Exploiting Parallelism to Support Scalable Hierarchical Clustering

Rebecca J. Cathey, Eric C. Jensen, Steven M. Beitzel,* Ophir Frieder, and David Grossman
Information Retrieval Laboratory, Department of Computer Science, Illinois Institute of Technology, 10 W. 31st Street, Chicago, IL 60616. E-mail: {cathey, jensen, beitzel, frieder, grossman}@ir.iit.edu

A distributed memory parallel version of the group average hierarchical agglomerative clustering algorithm is proposed to enable scaling the document clustering problem to large collections. Using standard message passing operations reduces interprocess communication while maintaining efficient load balancing. In a series of experiments using a subset of a standard Text REtrieval Conference (TREC) test collection, our parallel hierarchical clustering algorithm is shown to be scalable in terms of processors efficiently used and the collection size. Results show that our algorithm performs close to the expected $O(n^2/p)$ time on $p$ processors rather than the worst-case $O(n^2)$ time. Furthermore, the $O(n^2/p)$ memory complexity per node allows larger collections to be clustered as the number of nodes increases. While partitioning algorithms such as $k$-means are trivially parallelizable, our results confirm those of other studies which showed that hierarchical algorithms produce significantly tighter clusters in the document clustering task. Finally, we show how our parallel hierarchical agglomerative clustering algorithm can be used as the clustering subroutine for a parallel version of the buckshot algorithm to cluster the complete TREC collection at near theoretical runtime expectations.

Introduction

Document clustering has long been considered as a means to potentially improve both retrieval effectiveness and efficiency; however, the intensive computation necessary to cluster the entire collection makes its application to large datasets difficult. Accordingly, there is little work on effectively clustering entire large, standard-text collections and less with the intent of using these clusterings to aid retrieval. Rather, much work has focused on either performing simplified clustering algorithms or only using partial clusterings such as clustering only the results for a given query.

Clustering algorithms generally consist of a trade-off between accuracy and speed. Hierarchical agglomerative clustering algorithms calculate a full document-to-document similarity matrix. Their clusterings are typically viewed as more accurate than other types of clusterings; however, the computational complexity required for the algorithm’s quadratic behavior makes it unrealistic for large document collections. Other clustering algorithms such as the $k$-means and single pass algorithms iteratively partition the data into clusters. Although these partitioning algorithms run in linear time, the assignment of documents to moving centroids produces different clusterings with each run. Some algorithms combine the accuracy of hierarchical agglomerative algorithms with the speed of partitioning algorithms to get an algorithm that is fast with reasonable accuracy. One such algorithm is the buckshot algorithm, which uses a hierarchical agglomerative algorithm as a clustering subroutine.

We propose a hierarchical agglomerative clustering algorithm designed for a distributed memory system in which we use the message passing model to facilitate interprocess communication (Gropp, Lusk, & Skjellum, 1996; Snir, Otto, Huss-Lederman, Walker, & Dongarra, 1997). Our algorithm has an expected time of $O(\frac{n^2}{p^2})$ time on $p$ processors. Although partitioning algorithms generally have a linear time complexity, our focus is on the quality of the clusters. We show how a distributed system can be used to produce accurate clusters in a reasonable amount of time. Since we use an optimized serial hierarchical agglomerative clustering algorithm, our actual speed-up is $O(\frac{n^2}{p^2})$, half of the expected speed-up. That is, only half of the symmetric matrix is used by an optimized serial algorithm. In our parallel approach, to reduce internode communication, we process a complete matrix. Hence, our two node instances require roughly the same processing as the optimized serial version. The speed-up, however, is consistent and does not decay. Namely, from two nodes onward, doubling the nodes roughly halves runtime. We determine the quality of our clusters by comparing them with clusters generated using a bisection variant of the partitioning $k$-means algorithm. Furthermore, we show how our hierarchical agglomerative clustering algorithm can be used...
as a highly accurate clustering subroutine in the buckshot algorithm to facilitate clustering of larger document collections. The buckshot algorithm results in near-optimal speed-up.

**Background and Prior Work**

Although many clustering techniques are currently available (Fasulo, 1999), there are two main categories of approaches: partitioning and hierarchical clustering. Partitioning assigns every document to a single cluster iteratively (Duda & Hart, 1973; Hartigan, 1975) in an attempt to determine \( k \) partitions that optimize a certain criterion function (Guha, Rastogi, & Shim, 1998). Partitioning algorithms do not require every document to be compared to every other document; rather, they compare every document to a set of centroids which must be initialized through some external means (often randomly). For this reason, these algorithms commonly run in \( O(kn) \) time, where \( k \) is the number of desired clusters.

A hierarchical clustering is a sequence of partitions in which each partition is nested into the next partition in the sequence. Hierarchical clusterings generally fall into two categories: splitting and agglomerative methods. Splitting methods work in a top-down approach to split clusters until a certain threshold is obtained. The more popular agglomerative clustering algorithms use a bottom-up approach to merge documents into a hierarchy of clusters (Jardine & van Rijsbergen, 1971). Agglomerative algorithms typically use a stored matrix or stored data approach (Dash, Petrutiu, & Sheuermann, 2004). The stored matrix approach creates a similarity matrix to keep track of document-to-document similarity measurements. Stored matrix approaches include similarity matrix and priority queues. Similarity matrix methods use a matrix to store the document to document similarities in a similarity matrix. The matrix is searched to find the clusters that have the highest similarity. When those clusters are merged, the similarities in the matrix also are updated. The total time complexity for the similarity matrix method is \( O(n^2) \) time. This can be reduced to \( O(n^2 \log n) \) time using heap-based priority queues.

The priority queue method maintains a priority queue for each cluster. When a new cluster is found, a new priority queue is created, and all other priority queues are updated. A priority queue requires \( O(\log n) \) time for inserts and deletes. Each priority queue is updated by performing two deletes and one insert resulting in \( O(n \log n) \) time for \( n \) priority queues. Thus, the time reduces to \( O(n^2 \log n) \) time (Day & Edelsbrunner, 2005). Both the similarity matrix and priority queue methods require a memory complexity of \( O(n^2) \); however, note that since the priority queue method also must store document identifiers, it requires over double the memory of the similarity matrix method. Stored data approaches require the recaclulation of the similarity measurements for each time clusters are merged. The nearest neighbor method uses the stored data approach to store an array of nearest neighbors for each cluster. When the number of values that need to be changed after each iteration is \( \alpha \), the time complexity is \( O(\alpha n^2) \), and the memory complexity is \( O(n) \). When the memory is enough to store \( O(n^2) \) similarity values, the stored matrix approach is preferred as it performs less similarity computations; otherwise, the stored data approach is preferred (Dash et al., 2004).

