustering is the ideal method of clustering, but has not been used due to an $O(n^2)$ time complexity. (Pose clustering, for example, requires clustering a number of points polynomial in the number of image and model features.) Efficient parallel clustering algorithms can reduce this burden and make hierarchical clustering algorithms more useful for many applications.

The final hierarchical cluster structure can be represented by a dendrogram (see Figure 1.) The dendrogram can be easily broken at selected links to obtain clusters of desired cardinality or radius. We can create a representation of the dendrogram internal to the computer by simply storing which pair of clusters are merged at each step. This representation is easy to generate even in a distributed fashion, so I will ignore its generation for the most part, concentrating on the determination of which clusters to merge.

Individual data points are vectors in a d-dimensional space. I consider cases where the dimensionality of the space d is fixed. Most of the algorithms presented have a linear time dependence on d. I will note where this is not the case. The Euclidean distance is usually used to determine the distance between any two points. But, even restricting ourselves to hierarchical clustering, there are a number of methods of determining the distances between clusters. See Murtagh [18] for additional details. The distance metrics that I will examine can be broken into two general classes :

1. **Graph methods** These methods determine intercluster distances using the graph of points in the two clusters. Examples include:
   - Single link: The distance between any two clusters is the minimum distance between two points such that one of the points is in each of the clusters.
   - Average link: The distance between any two clusters is the average distance between each pair of points such that each pair has a point in both clusters.
   - Complete link: The distance between any two clusters is the maximum distance between two points such that one of the points is in each of the clusters.
2. **Geometric methods** These methods define a cluster center for each cluster and use these cluster centers to determine the distances between clusters. For example:

