Appeared in *Parallel Computing*, 21:1313-1325, 1995.

Parallel Algorithms for Hierarchical Clustering

Clark F. Olson[1]
Computer Science Department
Cornell University
Ithaca, NY 14853
clarko@cs.cornell.edu

Abstract

Hierarchical clustering is a common method used to determine clusters of similar data points in multidimensional spaces. $O(n^2)$ algorithms are known for this problem [3, 4, 10, 18]. This paper reviews important results for sequential algorithms and describes previous work on parallel algorithms for hierarchical clustering. Parallel algorithms to perform hierarchical clustering using several distance metrics are then described. Optimal PRAM algorithms using $\frac{n}{\log n}$ processors are given for the average link, complete link, centroid, median, and minimum variance metrics. Optimal butterfly and tree algorithms using $\frac{n}{\log n}$ processors are given for the centroid, median, and minimum variance metrics. Optimal asymptotic speedups are achieved for the best practical algorithm to perform clustering using the single link metric on a $\frac{n}{\log n}$ processor PRAM, butterfly, or tree.

Keywords. Hierarchical clustering, pattern analysis, parallel algorithm, butterfly network, PRAM algorithm.

1 Introduction

Clustering of multidimensional data is required in many fields. One popular method of performing such clustering is *hierarchical clustering*. This method starts with a set of distinct points, each of which is considered a separate cluster. The two clusters that are closest according to some metric are *agglomerated*. This is repeated until all of the points belong to one hierarchically constructed cluster. The final hierarchical cluster structure is called a *dendrogram* (see Figure 1), which is simply a tree that shows which clusters were agglomerated at each step. A dendrogram can easily be broken at selected links to obtain clusters of desired cardinality or radius. This representation is easy to generate and store, so this paper will concentrate on the determination of which clusters to merge at each step.

We must use some metric to determine the distance between pairs of clusters. For individual points, the Euclidean distance is typically used. For clusters of points, there are a number of metrics for determining the distances between clusters. The distance metrics can be broken into two general classes, graph metrics and geometric metrics.

1. **Graph metrics.** Consider a completely connected graph where the vertices are the points we wish to cluster and the edges have a cost function that is the Euclidean

[1]Supported in part by the National Science Foundation under Grants IRI-8957274 and IRI-911446.