The main difference between hierarchical and partitioning methods is the assignment of documents to clusters.

With hierarchical clustering, once a document is assigned to a cluster it remains in that cluster. Partitioning algorithms often move documents among clusters to obtain the final result. Some studies have found that hierarchical agglomerative clustering algorithms, particularly those that use group-average cluster-merging schemes, produce better clusters, purportedly because of their complete document-to-document comparisons (Dubes & Jain, 1988; Larsen & Aone, 1999; Willet, 1988). More recent work has indicated that this may not be true across all metrics and that some combination of agglomerative and partitioning algorithms can outperform either one or the other individually (Steinbach, Karypis, & Kumar, 2000; Zhao & Karypis, 2002b). As these studies use a variety of different experiments, using different metrics and (often very small) document collections, it is difficult to conclude which clustering method is “definitively” superior, but they do agree that hierarchical agglomerative clustering is an effective choice.

There exist several algorithms that combine the accuracy of the hierarchical approach with the lower time complexity of the partitioning approach to form a hybrid approach. A popular algorithm for accomplishing this is the buckshot algorithm, which combines a hierarchical agglomerative clustering algorithm performed on a subset of the collection with a partitioning algorithm (Cutting, Karger, Pedersen, & Tukey, 1992). This reduces the computational complexity to \( O(kn) \) time (Zamir & Etzioni, 1998); however, this sequential algorithm is still very slow for today’s large collections. Even the most simplistic modern clustering algorithms are often too slow for real-time applications (Schütze & Silverstein, 1997).

There has been work done to develop scalable algorithms for clustering. A scalable clustering approach has three main aspects (Ghosh, 2003). The first aspect is scalability to a large number of documents. Linear algorithms as well as a minimum number of collection scans are desirable for large collections of data stored in secondary storage. Bradley, Fayyad, and Reina (1998) minimized the number of scans by using the \( k \)-means algorithm, with a limited memory buffer to store summaries of the documents already scanned. Ordonez and Omiecinski (2004) used a relational database to store the dataset, generally reducing the number of disk scans to three. Another approach to deal with large document collections is to run the clustering algorithm on a sample of the dataset or data summaries instead of the entire collection (Chen & Liu, 2004; Guha et al., 1998; Nassar, Sander, & Cheng, 2004; Zhang, Ramakrishnan, & Livny, 1996). These methods can be used to compress very large data collections into representative points that can be used to hierarchically cluster data.
The second aspect is scalability to a large number of attributes or dimensions. High-dimensional data have properties that inhibit the performance of algorithms that work well with low dimensions. Because text data are high-dimensional data, much work has gone into selecting the correct features (Duda, Hart, & Stork, 2001; Globerson & Tishby, 2003; Mao & Jain, 1995). He, Cai, Liu, and Ma (2004) represented the document as a low-dimensional vector from a compact representation subspace. A δ tree index, where the number of dimensions increases toward the leaf level, has been used to speed up processing of high-dimensional k-nearest neighbor queries (Cui, Ooi, Su, & Tan, 2003). Orlandic, Lai, and Yee (2005) used a data-reduction method that represents the data space as a set of dense cells.

The third aspect is in number of processors, ideally splitting the total computation involved into p equal parts. Work in this area involves the parallelization of several algorithms. Dhillon and Modha (2000) used a parallel k-means algorithm to create up to 16 clusters from generated test collections of documents having 8 to 128 terms in length, the largest of which was 2 GB. In addition, Dhillon, Fan, and Guan (2001) multithreaded the spherical k-means partitioning algorithm and achieved near linear speed-up and scale-up when running on 113,716 National Science Foundation award abstracts averaging 72 terms in length after term filtering. Some work on parallel hierarchical agglomerative clustering exists, but most of these algorithms have large computational overhead or have not been evaluated for document clustering (Guha et al., 1998; Li, 1990; Rajasekaran, 2005; Zhang et al., 1996). Our approach addresses scalability primarily with respect to the number of nodes.

Document clustering is a unique clustering task because of its immense and sparse feature space. Typical clustering studies that focus on a small number of features are not applicable to the document clustering task. Dash et al. (2004) examined a parallel hierarchical agglomerative clustering algorithm based on dividing the data into partially overlapping partitions. Experiments showed that the sequential algorithm reduced existing time and memory complexities; however, a small number of dimensions was used as the focus was not on document clustering. Some prior work developed parallel algorithms for hierarchical document clustering; however, these algorithms require specialized interconnection networks (Li, 1990; Olson, 1995; Wu, Horng, & Tsai, 2000). Ruocco and Frieder’s (1997) single-pass partitioning algorithm showed near linear speed-up on subsets of the Tipster document collection, the largest of which contained 10,000 documents. These collections have the disadvantage of being small as compared to today’s collections.

Prior work has gone into using document clustering to improve retrieval effectiveness. Salton (1975; Salton & McGill, 1983) performed experiments on changing the spatial density of a document collection using clustering with the vector space model. Xu and Croft (1999) described a method for improving distributed retrieval effectiveness using language models of clustered collections. More recently, models were presented by which retrieval effectiveness might be improved through modified term weighting in clustered document collections (Zhao & Karypis, 2002a). Query-time efficiency also can be improved through clustering given the additional collection metadata that it provides—namely, which documents are similar. This provides the opportunity to shortcut document retrieval.

Sequential Methods

Next, we discuss two algorithms for sequential document clustering. The first is a hierarchical agglomerative clustering algorithm, and the second is the buckshot algorithm that uses the hierarchical agglomerative clustering algorithm for the clustering subroutine.

