Efficient Parallel Hierarchical Clustering

Manoranjan Dash[1], Simona Petrutiu[2], and Peter Scheuermann[2]

[1] Department of Information Systems, School of Computer Engineering, Nanyang Technological University, Singapore 639798
[2] Department of Electrical & Computer Engineering, Northwestern University, Evanston, IL 60208*

Abstract. Hierarchical agglomerative clustering (HAC) is a common clustering method that outputs a dendrogram showing all N levels of agglomerations where N is the number of objects in the data set. High time and memory complexities are some of the major bottlenecks in its application to real-world problems. In the literature parallel algorithms are proposed to overcome these limitations. But, as this paper shows, existing parallel HAC algorithms are inefficient due to ineffective partitioning of the data. We first show how HAC follows a rule where most agglomerations have very small dissimilarity and only a small portion towards the end have large dissimilarity. Partially overlapping partitioning (POP) exploits this principle and obtains efficient yet accurate HAC algorithms. The total number of dissimilarities is reduced by a factor close to the number of cells in the partition. We present pPOP, the parallel version of POP, that is implemented on a shared memory multiprocessor architecture. Extensive theoretical analysis and experimental results are presented and show that pPOP gives close to linear speedup and outperforms the existing parallel algorithms significantly both in CPU time and memory requirements.

Keywords: hierarchical agglomerative clustering, partitioning, parallel algorithm, shared memory architecture.

1 Introduction

Hierarchical agglomerative clustering (HAC) is often used in various applications due to its capability to output a dendrogram showing all agglomerations. Unlike K-means and other types of clustering where objects are clustered into a given number of clusters, a dendrogram can be used to get any number of clusters. HAC algorithms are non-parametric, natural and simple in grouping objects, and capable of finding clusters of different shapes by using different similarity measures. However, they are limited in their application to real-world problems mainly due to high CPU time and memory complexities. Existing algorithms take $O(N^2 \log N)$ CPU time and require $O(N^2)$ memory. Parallel algorithms

* Research of the third author on this project was supported by NSF grant IIS-0325144.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 363–371, 2004.

are proposed to alleviate this limitation. Existing parallel algorithms either parallelize other clustering methods such as K-means (Dhillon and Modha [1]) and subspace clustering (Nagesh *et al.* [2]), or are not very efficient due to lack of performance enhancing partitioning [3].

In [4] we have shown that complexities of the existing sequential HAC algorithms can be reduced significantly by an efficient partitioning scheme without losing accuracy. The proposed methods are based on an observation that in HAC most iterations agglomerate very small clusters separated by very small dissimilarity. Only a small number of iterations towards the end agglomerate the large clusters. Using this observation a structure called *p*artially *o*verlapping *p*artitioning (POP) divides the data into a number of overlapping cells. Analysis and experiments showed that POP-based sequential HAC algorithms reduce existing time and memory complexities by a factor close to the number of cells c.

In this paper we present parallel versions of POP, called pPOP. Due to the independent nature of each partitioned cell, parallelization is able to achieve similar reduction in time and memory complexities as POP, i.e., by a factor close to the number of cells c. We implement pPOP over a shared memory architecture. Experimental evaluations show that for large data sets pPOP obtains near linear speedup. In addition, for stored matrix implementations, pPOP results in a two order of magnitude improvement in computation time over the existing parallel HAC algorithms.

2 Background

Let us assume that there are N objects each with M attributes. We use real type data and Euclidean (L_2) distance to measure dissimilarity. Other distance measures, e.g. *Manhattan*, can be used (see [4]).

The 90-10 Rule: In an experiment we ran the centroid type HAC method over a 2-D data set with 100 clusters and some noise. In the centroid type, each cluster is represented by a centroid and the pair with the closest centroids is merged in each iteration. In Figure 1, we plot the closest pair distance for each iteration. Notice that most agglomerations except for a small portion towards the end have very small closest pair distance compared to the maximum closest pair distance. This maximum distance is taken over all agglomerations. If we plot the size of clusters merged in an iteration it also shows a similar plot. We experimented with many data sets having varying characteristics. For varying M, N (typically large – at least a few thousand objects), and K (number of clusters), the general trend is as follows: *if a majority of the objects are inside clusters then the shape of the distance plot is as shown in Figure 2.* We name this as '*90-10 rule*' to convey the idea that *in a dendrogram, most levels from the bottom merge pairs of very small clusters separated by a very small portion of the maximum closest pair distance.* The 90-10 rule extends to other HAC algorithms beyond the centroid method for both the geometric and the graph metrics. For space constraints, we restrict all discussions in this paper to centroid method.