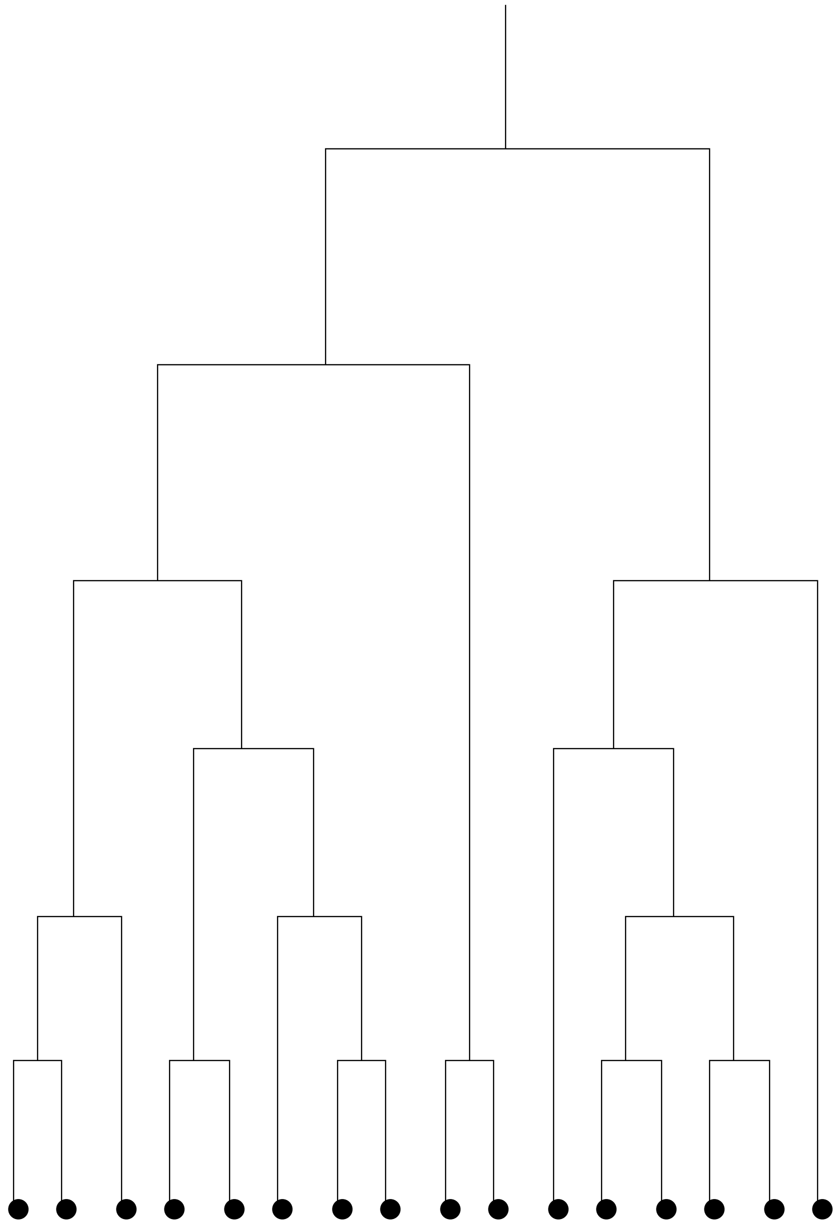


Figure 1: A dendrogram shows how the clusters are merged hierarchically.

distance between the points. The graph metrics determine intercluster distances according to the cost functions of the edges between the points in the two clusters. The common graph metrics are:

- Single link: The distance between two clusters is given by the minimum cost edge between points in the two clusters.
- Average link: The distance between two clusters is the average of all of the edge costs between points in the two clusters.
- Complete link: The distance between two clusters is given by the maximum cost edge between points in the two clusters.

2. **Geometric metrics.** These metrics define a cluster center for each cluster and use these cluster centers to determine the distances between clusters. Examples include:

   - Centroid: The cluster center is the centroid of the points in the cluster. The Euclidean distance between the 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 the 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 amount of increase in the sum of squared distances from each point to the center of its cluster that would be caused by agglomerating the clusters.

Useful clustering metrics can usually be described using the Lance-Williams updating formula [8]. The distance from the new cluster $i+j$ to any other cluster k is given by:

$$d(i+j,k) = a(i)d(i,k) + a(j)d(j,k) + bd(i,j) + c\left|d(i,k) - d(j,k)\right|$$

Table 1 gives the coefficients in the Lance-Williams updating formula for the metrics described above. $O(n^2)$ time algorithms exist to perform clustering using each of these metrics [3, 4, 11, 18]. Any metric that can be described by the Lance-Williams updating formula can be performed in $O(n^2 \log n)$ time [3].

This paper reviews several important sequential algorithms and discusses previous work on parallel algorithms for hierarchical clustering. Optimal algorithms are given for hierarchical clustering using several intercluster distance metrics and parallel architectures.

2 Sequential Algorithms

Several important results on sequential hierarchical clustering algorithms are summarized in [10]. Additional results are presented in [3]. This section describes some of these results. In each of the following algorithms, $D(i,j)$ will represent the distance between clusters i and j and $N(i)$ will represent the nearest neighbor of cluster i.

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

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

```
Function cluster-single-link(input: point-set)
  For each {i, j : 0 ≤ i < j ≤ n} compute D(i, j).
  For each {i : 0 ≤ i ≤ n} compute N(i).
  Repeat n − 1 times:
    Determine i, j such that D(i, j) is minimized.
    Agglomerate clusters i and j.
    Update each D(i, j) and N(i) as necessary.
End
```

Figure 2: An efficient algorithm to perform single link clustering.

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 and they require the same computational complexity since the minimal spanning tree can easily be transformed into the cluster hierarchy. While $o(n^2)$ algorithms exist to find the Euclidean minimal spanning tree [20], these algorithms are impractical when the dimensionality of the cluster space $d > 2$.