Hierarchical Agglomerative Clustering

For hierarchical agglomerative clustering, each document is initially a separate cluster. The clusters are then merged in stages until the desired number of clusters are found. We use the sequential hierarchical agglomerative algorithm (Anderberg, 1973) shown in Figure 1. The complexities given for each step of the algorithm are relatively loose in terms of order; they provide an upper bound for the number of computations. This algorithm uses a stored matrix method to store an \( n \times n \) similarity matrix. In addition, two arrays of the nearest neighbor to each cluster and the corresponding maximum similarity measurement also are stored.

The hierarchical agglomerative clustering algorithm has two distinct phases. The first phase builds a similarity matrix and the nearest neighbor arrays for the entire collection of size \( n \). The similarity matrix contains the document-to-document similarity scores for the entire collection. The nearest neighbor to each cluster and the corresponding maximum similarity measurement are found using the similarity matrix and stored in two separate arrays. There are many techniques for calculating a measure of similarity between two documents (Grossman & Frieder, 2004). Although any similarity measure can be used, in our experimentation we use a cosine similarity measure (Salton, 1975; Salton & McGill, 1983) that includes document and query length normalization factors estimated from their number of unique terms (Lee, Chuang, & Seamons, 1997) coupled with a modern term-weighting scheme (Chowdhury et al., 2000). Since we calculate a similarity matrix for \( n \) documents and find the maximum values for each of \( n \) rows, the time complexity for this phase is \( O(n^2) \) time. A sample document-to-document similarity matrix for \( n = 6 \) documents is shown in Figure 2. Also shown are the arrays containing the nearest neighbors, \( m_{array} \) and the corresponding maximum similarity values, \( max_{array} \). The \( n_{array} \) is an array that contains the nearest neighbor for each cluster. The \( max_{array} \) is an array that contains the similarity score from each cluster to the nearest neighbor of that cluster. Each row in \( max_{array} \) and \( m_{array} \) corresponds to the same row and represented cluster in the original matrix. For example, the nearest neighbor to Cluster 1 is Cluster 6. Thus, 6 is placed in the first position of \( n_{array} \).
Similarly, the first position of \( \text{maxarray} \) contains the similarity score between Clusters 1 and 6—in this case 10.

In this simple example, the columns and rows are labeled with document identifiers, and the matrix is filled with similarity coefficient scores. In practice, when using cosine and other popular similarity measures, the scores are very often real values between 0 and 1. For simplicity, these scores are represented here as integers. A memory-efficient sequential implementation of the hierarchical agglomerative clustering algorithm requires only approximately \( \frac{n^2}{2} \) entries (Rounding to whole numbers is left out for simplicity throughout.) in the similarity matrix, as the matrix is symmetrical over the diagonal.

The final phase of the hierarchical agglomerative clustering algorithm is to create clusters from the \( n \) documents. Once the document-to-document similarities for the \( n \) documents are known, each document is assigned to a cluster, resulting in \( n \) clusters each containing one item. The similarity measurements between the clusters are the same as the similarity measurements between the items they contain. The closest pair of clusters, \( i \) and \( j \), are merged into a single cluster, \( h \). The similarity measurements between \( h \) and every other cluster are recalculated, and the similarity matrix is updated. We use a variation of the Voorhees (1986) method to calculate the group average similarity between two clusters. The similarity between the new cluster \( h \) and any arbitrary cluster \( c \) can be found using Equation 1. Once the matrix is updated, the nearest neighbor arrays are updated. Whenever, the nearest neighbor of a cluster is \( i \) or \( j \), the corresponding row in the similarity matrix is searched to find the newest nearest neighbor and maximum similarity, which are used to update \( \text{nnarray} \) and \( \text{maxarray} \). Assuming \( \alpha \) updates are performed, this step runs in \( O(\alpha n) \) time. The final phase is repeated until a specified threshold is obtained. Different thresholds can be used to determine when to stop clustering. We use the number of clusters, \( k \), as a threshold.

\[
sim(h, c) = \frac{\sum_{i \in h} \sim(i, c) + \sum_{j \in c} \sim(j, c)}{|i||j|}
\]  

(1)

The computational complexity of the sequential hierarchical agglomerative clustering algorithm is both \( O(n^3) \) and \( \Omega(n^2) \) (Day & Edelsbrunner, 2005). In a worst-case scenario, when \( \alpha = n \), the algorithm runs in \( O(n^3) \) time; however, Anderberg (1973) theorized that \( \alpha \) averages a constant number of updates per iteration. In our experiments, we found that \( \alpha \) was generally a constant number significantly less than \( n \), making the expected time complexity \( O(n^2) \). The memory complexity of this algorithm is \( O(n^2) \), as it stores the entire \( n \times n \) similarity matrix.

**Buckshot Approach**

The buckshot algorithm is a combination of hierarchical and partitioning algorithms designed to take advantage of
the accuracy of hierarchical clustering as well as the low computational complexity of partitioning algorithms. The buckshot algorithm takes a random sample of \( s \) documents from the collection and uses the hierarchical agglomerative clustering algorithm as the high-precision clustering subroutine to find initial centers from this random sample. Traditionally, \( s = \sqrt{kn} \) to reduce the computationally complex task of hierarchical agglomerative document clustering can be used as the basis for clustering the entire collection in a high-performance manner, by assigning the remaining documents in the collection to the most appropriate initial center. The original buckshot algorithm gives no specifics on how best to assign the remaining documents to appropriate centers, although various techniques are given. We use an iterated assign-to-nearest algorithm with two iterations similar to the one discussed in the original proposal of the buckshot algorithm (Cutting et al., 1992).

The sequential buckshot clustering algorithm consists of the two phases shown in Figure 3. The first is to cluster \( s \) documents using the hierarchical agglomerative clustering algorithm. We use \( s = \sqrt{kn} \), where \( k \) is the number of desired clusters and \( n \) is the total number of documents to be clustered. The second phase iterates over the remaining \( n - s \) documents in the collection and assigns them to the appropriate clusters based on their similarities to the initial centers. For each document, the similarity to every cluster centroid is calculated to find the cluster that is most similar to the document. The document then is assigned to the most similar cluster. This is repeated until every document in the collection has been processed, taking \( O(s^2) \) time. Although, the second phase can be iterated indefinitely, the quality of the resulting clusters improves the most in the first few iterations. Thus, it is typically iterated only a small fixed number of times (Cutting et al., 1992). Our algorithm performs two iterations of the second phase.