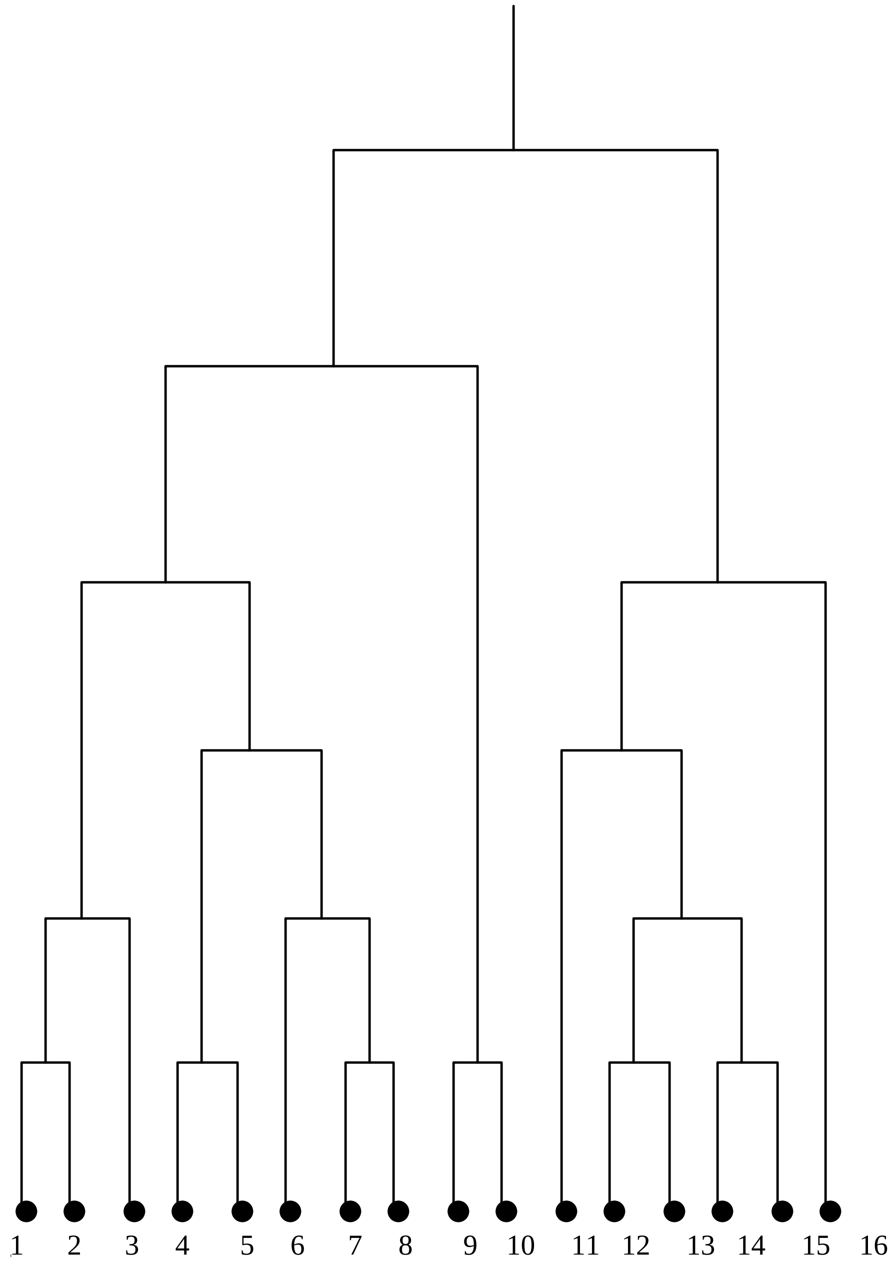

Figure 1: A dendrogram shows how the clusters are merged hierarchically.

- Centroid: The cluster center is the centroid of the points in the cluster. The Euclidean distance between cluster centers is used.
- Median: The cluster center is the (unweighted) average of the centers of the two clusters agglomerated to form it. The Euclidean distance between cluster centers is used.
- Minimum Variance: The cluster center is the centroid of the points in the cluster. The distance between two clusters is the increase in the sum of squared distances from each point to the center of its cluster caused by agglomerating the clusters.

Useful clustering metrics can usually be described using the Lance-Williams updating formula [12]:

$$d(i+j,k) = a(i)d(i,k) + a(j)d(j,k) + bd(i,j) + c \mid d(i,k) - d(j,k) \mid$$

So, the distance between a new cluster (combining the two previous clusters i and j) and the cluster k is a function of the previous distances between the clusters, with coefficients (shown in Table 1 for the metrics described above) that are functions of the cardinality of the clusters.

$O(n^2)$ time algorithms exist to perform clustering using each of the metrics described above. This is not quite of optimal worst-case efficiency for single link clustering, since the single link hierarchy can be easily determined from the minimum spanning tree of a set of points [18], and the Euclidean minimum spanning tree of n points in d dimensions can be determined in $O(n^{2-a(d)} \log^{1-a(d)} n)$ time where $a(d) = 2^{-d-1}$ [27]. This algorithm is not practical in most cases, so I will consider $O(n^2)$ to be the optimal practical efficiency. In addition, any metric that can be described by the Lance-Williams updating formula can be performed in $O(n^2 \log n)$ time [6].

Metric	$a(i)$	b	c
Single link	$\frac{1}{2}$	0	$-\frac{1}{2}$
Average link	$\frac{\|i\|}{\|i\|+\|j\|}$	0	0
Complete link	$\frac{1}{2}$	0	$\frac{1}{2}$
Centroid	$\frac{\|i\|}{\|i\|+\|j\|}$	$-\frac{\|i\|\|j\|}{(\|i\|+\|j\|)^2}$	0
Median	$\frac{1}{2}$	$-\frac{1}{4}$	0
Minimum variance	$\frac{\|i\|+\|k\|}{\|i\|+\|j\|+\|k\|}$	$-\frac{\|k\|}{\|i\|+\|j\|+\|k\|}$	0

Table 1: Parameters in the Lance-Williams update formula for various clustering metrics. ($\mid x \mid$ is the number of points in cluster x.)

I present $O(n)$ time algorithms for hierarchical clustering using each of the specific metrics above on a n node CRCW PRAM, and $O(n \log n)$ time algorithms for these metrics (except average link and complete link) on an $\frac{n}{\log n}$ node tree or butterfly. In addition, I show how general clustering algorithms that can be described by the Lance-Williams updating formula can be performed in $O(n \log n)$ time on an n node butterfly. Thus, I give optimal, worst-case efficiency parallel algorithms for a general class of hierarchical clustering algorithms and for several cases of clustering using the specific metrics.

2 Sequential Algorithms

Several important results on sequential hierarchical clustering algorithms are summarized in [17] and some more recent results are presented in [6]. I will briefly describe some of these.