Figure 2 gives a practical algorithm for the single link metric. Computing arrays storing each $D(i,j)$ and $N(i)$ requires $O(n^2)$ time. Given these arrays, computing the two closest clusters can be performed $O(n)$ time by examining each cluster's nearest neighbor. To agglomerate the clusters, we simply store which clusters were agglomerated and update the arrays. The single link metric has the property that if we agglomerate clusters i and j into $i + j$, any cluster that had either i or j as its nearest neighbor now has $i + j$ as its nearest neighbor. We will call this the *same agglomerative nearest neighbor property* or *SANN property*. See Figure 3. The SANN property allows us to update the arrays in $O(n)$ time for the single link metric, yielding an $O(n^2)$ time algorithm.

To perform clustering using other metrics, the updating step is more complicated, since the SANN property does not hold. Determining the new nearest neighbors when using these metrics requires $O(n)$ time each. For the centroid and median metrics, Day and

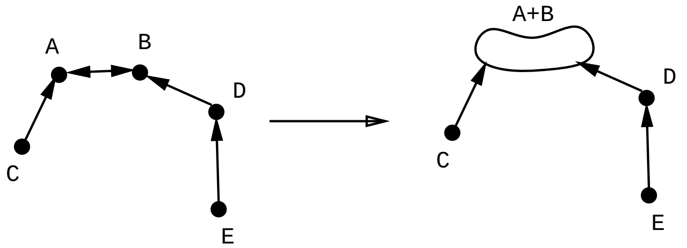


Figure 3: Nearest neighbors for a set of clusters before and after an agglomeration. When the SANN property holds, the nearest neighbors remain the same, with the new cluster taking the place of both of the clusters that were agglomerated to form it.

Edelsbrunner [3] have shown that the number of clusters for which we need to determine the new nearest neighbor is bounded by a function that is $O(\min(2^d, n))$, which for constant d is $O(1)$. Thus, for these metrics, the algorithm still requires $O(n)$ per iteration to update the arrays and $O(n^2)$ time overall.

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 and generate the distances as needed.) Sibson [18] gives an $O(n^2)$ time algorithm for the single link case requiring $O(n)$ space and Defays [4] gives a similar algorithm to perform complete link clustering.

2.2 Metrics satisfying the reducibility property

For metrics that satisfy the *reducibility property* [10] we can perform clustering in $O(n^2)$ time by computing nearest neighbor chains. The reducibility property requires that when we agglomerate clusters i and j, the new cluster $i+j$ cannot be closer to any cluster than both clusters i and j were. Formally, 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 all of the graph 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 (see Figure 4). This algorithm still provides a good approximate algorithm for these cases, but the order of examining the points can change the final hierarchy.

Figure 5 gives the algorithm. This algorithm works by following a nearest neighbor chain until a pair of mutual nearest neighbors are found and then agglomerating them. See Figure 6. By amortizing the cost of determining the nearest neighbors, it can be seen that

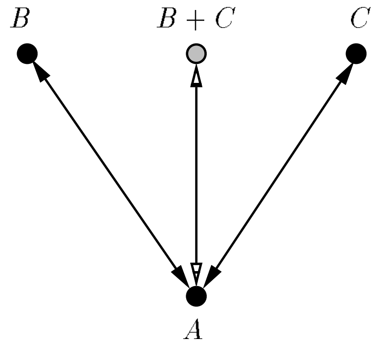


Figure 4: The centroid and median metrics do not satisfy the reducibility property. In this case, when clusters B and C are agglomerated, the center of the new cluster is closer to A than either of the previous clusters.

this algorithm requires $O(n^2)$ time and $O(n)$ space for clustering techniques using cluster centers. It can also be used to perform clustering using graph theoretical metrics by keeping an array 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 updating formula), clustering can be performed in $O(n^2 \log n)$ time using the algorithm in Figure 8 [3].

In this algorithm, we use priority queues to determine the nearest neighbor of each cluster. These can be implemented as heaps and thus each can be generated in $O(n)$ time. At each iteration of the loop, we determine the closest pair of clusters in $O(n)$ time by examining the head of each priority queue. We must then create a new priority queue for the agglomerated cluster, remove a cluster from each queue and update the distance to a cluster in each queue. This step requires $O(n \log n)$ time and is performed $O(n)$ times. The total time required is thus $O(n^2 \log n)$.

3 Previous Parallel Algorithms

Several authors have previously examined parallel algorithms for hierarchical clustering. In addition, there has been much recent work on parallelizing partitional clustering algorithms (another popular type of clustering.) See, for example, [14, 16, 21].