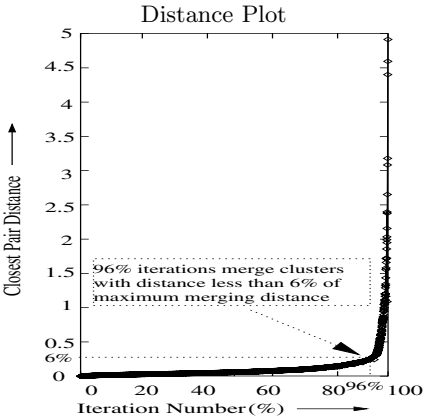

Fig. 1. An important property of HAC: the distance plot shows that the closest pair distance is very small even until last stage of agglomeration.

See [4] for detailed discussion on the 90-10 rule and other metrics. Next we show how to exploit this inherent characteristic of HAC.

2.1 Partially Overlapping Partitioning (POP)

An axis-parallel POP divides the data-space uniformly into c number of overlapping cells. The overlapping region is called δ-region where δ is the overlapping distance between two cells. Figure 2 depicts the axis-parallel POP. For the centroid metric (and other geometric metrics), if the representative point of a cluster falls in a δ-region then each affected cell that contains this δ-region holds it, otherwise only one cell holds it.

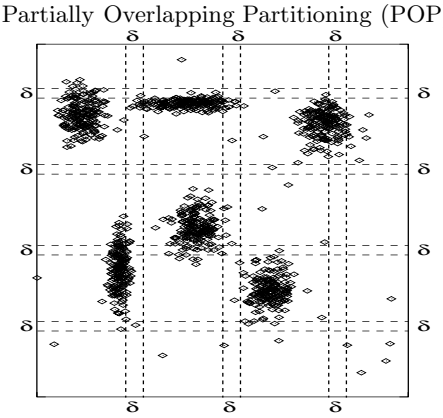

Fig. 2. The 90-10 rule is exploited by POP for efficient HAC.

Before discussing POP any further, we very briefly describe some existing HAC algorithms. HAC algorithms are mainly of two types: stored matrix (e.g., dissimilarity matrix and priority queues) and stored data (e.g., nearest neighbor). The *dissimilarity matrix method* stores dissimilarities between each pair of clusters. When a pair is merged dissimilarities are computed for the new cluster and the matrix is updated. The memory complexity of this method is $O(N^2)$ and the time complexity is $O(N^3)$. In the *priority queue* method a heap-based priority queue is maintained for each cluster. Because a priority queue requires $O(\log n)$ time for each insert and delete operation for n elements, the time complexity reduces to $O(N^2 \log N)$ although the memory complexity stays at $O(N^2)$. The *nearest neighbor array* method maintains nearest neighbors for each cluster in an array. If after each iteration the average number of clusters whose nearest neighbors need to be changed is α, then the time complexity reduces to $O(\alpha N^2)$ and the memory complexity reduces to $O(N)$. An upper bound for α is $2(3^M - 2)$. When memory is enough to store $O(N^2)$ dissimilarities, stored matrix algorithms are preferred as they do fewer computations. Otherwise, the stored data type is preferred.