2.1 Single link, median, and centroid metrics

Clustering using the single link metric is closely related to finding the Euclidean minimal spanning tree of a set of points. The edges of the minimal spanning tree correspond to the cluster agglomerations if taken in order from smallest edge to largest edge, where the clusters of points agglomerated are those connected by the subtree of edges already examined. While $o(n^2)$ algorithms exist to find the Euclidean minimal spanning tree [27], these algorithms are impractical for $d > 2$.

An $O(n^2)$ time sequential algorithm for the single link method proceeds as follows:

1. Determine and store the distance between each pair of clusters. (Initially, each point is considered a cluster by itself.) Also, for each cluster determine its nearest neighbor.

2. Determine the pair of clusters with the smallest distance between them and agglomerate them.

3. Update the pairwise distances and the new nearest neighbors.

4. If more than one cluster still exists goto Step 2.

This algorithm can be performed in $O(n^2)$ time, since Step 1 requires $O(n^2)$ time and Steps 2-4 are performed $n-1$ times and they can be completed in $O(n)$ time. Step 3 requires determining the new nearest neighbor for clusters that had one of the agglomerated clusters as their nearest neighbor, but in the single link case we are guaranteed that the new nearest neighbor will be the new agglomerated cluster. For clustering using other metrics, this step is more complicated since the new nearest neighbor may be any cluster. Finding the new nearest neighbor requires $O(n)$ time for each cluster whose nearest neighbor is agglomerated in these cases. For the centroid and median metrics, Day and Edelsbrunner [6] have shown that the number of such clusters is limited by a function of size $O(\min(2^d, n))$, which for constant d is $O(1)$. So, we can modify the above algorithm to perform clustering using the centroid metric by including in Step 3, the generation of the nearest neighbors for the $O(1)$ clusters whose nearest neighbor was agglomerated without changing the order of the running time.

Note that this algorithm requires $O(n^2)$ space to store each of the pairwise distances for the single link metric. (For the centroid and median metrics, we can store the cluster centers in $O(n)$ space a generate the distances as needed.) Sibson [24] gives an algorithm for the single link case requiring $O(n)$ space and Defays [7] gives a similar algorithm to perform complete link clustering in $O(n^2)$ time and $O(n)$ space.

2.2 Metrics satisfying the reducibility property

An $O(n^2)$ algorithm using nearest neighbor chains is given next. To produce exact results, this algorithm requires that the distance metric satisfies the reducibility property [5]. The reducibility property requires that if the following distance constraints hold for clusters i, j, and k for some distance ρ:

$$d(i,j) < \rho$$
$$d(i,k) > \rho$$
$$d(j,k) > \rho$$

then we must have for the agglomerated cluster $i + j$:

$$d(i+j,k) > \rho.$$

The minimum variance metric and the graph theoretical metrics satisfy this property, so they are ideal metrics for use with this algorithm. The centroid and median metrics do not satisfy the reducibility property. This algorithm still provides a good approximate algorithm for these cases, but the order of examining the points can change the final hierarchy.

The algorithm is as follows:

1. Pick any cluster i_1.

2. Determine the list $i_2 = NN(i_1)$, $i_3 = NN(i_2)$,... until $i_k = NN(i_{k-1})$ and $i_{k-1} = NN(i_k)$, where $NN(x)$ is the nearest neighbor of cluster x.

3. Agglomerate clusters i_{k-1} and i_k and replace them by a single cluster represented by the new cluster center.

4. If more than one cluster still exists goto Step 2 using the previous i_{k-2} as the new i_1 if $k > 2$, otherwise choose arbitrary i_1.

By amortizing the cost of Step 2 over the n iterations, it can be seen that this algorithm requires $O(n^2)$ time and $O(n)$ space for clustering techniques using cluster centers. It can also be modified to perform clustering using graph theoretical metrics by keeping a matrix of the intercluster distances, increasing the space requirement to $O(n^2)$.

2.3 General metrics

For any metric where the modified intercluster distances can be determined in $O(1)$ time (e.g. using the Lance-Williams update formula) clustering can be performed in $O(n^2 \log n)$ time as follows [6]:

1. For each cluster generate a priority queue of the distances to each other cluster.

2. Determine the closest two clusters.

3. Agglomerate the clusters.

4. Update the distances in the priority queues.

5. If more than one cluster remains goto Step 2.

Step 1 can be performed in $O(n^2)$ time. Steps 2-5 are performed $n-1$ times each. The bottleneck is Step 4, which requires creating a new priority queue (for the agglomerated cluster) and updating $O(n)$ priority queues each time it is performed. Since this step is performed n times and each time requires $O(n \log n)$ time, the total time required is $O(n^2 \log n)$.