Rasmussen and Willett [15] discuss parallel implementations of clustering using the single link metric and the minimum variance metric on a SIMD array processor. They have implemented parallel versions of the SLINK algorithm [18], Prim's minimal spanning tree algorithm [13], and Ward's minimum variance method [19]. 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 factor speedup is

Function cluster-reducible-metrics(input: point-set)
  Pick $0 \leq c_1 \leq n$ at random.
  $i := 1$.
  Repeat $n-1$ times:
    Repeat:
      $i := i+1$.
      Determine $c_i = N(c_{i-1})$.
    until $c_i = c_{i-2}$.
    Agglomerate clusters c_i and $N(c_i)$.
    if $i > 3$
      i:=i-3.
    else
      i:=1.
End

Figure 5: An algorithm to efficiently perform clustering using metrics that satisfy the reducibility property.

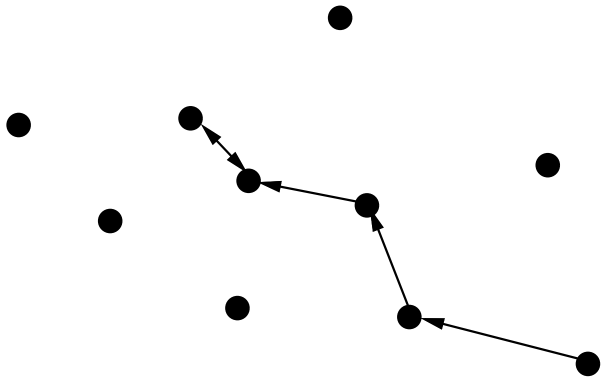


Figure 6: The algorithm for reducible metrics determines which clusters to merge by following a nearest neighbor chain to its end.

Function cluster-general(input: point-set)
  For each point i:
    Generate a priority queue P_i of the distances to each other point.
  Repeat $n-1$ times:
    Determine the closest two clusters i and j.
    Agglomerate clusters i and j.
    Update each P_i.
End

Figure 7: An algorithm to perform clustering efficiently in the general case.

achieved. Their parallel implementation of Prim's minimal spanning tree algorithm achieves $O(n \log n)$ time with sufficient processors.

Li and Fang [9] describe algorithms for hierarchical clustering using the single link metric on an n-node hypercube and an n-node butterfly. Their algorithms are parallel implementations of Kruskal's minimal spanning tree algorithm [7] 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 appear to have a fatal flaw causing incorrect 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 algorithms in a straightforward manner, the times required by their algorithms increase to $O(n^2)$.

Driscoll *et al.* [6] have described a useful data structure called the relaxed heap and they have shown how it can be applied to the parallel computation of minimal spanning trees. The relaxed heap is a data structure for manipulating priority queues that allows deleting the minimum element to be performed in $O(\log n)$ time and the decreasing the value of a key to be performed in $O(1)$ time. The use of this data structure allows the parallel implementation of Dijkstra's minimal spanning tree algorithm [5] in $O(n \log n)$ time using $\frac{m}{n \log n}$ processors on a PRAM, where n is the number of vertices in the graph and m is the number of edges.

Bruynooghe [2] describes a parallel implementation of the nearest neighbors clustering algorithm suitable for a parallel supercomputer. At each step, this algorithm dispatches tasks to determine the nearest neighbor of each cluster in parallel and then agglomerates each pair of reciprocal nearest neighbors in parallel. While this will achieve a speedup of the algorithm in most cases, some cases require $n-1$ iterations each of which require $O(n)$ time. The worst-case complexity is thus the same as the sequential algorithm.

4 Parallel Algorithms

This section discusses parallel implementations for each for the metrics on PRAMs and butterflies. In these algorithms, each cluster is the responsibility of one processor. When two clusters are agglomerated, the processor with the lower number of the two takes responsibility for the new cluster. If the other processor no longer has any clusters in its responsibility, it becomes idle.