2-Phase Algorithm: In [4] we proposed a new 2-phase algorithm for HAC based on the axis-parallel POP. In *phase 1* clusters are partitioned into c overlapping cells. The basic idea is that in each iteration the closest pair is found for each cell and from those the overall closest pair is found. If the overall closest pair distance is less than δ then the pair is merged and the priority queues (or the dissimilarity matrix or the nearest neighbor array) of only the container cell are updated. If the closest pair or the merged cluster is in a δ-region then the priority queues of the affected cells are also updated. *Phase 1* terminates when the closest pair distance exceeds δ. *Phase 2* merges the remaining clusters of *phase 1* using the existing algorithm, thus completing the dendrogram. **Accuracy**: POP in *phase 1* ensures that *any pair with distance less than δ must reside together in at least one cell*. Hence, as *phase 2* is the existing algorithm itself, the 2-phase algorithm guarantees the correct dendrogram. **Complexity Analysis**: By setting δ to the closest pair distance at the turning point of the distance plot (see Figure 1), a large number of small clusters are merged in *phase 1* while only a small number of larger clusters are merged in *phase 2*. Recall that phase 1 uses POP which is very efficient whereas phase 2 uses the existing algorithm which is not so efficient. In Figure 1, if δ is set to the turning point of the distance plot, 96% agglomerations from the beginning are merged in phase 1 and the remaining 4% in phase 2. Therefore, the overall computational time is reduced drastically. So, we see that when δ is set to the turning point, the number of clusters remaining (k') for phase 2 is very small and the total number of clusters in the δ-region ($|\delta|$)is also very small. For simplification of the complexity analysis, we consider k' and $|\delta|$ to be negligible. This is reasonable because the 90-10 rule holds for all data sets that have clusters in it. We assume equal cell size and equal δ-region size for each cell. In [4] we give the detailed complexity analysis comparison between the existing and the 2-phase algorithms. Following is a brief overview of that. Stored matrix type that requires $O(N^2)$ memory now requires

$O(\frac{N^2}{c})$ in the 2-phase algorithm. Hence memory is reduced by a factor close to c. Because of this reduction, the 2-phase dissimilarity matrix algorithm, whose time complexity is dominated by the time required to create the matrix, enjoys a reduction by a factor close to c. The time complexity of the priority queue algorithm is dominated by the update effort required to maintain the priority queues. After each agglomeration of the closest pair, the priority queues of all other clusters are updated. But in the 2-phase algorithm this effort is restricted only to the cell that holds the closest pair, and if it happens to be in a δ-region then it is restricted only to the affected cells. So after simplification the reduction factor is $\log_{\frac{N}{c}} N \times c$, i.e., the time complexity reduces from $O(N^2 \log N)$ to $O(\frac{N^2}{c} \log \frac{N}{c})$. In stored data type there is no reduction in the memory complexity of $O(N)$. The time complexity is dominated by the time required to update the nearest neighbors of the affected clusters. For the existing algorithm the time required to find the nearest neighbor of one affected cluster is $O(N)$ but for the 2-phase algorithm it is $O(\frac{N}{c})$. So, the overall reduction factor is close to c.

Setting δ and c – Nested Algorithm: The performance of the 2-phase algorithm depends on c and δ. As shown in the distance plot of Figure 1, there exists an ideal δ at the turning point at which the total time taken by the 2-phase algorithm is minimum. But it is not straightforward to compute. So, we adopted a nested approach where in the beginning POP partitioning starts with a very small δ and gradually increases it until a few or just one cluster remain. As δ increases, c which is set initially to a high value, is gradually reduced. *Accuracy* of this nested algorithm is assured from the accuracy of the 2-phase algorithm. Experiments show that the nested algorithm is more efficient than the 2-phase algorithm even when δ is set ideally for the 2-phase algorithm. For example, for the data set described in Section 2, the minimum time for the 2-phase algorithm is 125.4 cpu sec while the nested algorithm takes only 57.8 cpu sec.

Higher Dimensional Data: The above discussion focuses on 2-D data. For higher dimensions we proposed a very efficient data structure as a replacement for the axis-parallel partitioning. Due to space constraint we limit the scope of this paper to 2-D and refer the interested reader to [4].