Parallel Methods

Using a distributed architecture can reduce the time and memory complexity of the sequential algorithms by a factor of \( p \), where \( p \) is the number of nodes used. Here, we present our parallel version of the hierarchical agglomerative clustering algorithm. In addition, we present a parallel version of the buckshot algorithm, which uses our parallel hierarchical agglomerative clustering algorithm as the clustering subroutine.

Each communication is either a broadcast or gather performed via recursive-doubling algorithms implemented in the MPICH implementation of MPI (Gropp, Lusk, Doss, & Skjellum, 1996). The time for broadcast and gather are given in Equations 2 and 3 (Thakur & Gropp, 2003).

$$\text{broadcast: } O(C_{latency} + N_{byte}C_{transfer} \lg p) \quad (2)$$

$$\text{gather: } O(C_{latency} \lg p + N_{byte}C_{transfer}) \quad (3)$$

- \( C_{latency} \)—start-up cost of communicating.
- \( N_{byte} \)—number of bytes to be communicated.
- \( C_{transfer} \)—time required to transmit a single byte.

Parallel Hierarchical Agglomerative Clustering Algorithm

The first phase of the hierarchical agglomerative clustering algorithm is fairly straightforward to parallelize, as the data can be partitioned easily among nodes, and there is little need for communication or coordination. The main effort involves parallelizing the creation of the clusters via hierarchical agglomerative clustering. A single similarity matrix must be kept consistent among all nodes, which requires communication whenever updates are performed. Our proposed approach reduces the amount of necessary communication. The parallel hierarchical agglomerative clustering algorithm is shown in Figure 4. All parts other than those under the label \( P_{manager} \), indicating that they are executed only on the managing node of the new cluster, are executed on every node.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precondition: ( s = # ) of sample documents, ( n = ) document size, ( k=)desired # of clusters</td>
<td>( O(\alpha s^2) )</td>
</tr>
<tr>
<td>Phase 1: Cluster random set of documents</td>
<td>( O(\alpha s^2) )</td>
</tr>
<tr>
<td>1. Run Clustering Subroutine (see Figure 1) with ( s ) documents</td>
<td>( O(\alpha s^2) )</td>
</tr>
<tr>
<td>Phase 2: Group remaining documents</td>
<td>( O(kn) )</td>
</tr>
<tr>
<td>2. Calculate Centroids of ( k ) clusters</td>
<td>( O(s) )</td>
</tr>
<tr>
<td>3. Repeat for each document, ( d ), not initially clustered</td>
<td>( O(kn) )</td>
</tr>
<tr>
<td>3.1. Calculate similarity between ( d ) and each centroid ( c_i )</td>
<td>( O(k) )</td>
</tr>
<tr>
<td>3.2. Assign ( d ) to cluster ( i ) where ( \text{sim} \ (d, c_i) &gt; \text{sim} \ (d, c_j) ) for all ( i \neq j )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>4. Repeat Phase 2</td>
<td>( O(kn) )</td>
</tr>
<tr>
<td>5. Output ( k ) clusters</td>
<td></td>
</tr>
</tbody>
</table>

FIG. 3. Buckshot clustering algorithm.
Our parallel algorithm produces the same results as does a sequential implementation. We describe our parallel approach for each phase of the hierarchical agglomerative clustering algorithm in the following two sections.

**Phase 1: Build similarity matrix for** \( n \) **documents.** Each row in the document-to-document similarity matrix represents a document in the collection and the similarity scores relating it to every other document. By using row-based partitioning, we are able to assign each node approximately \( \frac{n}{p} \) rows of the matrix to “manage,” where \( p \) is the number of processing nodes. The managing node is responsible for calculating its initial section of the similarity matrix and maintaining the similarity scores during the clustering subroutine. In Figure 5, we illustrate our sample similarity matrix after partitioning it among three nodes: N1, N2, and N3. Also shown are the nearest neighbor and corresponding maximum similarity arrays. The data and the computational load for the matrix and the nearest neighbor arrays are evenly partitioned over the available nodes in the system.

By distributing the similarity matrix and nearest neighbor arrays in this fashion, the data and computational load are

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Phase 1: Build Similarity Matrix</strong></td>
<td>( O\left(\frac{n^2}{p}\right) )</td>
</tr>
<tr>
<td>1. Broadcast document IDs to all processors using the MPI_Bcast collective operation</td>
<td>( O(N \text{ bytes} \log p) )</td>
</tr>
<tr>
<td>2. In Parallel: Partition the document collection ( C ) into ( p ) collections of ( \frac{n}{p} ) documents each.</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>3. In Parallel: Load term vectors for all sample documents in partition ( p_i ) from disk into memory on each processor</td>
<td>( O\left(\frac{n}{p}\right) )</td>
</tr>
<tr>
<td>4. In Parallel: For each document ( d_i ) in ( {C - p_i} )</td>
<td>( O\left(\frac{n^2}{p}\right) )</td>
</tr>
<tr>
<td>4.1. In Parallel: Load term vector for ( d_i ) into memory</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>4.2. In Parallel: Calculate ( \text{sim}(d_i, d_j) ) for each ( d_i ) in ( p_i ), place in submatrix ( D )</td>
<td>( O\left(\frac{n^2}{p}\right) )</td>
</tr>
<tr>
<td>5. In Parallel: Each processor searches ( \frac{n}{p} ) rows of ( D ) to find the nearest neighbor and corresponding similarity for each cluster ( c_i ), place in ( \text{nn}<em>\text{array} ) and ( \text{max}</em>\text{array} )</td>
<td>( O(\log p) )</td>
</tr>
<tr>
<td>6. Gather the size of each processor’s matrix portion on all processors using the MPI_Allgather collective operation</td>
<td>( O\left(\frac{m^2}{p}\right) )</td>
</tr>
<tr>
<td><strong>Phase 2: Create</strong> ( k ) <strong>clusters</strong></td>
<td></td>
</tr>
<tr>
<td>7. Repeat ( n - k ) times</td>
<td></td>
</tr>
<tr>
<td>7.1. In Parallel: Each processor searches the respective partition of ( \text{nn}<em>\text{array} ) and ( \text{max}</em>\text{array} ) for clusters ( i ) and ( j ) with maximum similarity</td>
<td>( O\left(\frac{n}{p}\right) )</td>
</tr>
<tr>
<td>7.2. The MPI_Allgather collective operation is used to gather the maximum similarity from each processor</td>
<td>( O(p) )</td>
</tr>
<tr>
<td>7.3 In Parallel: Each processor determines clusters ( i ) and ( j ) with the maximum similarity.</td>
<td>( O(p) )</td>
</tr>
<tr>
<td>7.4 In Parallel: Each Processor determines the managing processor, ( p_{\text{manager}} ), responsible for the new cluster and updates the load count for each processor</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>7.5 ( p_{\text{manager}} ) searches through all rows to find an empty row and broadcasts the row number to all other processors via the MPI_Bcast collective operation</td>
<td>( O\left(\frac{n}{p}\right) )</td>
</tr>
<tr>
<td>7.6 In Parallel: Each processor iterates through the respective partition of ( D ), calculating the similarity between the clusters it manages and the new cluster, ( h ), using the group-average calculation.</td>
<td>( O\left(\frac{n}{p}\right) )</td>
</tr>
<tr>
<td>7.7 Each Processor sends new similarities to ( p_{\text{manager}} ) via MPI_Gather collective operation</td>
<td>( O(\log p) )</td>
</tr>
<tr>
<td>7.8 ( p_{\text{manager}} ) updates the respective partition of ( D ) with the new similarities</td>
<td>( O\left(\frac{n}{p}\right) )</td>
</tr>
<tr>
<td>7.9 In Parallel: Each processor updates the respective partition of ( \text{nn}<em>\text{array} ) and ( \text{max}</em>\text{array} )</td>
<td>( O\left(\frac{m}{p}\right) )</td>
</tr>
<tr>
<td>8. Output ( k ) clusters</td>
<td></td>
</tr>
</tbody>
</table>