4.1 Single link metric

We can use a parallel version of the algorithm in Figure 2 to efficiently perform clustering using the single link metric on a PRAM. The intercluster distance and nearest neighbor arrays are easily computed $O(\frac{n}{p})$ time, where p is the number of processors. For each iteration of the loop, we find the minimum of the nearest neighbor distances. The indexes of these clusters are then broadcast. Each processor updates the distances from the clusters that it is responsible for to the new cluster. Since the SANN property holds for the single link metric, we can update the nearest neighbors efficiently. The nearest neighbor of the new cluster is determined by finding the cluster with the minimum distance to it.

```
Function minimal-spanning-tree(input: point-set)
  Pick 0 ≤ i ≤ n at random.
  For each 0 ≤ j ≤ n:
    A(j) :=the distance between points i and j.
  Repeat n − 1 times:
    Find the minimum A(j).
    Add j to the minimal spanning tree.
    For each k not in the minimal spanning tree:
      A(k) :=min(A(k), D(j, k)).
End
```

Figure 8: An efficient algorithm to determine the minimal spanning tree of a set of points on a butterfly.

If we use $\frac{n}{\log n}$ processors (each processor is responsible for $\log n$ clusters), we can perform the broadcast and minimization operations in $O(\log n)$ time on a PRAM. The entire algorithm thus requires $O(n \log n)$ time using $\frac{n}{\log n}$ processors, which is optimal.

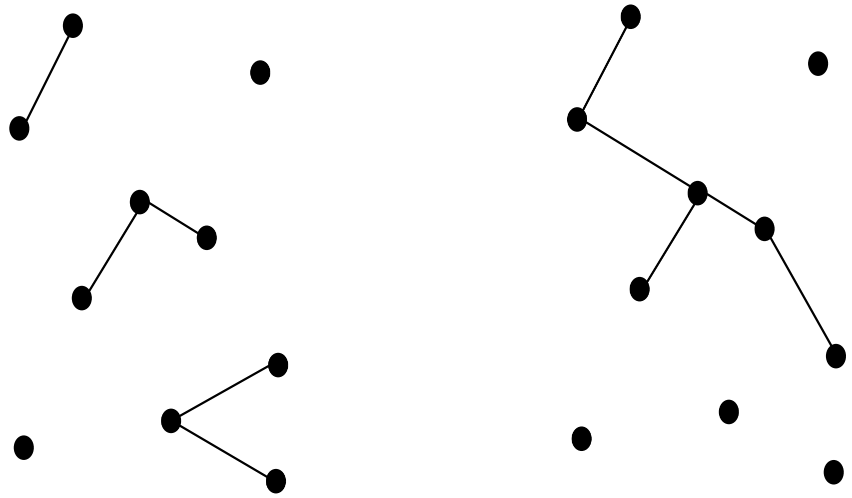
This algorithm is not efficient on a butterfly or tree since the distance from the new cluster to each of the other clusters would be stored on the same processor, requiring $O(n)$ time to update at each step. For this case, we can use a variant of the parallel minimal spanning tree algorithm given by Driscoll *et al.* [6] (see Figure 7). If we use $\frac{n}{\log n}$ processors, the loop requires $O(\log n)$ time to find the minimum $A(j)$ and thus $O(n \log n)$ time overall on a butterfly or tree. The minimal spanning tree can then easily be transformed into the cluster hierarchy [11, 17]. Figure 9 shows how the trees are built in these algorithms.

4.2 Centroid and median metrics

The single link PRAM algorithm can also be used for the centroid and median metrics with small modifications. When each pair of clusters is agglomerated, we now need to determine the new nearest neighbor of each of the clusters that had one of the agglomerated clusters as its nearest neighbor. Day and Edelsbrunner [3] have shown that there are $O(1)$ clusters that can have any specific cluster as a nearest neighbor for these metrics. We can write the indexes of the clusters that need new nearest neighbors to a queue on some processor. We then iterate through this queue, broadcasting the index of each cluster and determining the new nearest neighbor using a minimization operation. We can thus perform this algorithm in $O(n \log n)$ time using $\frac{n}{\log n}$ processors on a PRAM, butterfly or tree.