3 pPOP Algorithms

Parallel HAC algorithms have been studied by Li [5], Li and Fang [6], Olson [3], and Wu *et al.* [7]. The common feature of these algorithms is: for 'stored matrix' type the task of computing and maintaining $O(N^2)$ dissimilarities is divided among the processors, whereas for 'stored data' type the task of computing and maintaining the $O(N)$ nearest neighbors is divided among the processors. For example, Olson used p processors to reduce the time complexity of the dissimilarity matrix method to $O(\frac{N^3}{p})$ and that of the priority queue method to $O(\frac{N^2 * \log N}{p})$ [3]. The time complexity for the nearest neighbor array method

reduces to $O(\frac{\alpha * N^2}{p})$. These algorithms are not very efficient because they still require $O(N^2)$ total memory for 'stored matrix' type, and in each iteration they require to update all the priority queues or dissimilarity matrix. For 'stored data' type the existing methods need to check all the clusters after each agglomeration to determine whether the newly merged cluster is nearer than the previous nearest. So, the reduction in these parallel algorithms is mostly because of parallelization, but not due to efficient partitioning.

The advantage of pPOP is that each cell is sufficient by itself, and hence parallelization benefits by dividing the task of creating and maintaining the dissimilarities or priority queues or nearest neighbors of each *cell* among the processors. This reduces the total computation of searching for the closest pair and maintaining the data structure drastically. Below we give the complexities of sequential, existing parallel and pPOP algorithms. For complexity analysis we select the 2-phase algorithm of the stored matrix type since, as we shall show later, this algorithm achieves larger speedups compared to the existing algorithms. As before, we assume equal cell sizes, negligible $\|\delta\|$ size, and negligible phase 2 time. Among existing algorithms, those described by Olson [3] are selected. The number of processors is denoted by p.

Table 1. Comparison of time complexities of sequential, existing parallel, and pPOP algorithms. RF - Reduction Factor ($= \frac{ExistingParallel}{pPOP}$).

Priority Queues	Sequential	Existing Parallel	pPOP	RF
1. Create priority queues	$O(N^2)$	$O(\frac{N^2}{p})$	$O(\frac{N^2}{cp})$	
2. for $n = N$ to 2	$O(N)$	$O(N)$	$O(N)$	
3. find smallest distance	$O(n)$	$O(\frac{n}{p})$	$O(\frac{n}{p})$	
4. merge and update P	$O(n * \log n)$	$O(\frac{n * \log n}{p})$	$O(\frac{n * \log \frac{n}{c}}{p})$	
Overall	$O(N^2 * \log N)$	$O(\frac{N^2 * \log N}{p})$	$O(\frac{N^2 * \log \frac{N}{c}}{p})$	$\frac{\log N}{\log \frac{N}{c}}$
Overall (Dissimilarity Matrix)	$O(N^3)$	$O(\frac{N^3}{p})$	$O(\frac{N^3}{cp})$	c

In Table 1 (priority queues) step 1 of pPOP computes priority queues in $O(\frac{N^2}{cp})$ time. Recall that pPOP reduces the memory by a factor of c, i.e., $O(\frac{N^2}{c})$. pPOP divides the total computation for the c cells among p processors, and hence, assuming no synchronization delays the complexity becomes $O(\frac{N^2}{cp})$. Step 4 updates the priority queues of the affected clusters. In pPOP a priority queue holds $\frac{N}{c}$ elements in the beginning. Hence, due to parallelization the total time complexity of this step is $O(\frac{n * \log \frac{n}{c}}{p})$. So, the overall reduction factor is $\frac{\log N}{\log \frac{N}{c}}$. Table 1 shows the overall complexities for the dissimilarity matrix type as well. It has a reduction factor close to c. The memory requirement for priority queues and dissimilarity matrix types is reduced by a factor close to c. For the nearest neighbor type, the gain of pPOP over the existing parallel algorithms cannot be obtained directly from the complexity analysis. For the step where each cluster

is checked to find whether it is affected by the agglomeration, pPOP needs to do it for one (or a few, if in $|\delta|$-region) cell whereas the existing algorithm needs to do it for all clusters. Similarly, the existing algorithm needs to check all the clusters to find the new nearest neighbor of each affected cluster. But pPOP requires only the container cell to be checked. Experimental results in the next section show that pPOP outperforms the existing algorithms substantially for all the above three types of HAC.