FIG. 4. Parallel hierarchical agglomerative clustering algorithm.
nearly evenly partitioned among the available nodes in our system. Each node can perform its own updates and similarity calculations with a limited amount of communication. As stated earlier, efficient sequential implementations of the hierarchical agglomerative clustering algorithm only require the storage of one half of the symmetrical similarity matrix, consisting of \( \frac{e^2}{2} \) matrix entries instead of the full size of \( n^2 \). Our parallel approach requires the storage of the complete rows for the portion of the similarity matrix so that each node can find similarities between its managed clusters and the newly formed clusters with minimum communication during the clustering subroutine. If only half of the matrix is stored, there is a heavy cost associated with the communication required to fill in the missing pieces each time two clusters merge into one.

In Phase 1, Node 0 broadcasts the document IDs to all nodes. Once each node has received the document set, it proceeds with calculating similarity scores for each managed document to every other document in the collection. Once the similarity measurements are calculated, each node finds the nearest neighbor and corresponding similarity for each of the managed rows. Nodes manage the documents corresponding to their submatrix rows, which in turn correspond to an even, horizontal partitioning of the entire distributed matrix. The memory complexity for our parallel hierarchical agglomerative algorithm is \( O(n/\pi) \), allowing us to cluster increasingly large document collections as the number of nodes increases.

The complete algorithm including Phase 1 is shown in Figure 4. The total time taken to broadcast the document identifiers, read the documents into memory, calculate similarities for each node’s portion of the matrix, and find the similarities for each node’s portion of the matrix, and find the nearest neighbor and corresponding maximum similarity arrays for the clusters with the highest similarity. Single documents are viewed as clusters of Size 1. Once a node identifies the two most similar clusters, it notifies all other nodes in the system.

As the result of Phase 1 on our example, Node 1 broadcasts value 12, along with the two cluster identifiers, 2 and 3, that correspond to that similarity. Node 2 broadcasts 12 and its component cluster identifiers, and Node 3 broadcasts 11, and so on. Once each node has discovered the clusters that have the highest similarity over the entire matrix, it updates the respective portion of the similarity matrix to reflect the merge of the most similar clusters. This update operation involves several steps. First, a node must be selected to manage the new cluster. To enforce even cluster distribution and load balancing across nodes, the “managing node” for the new cluster is selected by keeping count of how many clusters are currently being managed by each node, and selecting the node with the smallest load. To avoid unnecessary communication, these counts are maintained on each node as merges take place. Ties are broken by assigning the node with the lowest rank to manage the new cluster. Once the managing node is selected, each node must update the similarity scores to the new cluster in each row of the respective portion of the similarity matrix. There are several methods of updating the similarity scores when a new cluster is formed. We used a variation of the group-average method to merge two clusters as defined in Equation 1.

In our example, N1 and N2 both had their loads reduced to one; however, N1 has the lower rank, so it is chosen to manage the new cluster. Each node updates the scores between the new cluster, created by merged Clusters 2 and 3, and each existing cluster. The matrix and arrays are updated as shown in Figure 6.

Note that both the individual Clusters 2 and 3 are no longer relevant in terms of the algorithm, as indicated by the dashes throughout. Nodes 1 and 2 are both underused due to the merge of Clusters 2 and 3; Node 1 is selected to manage the new cluster, as it has a lower rank. Once the managing node is identified, the first available empty row in the managing node’s submatrix is selected to hold the row for the new cluster. Consequently, all the similarity values between each

\[
\mathcal{O}\left(\left(C_{\text{latency}} + N_{\text{bytes}} C_{\text{transfer}}\right) \log p + \frac{n}{p} \cdot C_{\text{read}} + n \cdot \frac{n}{p} \cdot C_{\text{sim}} + n \cdot \frac{n}{p} \cdot C_{\text{compare}}\right) = \mathcal{O}\left(\frac{n^2}{p}\right)
\]  

- \( C_{\text{read}} \)—cost of reading a document from disk.
- \( C_{\text{sim}} \)—cost of calculating the similarity coefficient between two documents.
- \( C_{\text{compare}} \)—cost of comparing two numbers.

**Phase 2: A parallel clustering subroutine.** Each node is responsible for maintaining only a partition of the similarity matrix and nearest neighbor arrays. Therefore, the first phase in the cluster subroutine is for each node to scan the respective portion of the nearest neighbor and corresponding maximum similarity arrays for the clusters with the highest similarity. Single documents are viewed as clusters of Size 1. Once a node identifies the two most similar clusters, it notifies all other nodes in the system.