2.4 Probabilistic algorithms

The expected running time of probabilistic algorithms has also been examined. For example, Murtagh [16] describes an algorithm that achieves $O(n)$ expected time for the centroid method and $O(n \log n)$ time for the minimum variance method under the assumption that the points are uniformly and independently distributed in a bounded region. The algorithms are exponential in the dimensionality of the space, so are practical only for small values of d. Fast expected running time algorithms to determine the Euclidean minimal spanning tree also exist [4, 23], but these algorithms are also exponential in the number of dimensions in the cluster space.

3 Previous Parallel Algorithms

Several authors have previously examined some of the clustering problems I discuss or related problems. I will briefly describe these here.

Rasmussen and Willett [21] discuss parallel implementations of the single link clustering method and the minimum variance method on an SIMD array processor. They have implemented parallel versions of the SLINK algorithm [24], Prim's minimum spanning tree algorithm [20], and Ward's minimum variance method [10]. Their parallel implementations of the SLINK algorithm and Ward's minimum variance algorithm do not decrease the $O(n^2)$ time required by the serial implementation, but a significant constant speedup factor is obtained. Their parallel implementation of Prim's minimum spanning tree algorithm appears to achieve $O(n \log n)$ time with sufficient processors.

Li and Fang [14] describe algorithms for partitional clustering (the k means method) and hierarchical clustering (using the single link metric) on an n-node hypercube and an n-node butterfly. Their algorithms for the single link metric are basically a parallel implementation of Kruskal's minimum spanning tree algorithm [11] and run in $O(n \log n)$ time on the hypercube and $O(n \log^2 n)$ on the butterfly, but in fact the algorithms for hierarchical clustering appear to have a fatal flaw causing incorrect clustering operation. In their algorithm, each processor stores the distance between one cluster and every other cluster. When clusters are agglomerated, they omit the updating step of determining the distances from the new cluster to each of the other clusters. If this step is added to their algorithm in a straightforward manner, the time required by their algorithm increases to $O(n^2)$.

Driscoll *et al.* [9] have described a useful data structure called the relaxed heap and shown how it may be applied to the parallel computation of minimum spanning trees. The relaxed heap is a data structure for manipulating priority queues with the properties that the operation of deleting the minimum element can be performed in $O(\log n)$ time (where n is the number of items in the priority queue) and the operations of decreasing the value of a key can be performed in $O(1)$ time. The use of this data structure allows the parallel implementation of Dijkstra's minimum spanning tree algorithm [8] (also known as Prim's algorithm) in $O(n \log n)$ time using $\frac{m}{n \log n}$ processors, where n is the number of nodes in the graph and m is the number of edges. Since the clustering problem must consider the complete graph on n nodes this implies $\frac{n}{\log n}$ processors are used. This algorithm meets the optimal (practical) efficiency since it performs $O(n^2)$ work. In addition, it appears that using a butterfly or tree rather than a PRAM is sufficient to obtain these times.

4 Parallel Machines

In this section, I will discuss the parallel computers that I describe algorithms for. These can be divided into two categories. First, PRAMs, in which the memory is shared among all processors and thus it does not matter where a value is stored. Second, networks (e.g. butterfly or tree) on which each processor has its own local memory and a interconnection pattern between processors is specified upon which all communication must be performed.

4.1 PRAMs

PRAMs (parallel random access machines) allow each of the processors to access a single parallel memory simultaneously. Three types of PRAMs have been discussed that allow differing types of concurrent access to any single memory location:

1. EREW (exclusive read, exclusive write): Processors must not try to read or write to any memory location simultaneously.

2. CREW (concurrent read, exclusive write): Processors may read from a memory location simultaneously, but may not write to a memory location simultaneously.

3. CRCW (concurrent read, concurrent write): Processors may read from a memory location concurrently or write to a memory location concurrently. Some method of resolving concurrent writes must be specified (e.g. using priorities or arbitrary.)

I will be concerned primarily with CRCW PRAMs in this paper, but EREW PRAMs will also be examined. The key operations that will be needed to implement the clustering algorithms are determining the minimum value of a set of numbers (one for each processor) and broadcasting a value to each processor.