4 Experimental Results

We performed a number of experiments to study the performance and scalability of our proposed pPOP algorithms. Both stored matrix (priority queues) and stored data (nearest neighbors) types of pPOP were implemented using the 2-phase algorithm. For comparison purposes we implemented the corresponding existing parallel algorithms, hereby denoted as existing algorithms. These are described in [3]. The performance was measured in terms of CPU time, memory space and speedup. We experimented using several real, benchmark, and artificial data sets. Due to space constraint we show the results over an artificial data set that is used in [8]. Other results are available from www.ece.northwestern.edu/∼manoranj/research.html. The experiments were run on the SGI Origin2000 multiprocessor system which is a shared memory machine consisting of 8 R10000 CPUs running at clock rate of 195MHs. The secondary cache size is 4MB. We used OpenMP which is an API for directed based parallel programming applications in a shared memory environment [9]. We decided to use it because it is designed for fine-grained parallelism, which was predominant in our algorithm.

The pPOP implementation in OpenMP uses guided self scheduling clause in the assignment of iterations to threads, i.e., processors. During each iteration of HAC each processor is assigned in turn a chunk of cells to work on, with the chunk size being reduced as we proceed with the iteration. After an iteration is finished, a critical region is established in order to find the overall closest pair of clusters and merge them. The priority queues of the cells affected by the agglomeration can be updated in parallel.

In Figure 3 we show the results over the synthetic data set whose size varies from 3K to 60K. The existing stored matrix algorithms require $O(N^2)$ memory, hence we could experiment only with a data size up to 5K; on the other hand for pPOP we report results for data sets up to 30K. The number of processors varies from 1 to 8. In Figure 3 (a-b) we report the speedups of pPOP. Although the speedup of pPOP is small for smaller data sets, we observe that for larger data sets (30K or higher) the speedup of pPOP improves substantially and approaches linear speedup for data sets of 60K. Figure 3 (c-d) gives the relative speedup of pPOP over the existing algorithm. pPOP is always superior over the existing algorithm because of its efficient partitioning, and independent nature of each cell. The relative speedup increases with data size. Among stored matrix and stored data types, pPOP's performance is much better for stored matrix. It