![Figure 5](image-url)  
**FIG. 5.** A partitioned similarity matrix (a), nearest neighbor array (b), and maximum similarity array (c).

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>N1</strong></td>
<td>1</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td><strong>N2</strong></td>
<td>3</td>
<td>8</td>
<td>12</td>
<td>6</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td><strong>N3</strong></td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>

![Figure 6](image-url)  
**FIG. 6.** A modified similarity matrix (a), nearest neighbor array (b), and maximum similarity array (c).
of the clusters and the new cluster are written into the corresponding location. This guarantees the consistency of the entries in the matrix for all nodes and avoids allocating extra storage space to append new columns and rows to the submatrix on each node.

Once each node calculates the similarity scores between the documents it manages and the newly created cluster, it sends them to the new cluster’s managing node. This allows the managing node to fill in the columns for the row in its portion of the similarity matrix that represents the newly formed cluster. In our example, Node 2 sends \{4, 15\} to Node 1 to populate the similarities. Node 3 sends \{5, 6\} and \{6, 18\} to Node 1. Once Node 1 collects the scores from each node and updates the respective partition of the matrix, the entire matrix has been updated. Once the matrix is updated, the nearest neighbor scores are assigned. The third phase of the buckshot algorithm assigns these documents according to their similarity to the centroids of the initial clusters. This phase of the algorithm is trivially parallelized via data partitioning. First, the initial cluster centroids are calculated on every node. This was done in favor of communication because the centroids are relatively large, with term vectors in size, making transmitting them a significantly larger cost than calculating all of them. Note that the effectiveness of load balancing in Phase 1 of our parallel hierarchical agglomerative clustering algorithm and Phase 2 of our parallel buckshot algorithm depends to some degree on each node being assigned documents of roughly similar length. The documents in the 2-GB TREC Disks 4 and 5 test collection have a mean length of 168 distinct terms, with a maximum of 23,818. Although this range is large, the SD for distinct term count in a document from this collection is 144, and only 3.2% of documents have a distinct term count more than 1 SD from the mean. In general, this problem is easily alleviated by using simple document metadata to ensure a balanced distribution over available nodes.

To achieve centroid calculation on every node, the document identifiers corresponding to each initial cluster are gathered onto every node using the MPI_Gather collective operation. After centroid calculation is complete, each node is assigned round robin approximately \(\frac{p}{n} \) documents to process. Each node iterates through these documents in place, by reading the term vector from disk, comparing it to each centroid, making the assignment, discarding the term vector, reading the next one, and so on until all documents are assigned. The third phase is iterated two times. The second iteration recalculates the centroids and reassigns all the documents to one of the \( k \) clusters. Once this process has completed, the document identifiers for each final cluster are gathered onto the root node for writing out to disk. The complete algorithm for our parallel buckshot algorithm is shown in Figure 7. There are no sequential components to Phase 3; the nodes only synchronize at completion to combine their clusters onto Node 0. The total time taken to calculate the centroids, read each remaining document, calculate the similarity to each centroid, and gather the cluster identifiers assigned to each cluster on each node onto Node 0 is shown in Equation 7. Combined with Phase 1 of the buckshot algorithm results in a time complexity of \(O\left(\frac{kn}{p}\right)\).

\[
O\left(\frac{1}{p}\sum_{i=1}^{s}C_{\text{read}} + \frac{n-s}{p}C_{\text{read}} + k \cdot \frac{C_{\text{sim}}}{p}ight) = O\left(\frac{kn}{p}\right) \tag{7}
\]

- \(C_{\text{read}}\)—cost of reading two document vectors during centroid calculation.
- \(C_{\text{read}}\)—cost of reading a document from disk.

Parallel Buckshot Algorithm

The first phase of the parallel buckshot algorithm uses our parallel hierarchical agglomerative clustering algorithm to cluster \( s \) random documents. The final phase for the parallel version of the buckshot algorithm groups the remaining documents in parallel. After the clustering subroutine has finished, \( k \) initial clusters have been created from the random sample of \( s = \sqrt{kn} \) documents. From the total collection, \( n - s \) documents remain that have not yet been assigned to any cluster. The third phase of the buckshot algorithm assigns these documents according to their similarity to the centroids of the initial clusters. This phase of the algorithm is trivially parallelized via data partitioning. First, the initial cluster centroids are calculated on every node. This was done in favor of communication because the centroids are relatively large, with term vectors in size, making transmitting them a significantly larger cost than calculating all of them. Note that the effectiveness of load balancing in Phase 1 of our parallel hierarchical agglomerative clustering algorithm and Phase 2 of our parallel buckshot algorithm depends to some degree on each node being assigned documents of roughly similar length. The documents in the 2-GB TREC Disks 4 and 5 test collection have a mean length of 168 distinct terms, with a maximum of 23,818. Although this range is large, the SD for distinct term count in a document from this collection is 144, and only 3.2% of documents have a distinct term count more than 1 SD from the mean. In general, this problem is easily alleviated by using simple document metadata to ensure a balanced distribution over available nodes.

To achieve centroid calculation on every node, the document identifiers corresponding to each initial cluster are gathered onto every node using the MPI_Gather collective operation. After centroid calculation is complete, each node is assigned round robin approximately \(\frac{p}{n} \) documents to process. Each node iterates through these documents in place, by reading the term vector from disk, comparing it to each centroid, making the assignment, discarding the term vector, reading the next one, and so on until all documents are assigned. The third phase is iterated two times. The second iteration recalculates the centroids and reassigns all the documents to one of the \( k \) clusters. Once this process has completed, the document identifiers for each final cluster are gathered onto the root node for writing out to disk. The complete algorithm for our parallel buckshot algorithm is shown in Figure 7. There are no sequential components to Phase 3; the nodes only synchronize at completion to combine their clusters onto Node 0. The total time taken to calculate the centroids, read each remaining document, calculate the similarity to each centroid, and gather the cluster identifiers assigned to each cluster on each node onto Node 0 is shown in Equation 7. Combined with Phase 1 of the buckshot algorithm results in a time complexity of \(O\left(\frac{kn}{p}\right)\).

\[
O\left(\frac{1}{p}\sum_{i=1}^{s}C_{\text{read}} + \frac{n-s}{p}C_{\text{read}} + k \cdot \frac{C_{\text{sim}}}{p}ight) = O\left(\frac{kn}{p}\right) \tag{7}
\]

- \(C_{\text{read}}\)—cost of reading two document vectors during centroid calculation.
- \(C_{\text{read}}\)—cost of reading a document from disk.
Csim — cost of calculating the similarity coefficient between a document and a centroid.