On a CRCW PRAM, both broadcasting and minimization can be performed in constant time. Broadcasting is performed by simply writing the value to a location in memory that each processor can read. Minimization can be performed by each processor writing a value to the same location, using the value as a priority. Thus, the minimum value is the one that will be stored.

On an EREW PRAM, both of these operations take $O(\log n)$ time where n is the number of processors. Broadcasting requires communication between the processors in a binary-tree pattern. The processor that holds the information writes it to the shared memory. A single processor can then read it during the next time-step. Both of these processors can then write the information to the shared memory, allowing two more processors to access the information in the next time-step. This continues until all of the processors have accessed the information. Minimization is performed in a similar fashion, but we now reduce the information by eliminating half of the values at each step. We get the $O(\log n)$ time bound since a tree with n leaves have height $\log n + 1$.

4.2 Networks

In parallel computers with only local memory, determining which processor will store each piece of information is of utmost importance, since any request for this piece of data by a processor on which it is not stored requires communication over the interconnection network between the computers. I will discuss algorithms for two interconnection patterns: butterflies and trees. Details of these the butterfly and tree interconnection patterns can be found in [13]. Importantly, both of these networks have radius $O(\log n)$ where n is the number of processors, so broadcasts and minimizations can be performed in $O(\log n)$ time.

5 Parallel algorithms

A few basic algorithms will be used to implement clustering using the various distance metrics on these parallel computers. In each case, I will first describe the basic algorithm and then describe the implementation details for the specific metrics/computers that we will use them for. In these algorithms, each cluster will be the responsibility one processor (if we use n processors to perform clustering of n points then there is a 1-1 correspondence.) When two clusters are agglomerated, the lower numbered processor corresponding to the two clusters takes over full responsibility for the new cluster. If the other processor no longer has any clusters in its responsibility it becomes idle.

I will measure the amount of work performed by each algorithm as the number of processors used multiplied by the time required. A parallel algorithm will be said to be efficient if the work performed matches the sequential time of the best practical algorithm.

5.1 Algorithm 1

The first algorithm will be similar in operation to the sequential algorithm described in Section 2.1:

1. Create a data structure so that we can easily determine which pair of clusters is closest together.
2. Determine the pair of clusters with the smallest distance between them and agglomerate them.
3. Update the data structure.
4. If more than one cluster still exists goto Step 2.

The important element of this algorithm is the data structure used and method of determining the closest pair of clusters. Step 1 is performed only once and the rest of the steps are performed $n-1$ times.

5.1.1 Single link metric on a PRAM

We can use this algorithm to perform clustering using the single link metric very easily on a PRAM. We just create a two-dimensional array of all of the intercluster distances (each processor is responsible for one column) and a one-dimensional array storing the nearest neighbor of each cluster (and the distance to it.) We can then determine the closest pair of clusters by performing minimization on the nearest neighbor distances. The numbers of these two clusters are then broadcast.

To update the data structure, each processor updates the inter-cluster distance array to reflect the new distance between its clusters and the new agglomerated cluster. For each cluster it is responsible for, each processor must thus update a single location in the array (or two if we maintain a redundant array such that the distance can be indexed by either ordering of the clusters.) No operation need be performed for the agglomerated clusters or the new cluster, since the new distances will be determined by the remaining clusters.

Since the single link metric has the property that if the nearest neighbor j of a cluster k becomes agglomerated into a new cluster $i + j$, then the new nearest neighbor of j must be the new cluster $i + j$, we simply check for this case and modify the nearest neighbor array accordingly. The nearest neighbor of the new cluster is determined using minimization on the distances of each other cluster to it.

Aside from trivial constant time operations, this algorithm requires only $O(1)$ broadcast and minimization operations. Thus, an n processor CRCW PRAM can perform this algorithm in $O(n)$ time. Note that on an EREW PRAM we can perform the operation of $\log n$ clusters on a single processor and still achieve $O(n \log n)$ time since broadcasting and minimization can still be performed in $O(\log n)$ time and the other constant time operations become $O(\log n)$ operations. Thus, this algorithm can be performed with $O(n^2)$ work on both a CRCW PRAM and an EREW PRAM, which is efficient in practice.

Note that this algorithm cannot be performed efficiently on a local memory parallel computer since the distance from the new cluster to each other cluster would be stored on the same processor, requiring $O(n)$ time to update this value at each step.