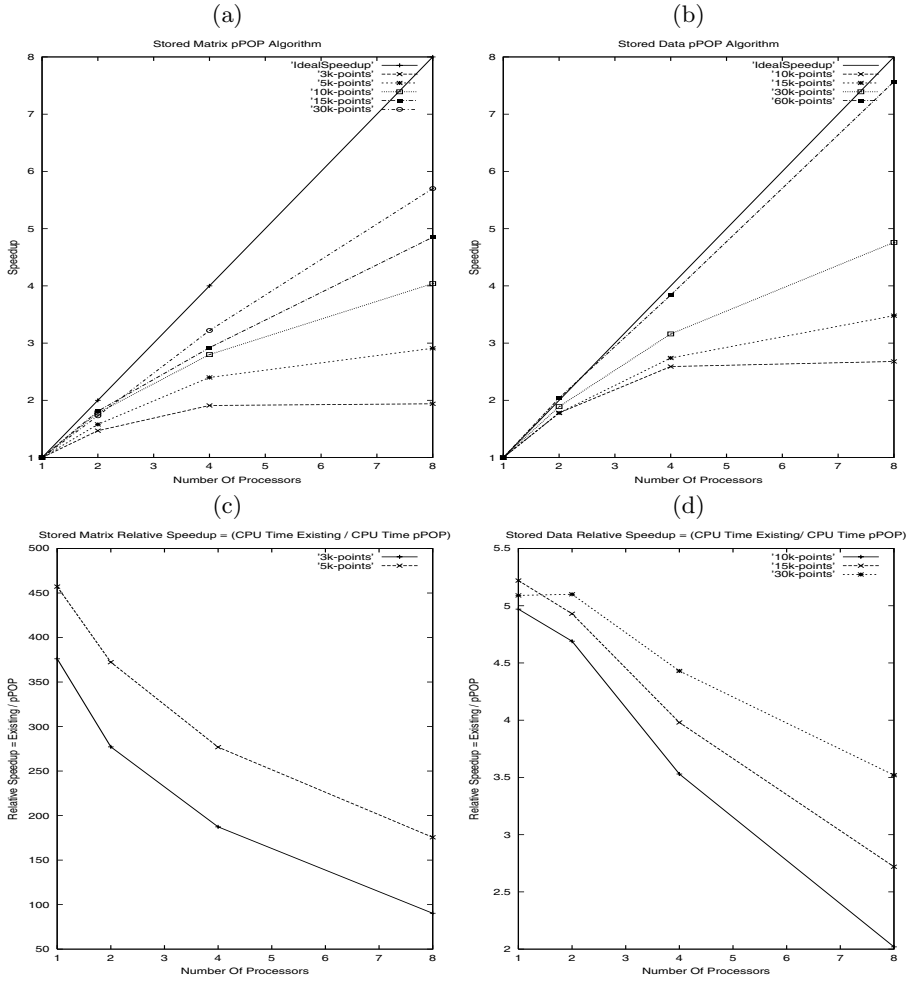

Fig. 3. Synthetic data results: For stored matrix and stored data types, and for varying #processors (1 to 8), (a-b) show performance of pPOP, and (c-d) show $RelativeSpeedUp = \frac{Existing}{pPOP}$.

achieves a two order of magnitude improvement in computation time over the existing algorithm.

As shown in Figures 3 (c-d) the relative speedup, $\frac{ExistingCPU}{pPOPCPU}$, decreases as the number of processors increases. This is due to the fact that for a small number of cells, when the number of processors is increased, some processors end up working on cells containing a very small number of clusters, and will therefore spend a lot of time being idle when they are done with the computation in a given iteration. However, as the data set size increases and/or the number of clusters increases, load balancing among the processors becomes better. This phenomena can be observed in our figures. Although for both 3K and 5K sizes for stored

matrix type the relative speedup drops by approximately the same amount (285) when the number of processors increased from 1 to 8, the noticeable fact is that relative speedup for 1 processor for 3K size is 375, but that for 5K size is 460. That is to say as the number of processors increased, with increasing size of data the *rate of drop in speedup* decreased. Although due to the high memory requirement of the existing parallel algorithms we could not test for higher data sizes, we postulate that for larger data sets this trend of reduction in relative speedup for more processors will continue to slow down further.

We compared the memory for stored matrix type. For 3K and 5k pPOP reduced the memory requirement by a factor of 97 and 189 respectively. For stored data type both algorithms require similar amount of memory.

5 Conclusion and Future Directions

In this paper we proposed pPOP for efficient parallel HAC. Analysis and experiments showed that, for both stored matrix and stored data types, pPOP outperforms the existing algorithms significantly both in CPU time and memory requirements. This is achieved by exploiting a 90-10 rule of HAC which states that in a dendrogram, most levels from the bottom merge pairs of very small clusters separated by a very small portion of the maximum closest pair distance. The data space was partitioned by partially overlapping cells each of which could be processed independent of other such cells without affecting accuracy. Future work includes parallelizing the high-dimensional data structure.

References

1. Dhillon, I.S., Modha, D.M.: Large-scale parallel data mining. Lecture Notes in Artificial Intelligence **1759** (2000) 245–260
2. Nagesh, H., Goil, S., Choudhary, A.: PMAFIA: A scalable parallel subspace clustering algorithm for massive datasets. In: Proc. International Conference on Parallel Processing. (2000) 21–24
3. Olson, C.F.: Parallel algorithms for hierarchical clustering. Parallel Computing **21** (1995) 1313–1325
4. Dash, M., Liu, H., Scheuermann, P., Tan, K.L.: Fast hierarchical clustering and its validation. Data and Knowledge Engineering **44(1)** (2003) 109–138
5. Li, X.: Parallel algorithms for hierarchical clustering and cluster validity. IEEE Transactions on Pattern Analysis and Machine Intelligence **12** (1990) 1088–1092
6. Li, X., Fang, Z.: Parallel clustering algorithms. Parallel Computing **11** (1989) 275–290
7. Wu, C.H., Horng, S.J., Tsai, H.R.: Efficient parallel algorithms for hierarchical clustering on arrays with reconfigurable optical buses. Journal of Parallel and Distributed Computing **60** (2000) 1137–1153
8. Zhang, T., Ramakrishnan, R., Livny, M.: BIRCH: An efficient data clustering method for very large databases. In: Proceedings of ACM SIGMOD Conference on Management of Data, Montreal, Canada (1996) 103–114
9. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R., eds.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers (2000)