\( t \) — number of iterations, usually very small.

Necessity to Maintain Fixed Memory

The serial version of our algorithms optimally store only \( \frac{n^2}{p} \) entries while the parallel version stores the entire matrix to reduce communication costs. Our parallel approach requires the storage of the complete rows for a portion of the similarity matrix. Each of these rows represents the similarity measurements between one document and all other documents. Each specific node must load \( \frac{n^2}{p^2} \) documents into memory. Then, for each document not in the node’s partition, the document is loaded, and the similarity measurement is calculated between that document and all of the documents in memory. Since only \( \frac{n^2}{p^2} \) rows are maintained by each node, the memory complexity for our parallel hierarchical agglomerative clustering algorithm is \( O(\frac{n^2}{p}) \). A key requirement is that each node in the system must have sufficient memory available to hold the term vectors for the \( \frac{n^2}{p^2} \) rows it manages. This is the dominating memory cost in our parallel algorithm, as the storage requirements for the similarity matrix are insignificant by comparison (The similarity scores are stored as single-precision floating point numbers.) This also allows our parallel hierarchical agglomerative clustering algorithm to cluster increasingly large document collections as the number of nodes increases.

Methodology

To demonstrate that our algorithms are scalable in terms of number of processing nodes and size of document collection, we performed a series of experiments varying each of these parameters while examining variation from the expected scaling behavior. In addition, we show that the parallel buckshot algorithm also is scalable as the number of clusters increases.

Setup

Our experiments were run using a Linux Beowulf cluster consisting of 12 total computers, each with two AMD Athlon 2-GHz CPUs and 2 GB of main memory. Communication is facilitated through a copper gigabit ethernet switch that interconnects the nodes. We implemented our algorithm in Java 1.4, using the MPI for Java library (Baker, Carpenter, Fox, Ko, & Lim, 1999) as a wrapper for the MPICH (Gropp et al., 1996) implementation of MPI. All communication operations in our implementation make use of underlying recursive doubling collective algorithms in the MPICH library (Thakur & Gropp, 2003). Experiments were run on dedicated nodes. All experiments used only one of the two processors in the computers, as the implementation was single-threaded; this prevented inaccuracies from contention for the machines’ single network and disk I/O channels.

We used Dataset 1, a 73-MB collection consisting of 20,000 documents to test the scalability of our hierarchical agglomerative clustering algorithm. Dataset 1 is a subset of the 2-GB SGML collection from TREC Disks 4 and 5 (NIST Text Retrieval Conference). The entire TREC Disks 4 and 5 were used to test the scalability of our parallel buckshot algorithm. We used our information retrieval engine, AIRE (Infantes-Morris, Bernhard, Fox, Faulkner, & Stripling, 2003), to facilitate document parsing and similarity calculations. Documents were parsed into term vectors prior to clustering.
using AIRE’s index process, which builds these term vectors for use in relevance feedback. Stopwords from the Cornell’s SMART 342-word stopword list were removed, and terms were stemmed as we have done in past TREC experiments (Chowdhury et al., 2000). No phrase indexing or processing was performed. Term vector files were replicated onto single local UDMA33 SCSI hard drives on each node. Lexicon data needed for similarity calculations were loaded in their entirety into memory on each node from an NFS-shared file system. No other significant disk I/O was necessary.

Performance Metrics

We experimented with various configurations and measured run time using the JVM libraries. Our timings begin from the completion of process launching and communications initialization (MPI.Init) to the completion of the gathering of the cluster definitions onto the root node. Disk I/O time consists of the sum of the time for bulk I/O sections such as the loading of the term lexicon with the time for repeated I/O operations such as reading document vectors. The hierarchical agglomerative clustering algorithm contains one initial pass through the collection while the buckshot algorithm contains a single pass through the collection for each iteration. Since the majority of the processing time for the hierarchical algorithm involves searching and modifying the similarity matrix, I/O cost is very low and is not included in the analysis of our hierarchical agglomerative clustering results. The buckshot algorithm, however, incurs a much larger I/O cost as each document needs to be loaded each iteration. Thus, I/O time is included in the analysis of our buckshot results.

Results

Experiments vary the number of computation nodes (p) and the collection size to demonstrate scalability. Since the computational complexity of the buckshot algorithm is \(O(knk)\), our experiments using the buckshot algorithm also vary \(k\). For the buckshot algorithm, increasing \(k\) is computationally similar to an increase in the dataset size since all portions of the algorithm scale with the product \(nk\) and never its components individually. We show the cost of our parallelization when increasing \(p\) is offset by the scaling of \(k\). Sequential implementations of the algorithm have difficulty scaling to large datasets. Initial centers and final clusters for each \(k\) were verified to be identical across all experiments with variant \(p\) to ensure that our algorithm does indeed produce the same clusters with different numbers of nodes.

We first examine the scalability of our parallel hierarchical agglomerative clustering algorithm. We then show that using our parallel hierarchical agglomerative algorithm as the clustering subroutine for the parallel buckshot algorithm scales linearly and allows larger document collections to be clustered at a faster rate than using the hierarchical agglomerative clustering algorithm alone.

Hierarchical Agglomerative Clustering Results

Our results focus on the examination of two key issues: scalability in number of nodes and collection size. In Table 1, we provide the raw timings for the hierarchical agglomerative clustering algorithm for clusterings of Dataset 1 to create 128 clusters with varying number of nodes. Table 2 shows the speed-up calculated using the sequential (i.e., one-node) and two-node hierarchical agglomerative clustering runs corresponding to the timings in Table 1.