5.1.2 Centroid and median metrics

The above algorithm cannot be applied directly to other metrics, since only the single link metric has the property that any cluster that had one of the agglomerated clusters as its previous nearest neighbor has the new cluster as its new nearest neighbor. We can sidestep this problem for the centroid and median metrics by using the result of Day and Edelsbrunner [6] that shows that when the dimensionality of the cluster space is fixed, there are $O(1)$ clusters that can have any specific cluster as a nearest neighbor. This number is exponential in the dimensionality of the space.

Note that for geometric intercluster distance metrics we do not need to store the distances between each cluster. We can simply store the cluster centers for each cluster (on each processor, if necessary) and generate each distance in $O(1)$ time as required.

Updating the data structure is now a little more difficult. Since we know that there is a constant maximum number of clusters that can have either of the agglomerated clusters as their nearest neighbors, we can write the numbers of these clusters to a constant sized queue on one of the processors. Determining the nearest neighbors of these clusters can the be performed in constant time in the same manner that the new nearest neighbor for cluster the new cluster is determined. It is now necessary to broadcast the first entry in the queue at each step, but this is easy. Thus, hierarchical clustering using the centroid and median metrics can also be performed in $O(n)$ time on an n node CRCW PRAM and $O(n \log n)$ time on an $\frac{n}{\log n}$ node EREW PRAM, butterfly, or tree.

5.1.3 General metrics

If we try to use this algorithm on general metrics, we have no bound on the number of clusters that may have any particular cluster as a nearest neighbor. In this case, we can employ a priority queue for each cluster to keep track of which clusters are closest to each. Since we only need to know which cluster is the closest at each step this can easily be implemented using a heap (see, for example, [1].)

We can create a priority queue in $O(n)$ time on a single processor, but we must now update each priority queue after each agglomeration, a step that takes $O(\log n)$ time on a PRAM. This algorithm thus requires $O(n \log n)$ time on an n node CRCW

Processor

0	1	2	3
$D_{0,0}$	$D_{1,0}$	$D_{2,0}$	$D_{3,0}$
$D_{0,1}$	$D_{1,1}$	$D_{2,1}$	$D_{3,1}$
$D_{0,2}$	$D_{1,2}$	$D_{2,2}$	$D_{3,2}$
$D_{0,3}$	$D_{1,3}$	$D_{2,3}$	$D_{3,3}$

(a)

Processor

0	1	2	3
$D_{0,0}$	$D_{1,0}$	$D_{2,0}$	$D_{3,0}$
$D_{3,1}$	$D_{0,1}$	$D_{1,1}$	$D_{2,1}$
$D_{2,2}$	$D_{3,2}$	$D_{0,2}$	$D_{1,2}$
$D_{1,3}$	$D_{2,3}$	$D_{3,3}$	$D_{0,3}$

(b)

Figure 2: Distribution of D values between processors. (a) Straightforward distribution (b) Skewed distribution of values

or EREW PRAM, but since the sequential algorithm for general metrics required $O(n^2 \log n)$ time this is still efficient.

Things become more complicated on local memory machines, since we cannot store all of the distances to a single cluster on the same processor (each agglomerated cluster would required $O(n)$ work on a single processor at each step.) We can use a new storage arrangement to facilitate this implementation (see Figure 1.) In the skewed storage arrangement, each processor p stores the distance between clusters i and j if $i + j \bmod n = p$. For this case, each processor must update only two values per agglomeration.

Updating the data structure now requires more work, since each updated distance is the function of three inter-cluster distances in the Lance-Williams update formula and only one of them will be stored on the processor where the result will be determined (and stored.) We need to perform two permutation routing steps to collect this information into the appropriate processors. Permutation routing on a butterfly requires $O(\log^2 n)$ time in the worst case (tree networks will not be considered for this case,) but since we only need to worry about $O(n^2)$ possible permutations (corresponding to the $n(n-1)/2$ pairs of clusters we could merge,) we can compute deterministic $O(\log n)$ time routing schedules for each of them off-line [3]. These schedules are then indexed by the numbers of the clusters that are merged. Thus, we have an efficient parallel algorithm for general algorithms on a butterfly network, but we now require computing $O(n^2)$ routing schedules off-line (each of which requires $O(n \log n)$ time to compute) and we need memory polynomial in n to store the schedules on each processor.