4.3 Minimum variance metric

For the minimum variance metric, we will use a parallel version of the algorithm in Figure 5. We must perform the inner loop (determining the nearest neighbor of a cluster) no more than $3n$ times and this step dominates the computation time. If we use $\frac{n}{\log n}$ processors, we can simply store the location of the center of each cluster on each processor and find the nearest neighbor in $O(\log n)$ on a PRAM, butterfly or tree by performing minimization on



(a) (b)

Figure 9: The minimal spanning tree is built differently in the two algorithms. Shown is a point set after 5 agglomerations for both cases. (a) The PRAM algorithm generates a forest of trees and agglomerates the closest two clusters at each step. (b) The butterfly algorithm generates a single tree and adds the closest cluster to it at each step.

the distance from the cluster in question to the clusters that each processor is responsible for. The algorithm thus requires $O(n \log n)$ and is optimal.

4.4 Average and complete link metrics

For the average and complete link metrics on a PRAM, we can use the same algorithm as for the minimum variance metric, except that we must now keep an explicit array of the intercluster distances. This array can be updated in $O(\log n)$ time per agglomeration.

This algorithm does not achieve optimality for the average and complete link metrics on a butterfly or tree, since in this case we must specify which processor stores each distance. If each processor stores all of the distances for the clusters it is responsible for, then all of the distances for the agglomerated cluster will be stored on the same processor and we will not be able to update the array efficiently. The following subsection gives the best known algorithm for this case.

4.5 General metrics

For general metrics, we have no bound on the number of clusters that may have any particular cluster as a nearest neighbor. In addition, we cannot use the nearest neighbor chain algorithm since the metric may not satisfy the reducibility property. In this case, we can employ a parallel variant of the algorithm in Figure 7 for any metric where the new intercluster distance can be determined in constant time, given the old intercluster distances (i.e. using the Lance-Williams updating formula.)

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 10: Distribution of D values between processors. This example shows the case for $p = n = 4$. (a) Straightforward distribution (b) Skewed distribution of values.

We can create a priority queue in $O(n)$ time on a single processor and in $O(\log n)$ time on a n processor PRAM, but we must now update each priority queue after each agglomeration, a step that requires $O(\log n)$ time. This algorithm thus requires $O(n \log n)$ time on an n processor PRAM, but since the sequential algorithm for general metrics required $O(n^2 \log n)$ time, this is optimal.

We cannot use this algorithm naively on local memory machines, since we cannot store all of the distances to a single cluster on the same processor (each agglomerated cluster would require $O(n)$ work on a single processor at each step). We can use a new storage arrangement to facilitate this implementation (see Figure 10). 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 for each cluster that it is responsible for.

Updating the data structure now requires more work, since each updated distance is a function of three intercluster distances in the Lance-Williams updating 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. General permutation routing on a butterfly requires $O(\log^2 n)$ time in the worst case, but since we only need to consider $O(n^2)$ 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 [1]. 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 and sufficient memory to store the schedules on each processor.

5 Summary

We have considered parallel algorithms for hierarchical clustering using several intercluster distance metrics and parallel computer architectures. Table 2 summarizes the complexities achieved. In addition to the results discussed here, $O(n)$ time algorithms for n processor

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

† 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 $a(d) = 2^{-d-1}$ [20], 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 architectures.

CRCW PRAMs exist for each metric except for the general case [12]. We have achieved optimal efficiency for each metric on a PRAM and for the single link, centroid, median, and minimum variance metrics on a butterfly or tree. Due to the nature of the average link and complete link metrics, we hypothesize that optimal parallel performance is not possible for them on parallel architectures with local memory.

References

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

[2] M. Bruynooghe. Parallel implementation of fast clustering algorithms. In *Proceedings of the International Symposium on High Performance Computing*, pages 65–78, 1989.

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

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

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

[6] 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.

[7] 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.

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

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

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

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

[12] C. F. Olson. Parallel algorithms for hierarchical clustering. Technical Report UCB//CSD-94-786, Computer Science Division, University of California at Berkeley, January 1994.

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

[14] S. Ranka and S. Sahni. Clustering on a hypercube multicomputer. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):129–137, April 1991.

[15] 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.

[16] F. F. Rivera, M. A. Ismail, and E. L. Zapata. Parallel squared error clustering on hypercube arrays. *Journal of Parallel and Distributed Computing*, 8:292–299, 1990.

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

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

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

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

[21] E. L. Zapata. Parallel fuzzy clustering on fixed size hypercube SIMD computers. *Parallel Computing*, 11:291–303, 1989.