5.2 Algorithm 2

The second algorithm that we will use is based on the sequential algorithm in Section 2.2:

1. Create a data structure in which we can determine nearest neighbors easily.
2. Pick any cluster i_1.
3. Determine the list $i_2 = NN(i_1)$, $i_3 = NN(i_2)$,... until $i_k = NN(i_{k-1})$ and $i_{k-1} = NN(i_k)$, where $NN(x)$ is the nearest neighbor of cluster x.
4. Agglomerate clusters i_{k-1} and i_k.
5. Update the data structure.
6. If more than one cluster still exists goto Step 3 using the previous i_{k-2} as the new i_1 if $k > 2$, otherwise choose arbitrary i_1.

Steps 1 and 2 are performed only once. Over the entire algorithm, we will need to perform no more than $3n$ nearest neighbor computations for Step 3 (each cluster is examined only once until it is agglomerated or its nearest neighbor is agglomerated, but each agglomeration causes only two clusters in the nearest neighbor chain to have new nearest neighbors due to its construction, assuming the metric satisfies the reducibility property.) Steps 4-6 are performed $n - 1$ times each and are trivial. So, the key to fast execution of this algorithm is determining the nearest neighbors quickly.

Note that the we could use this algorithm instead of algorithm 1 for the case of the single link metric on a PRAM (the median and centroid metrics don't satisfy the reducibility property,) but this algorithm requires a larger constant factor, since we will have to perform up to $3n$ minimizations rather than $2n$ in algorithm 1.

5.2.1 Minimum variance metric

For the minimum variance metric, we can simply store each of the cluster centers (on each processor in a local memory parallel computer) as our data structure and generate the distance to each cluster as necessary. The nearest neighbor of a cluster can then be determined by minimization on the distances of the other clusters to that cluster. So, we obtain an $O(n)$ time algorithm on an n node CRCW PRAM or an $O(n \log n)$ algorithm on an $\frac{n}{\log n}$ node EREW PRAM, butterfly, or tree.

5.2.2 Average and complete link metrics on a PRAM

The average and complete link metrics can be handled in a similar manner on a PRAM except that we must now explicitly store each of the intercluster distances in a two-dimensional array. Note that this will not work on a local memory machine, since in that case we must specify where each distance is held. If a single processor stores all of the distances for a cluster then all of the distances for the new agglomerated cluster must be stored on some processor, and we won't be able to update the data structure efficiently. We could use the skewed memory arrangement describe above, but this results in an $O(n^2 \log n)$ work algorithm as in the case of general metrics.

5.3 Algorithm 3

The third algorithm is for the special case of the single link metric on a local memory computer. Here we can use a variant on the parallel minimum spanning tree algorithm given by Driscoll *et al.* [9]. This minimum spanning tree can then easily be transformed into the cluster hierarchy [22, 18]. Since we are dealing with the special case of the complete graph on n vertices, it turns out that we do not need to use relaxed heaps to obtain an optimal algorithm as Driscoll *et al.* do.

1. Create a data structure to keep track of the distance of each point from the current minimum spanning tree. (At the start of the algorithm an arbitrary point comprises the minimum spanning tree.)

2. Find the point p not in the minimum spanning tree that is closest to the minimum spanning tree.

3. Add p to the minimum spanning tree.

4. Update the data structure.

5. If any points are not in the minimum spanning tree goto Step 2.

The data structure used can simply be an array of distances of each cluster from the minimum spanning tree. This array is distributed across the processors such that the processor responsible for the cluster stores the distances for that cluster. In Step 1, this array is set to the array of distances of each point from the arbitrary starting point.

Step 2 is performed by minimization on the values in this array. To update the data structure, the location of point p is broadcast and the distance of each point from the minimum spanning tree is simply the minimum of the previous distance and the distance to point p.

If we use $\frac{n}{\log n}$ processors (each responsible for $\log n$ points,) this algorithm requires $O(n \log n)$ time since Steps 2 and 4 require $\log n$ time and are performed n times.

6 Summary

I have considered parallel algorithms for hierarchical clustering using several inter-cluster distance metrics and parallel computer networks (see Table 2.)

Network	Distance Metric	Time	Processors	Work
Sequential	Single Link	$O(f(n,d))^a$	1	$O(f(n,d))^a$
	Average Link Complete Link Centroid Median Minimum Variance	$O(n^2)$	1	$O(n^2)$
	Lance-Williams	$O(n^2 \log n)$	1	$O(n^2 \log n)$
CRCW PRAM	Single Link Average Link Complete Link Centroid Median Minimum Variance	$O(n)$	n	$O(n^2)$
EREW PRAM, Butterfly or Tree	Single Link Centroid Median Minimum Variance	$O(n \log n)$	$\frac{n}{\log n}$	$O(n^2)$
EREW PRAM or Butterfly	Lance-Williams Average Link Complete Link	$O(n \log n)$	n	$O(n^2 \log n)$

[a] The best known complexity derives from the time to find the Euclidean minimal spanning tree, which can be computed in $O(n^{2-a(d)} \log^{1-a(d)} n)$ time where $a(d) = 2^{-d-1}$ [27], but this algorithm is impractical for most purposes. The best practical algorithms for $d > 2$ use $O(n^2)$ time.

Table 2: Summary of worst case running times on various networks.

The algorithms for each of the clustering metrics have achieved optimal efficiency on an n processor CRCW PRAM. I have also given optimal efficiency algorithms on a butterfly and/or tree for each metric examined except the average link and complete link metrics and for a general class of distance metrics. Due to the nature of the average link and complete link metrics, I hypothesize optimal parallel performance is not possible for them.

Acknowledgements

The author thanks Jitendra Malik and Abhiram Ranade for their advice on aspects of this research. This research has been supported by a National Science Foundation

Graduate Fellowship to the author, NSF Presidential Young Investigator Grant IRI-8957274 to Jitendra Malik, and NSF Materials Handling Grant IRI-9114446.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algoriths*. Addison-Wesley, 1974.

[2] D. H. Ballard. Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111–122, 1981.

[3] V. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, 1965.

[4] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected time algorithms for closest point problems. *ACM Transactions on Mathematical Software*, 6:563–580, 1980.

[5] M. Bruynooghe. Classification ascendante hiérarchique, des grands ensembles de données: un algorithme rapide fondé sur la construction des voisinages réductibles. *Les Cahiers de l'Analyse de Données*, III:7–33, 1978. Cited in [17].

[6] W. H. E. Day and H. Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1(1):7–24, 1984.

[7] D. Defays. An efficient algorithm for a complete link method. *Computer Journal*, 20:364–366, 1977.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[9] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, November 1988.

[10] J. H. Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58:236–244, 1963.

[11] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[12] G. N. Lance and W. T. Williams. A general theory of classificatory sorting strategies. 1. hierarchical systems. *Computer Journal*, 9:373–380, 1967.

[13] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgon Kaufmann, 1991.

[14] X. Li and Z. Fang. Parallel clustering algorithms. *Parallel Computing*, 11:275–290, 1989.

[15] S. Linnainmaa, D. Harwood, and L. S. Davis. Pose determination of a three-dimensional object using triangle pairs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(5), September 1988.

[16] F. Murtagh. Expected-time complexity results for hierarchical clustering algorithms which use cluster centers. *Information Processing Letters*, 16:237–241, 1983.

[17] F. Murtagh. A survey of recent advances in hierarchical clustering algorithms. *Computer Journal*, 26:354–359, 1983.

[18] F. Murtagh. *Multidimensional Clustering Algorithms.* Physica-Verlag, 1985.

[19] C. F. Olson. Time and space efficient pose clustering. Technical Report UCB//CSD-93-755, University of California at Berkeley, July 1993.

[20] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[21] E. M. Rasmussen and P. Willett. Efficiency of hierarchical agglomerative clustering using the ICL distributed array processor. *Journal of Documentation*, 45(1):1–24, March 1989.

[22] F. J. Rohlf. Algorithm 76: Hierarchical clustering using the minimum spanning tree. *Computer Journal*, 16:93–95, 1973.

[23] F. J. Rohlf. A probabilistic minimum spanning tree algorithm. *Information Processing Letters*, 7:44–48, 1978.

[24] R. Sibson. SLINK: An optimally efficient algorithm for the single link cluster method. *Computer Journal*, 16:30–34, 1973.

[25] G. Stockman. Object recognition and localization via pose clustering. *Computer Vision, Graphics, and Image Processing*, 40:361–387, 1987.

[26] D. W. Thompson and J. L. Mundy. Three-dimensional model matching from an unconstrained viewpoint. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 208–220, 1987.

[27] A. C. Yao. On constructing minimum spanning trees in k-dimensional space and related problems. *SIAM Journal on Computing*, 4:21–23, 1982.