In Figure 8, we plot the speed-up corresponding to the values in Table 2. The sequential run uses a version of the program optimized for sequential execution. Most significant is that the serial version performs the necessary comparisons between distinct pairs of documents only to build the requisite similarity matrix. That is, since the matrix is symmetric, only half of the similarity matrix is needed. In other words, the sequential time complexity of \(O(\frac{1}{2}kn^2)\) time is equal to the time complexity using two nodes of \(O(\frac{1}{2}kn)\) time. Furthermore, the sequential run does not incur communication costs, making it faster than the two-node parallel run. Clearly, as the number of nodes increases, the

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total w/o I/O</td>
<td>540</td>
<td>715</td>
<td>360</td>
<td>242</td>
<td>184</td>
<td>148</td>
<td>125</td>
</tr>
</tbody>
</table>

TABLE 1. Execution times (min) to cluster Dataset 1.

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>0.76</td>
<td>1.50</td>
<td>2.23</td>
<td>2.93</td>
<td>3.65</td>
<td>4.32</td>
</tr>
<tr>
<td>2-node</td>
<td>1.00</td>
<td>1.99</td>
<td>2.95</td>
<td>3.89</td>
<td>4.83</td>
<td>5.72</td>
</tr>
</tbody>
</table>

TABLE 2. Speed-up calculated using the sequential and 2-node hierarchical runs on Dataset 1.

FIG. 8. Speed-up on Dataset 1.
time to cluster decreases. Our goal is to demonstrate scalability as the nodes increase. Because the scalability is offset by a factor of 2, we expect our algorithm to exhibit a speed-up of \( \frac{p}{2} \) rather than \( p \). In Figure 8, we show the speed-up calculated using the sequential and two-node runs. In addition, the theoretical speed-up of \( \frac{p}{2} \) also is shown. As can be seen from this experiment, when the number of nodes is increased, the execution time decreases in a nearly linear fashion, as predicted by the algorithm’s \( O\left(\frac{n}{p}\right) \) time. For 128 clusters, scaling from the two-node parallel run to the 12-node one provides a speed-up of 5.72 of the theoretically optimal 6.0.

In Table 3 and Figure 9, we examine scaling the collection size. Three separate collections are examined, one consisting of 5,000 documents (Dataset 2), one of 10,000 documents (Dataset 3), and one of 15,000 documents (Dataset 4). Dataset 2 to 4 are all subsets of Dataset 1. As with scaling the number of clusters, we see that the algorithm scales close to run time. In one example from our experiments, when we double the collection size on 12 nodes from 10,000 to 20,000 documents, our system takes 4.31 times as long to execute in contrast to the four times predicted by the theoretical analysis.

Cluster Quality

To evaluate the quality of our clusters, we compared against a bisection variant of the well-known and commonly used \( k \)-means algorithm generated using the vcluster program included in the CLUTO package (Karypis, 2002). We evaluate the tightness of a cluster by measuring the average internal similarity between each document in a cluster with that cluster’s centroid, similar to the evaluation performed by Zhong (2005).

Table 4 shows the average internal normalized cosine similarity measurements between the documents in a cluster and the centroid of the cluster. The fourth column shows the results of a paired \( t \) test using the similarity measurements from each document to the cluster centroid. The rows with a check mark show that there is a statistically significant difference with a 95% confidence. Our results show that the hierarchical algorithm produces clusters with better quality when \( k \) is greater than 64. Furthermore, as \( k \) increases, the quality of the hierarchical clusters improves at a faster rate than that for the \( k \)-means clusters.

Parallel Buckshot Clustering Algorithm Results

We show scalability of our parallel buckshot clustering algorithm by performing experiments on the entire 2-GB TREC Disks 4 and 5 collection. Our experiments examine the scalability of the buckshot algorithm when our parallel hierarchical agglomerative clustering algorithm is used as
the initial clustering subroutine. The principal comparison is between a fully optimized implementation of the sequential buckshot algorithm from prior work and our parallel buckshot algorithm. Our results focus on the examination of three key issues: scalability in number of nodes, collection size, and number of clusters.

Table 5 provides the raw timings with and without input/output cost (e.g., reading documents from disk, etc.) for clusterings of the 2 GB of SGML data on TREC Disks 4 and 5 with varying numbers of nodes and clusters. Also shown are the timings for Phase 1 of the buckshot clustering algorithm, the hierarchical agglomerative clustering subroutine. Table 6 shows the speed-up of the hierarchical agglomerative clustering subroutine and the entire buckshot algorithm corresponding to the timings in Table 5. Our results show that although our parallel hierarchical agglomerative clustering subroutine runs in half of the expected time $O(\frac{k}{p})$, our parallel buckshot algorithm results in near optimal speed-up. This is clearly due to the dominance of the latter stage in terms of processing time. In Figure 10, we plot the speed-up corresponding to the timings in Table 5. As can be seen from this graph, when the number of nodes is increased, the execution time decreases in a nearly linear fashion, as predicted by the algorithm’s time.

In Figure 11, we examine scaling the number of clusters based on the same runs on TREC Disks 4 and 5. These experiments show that scaling the number of clusters by a factor of 2 is close to the doubled execution time expected by $O(\alpha k)$ growth. For example, scaling from 256 to 512 clusters on 12 nodes including IO time takes 2.04 times as long to execute in contrast to the two-times increase projected by the theoretical analysis.

In Table 7 and Figure 12, we give timings for and examine scaling the collection size by beginning with the 484-MB subset of 131,890 Los Angeles Times documents and duplicating it to achieve collections of 968 MB and 1,452 MB containing 263,780 and 395,670 documents, respectively. While this does decrease the diversity of the term distributions used in the resulting collections, it is not likely to drastically affect running time, which is primarily defined by the number of document-to-document comparisons being performed. Rather, duplicating a reasonable-sized natural

---

**Table 5. Execution times on TREC Disks 4 and 5 (min).**

<table>
<thead>
<tr>
<th>Number of clusters</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>w/o I/O</td>
<td>I/O</td>
<td>w/o I/O</td>
<td>I/O</td>
<td>w/o I/O</td>
</tr>
<tr>
<td>Nodes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>49</td>
<td>251</td>
<td>543</td>
<td>93</td>
<td>493</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>167</td>
<td>316</td>
<td>134</td>
<td>339</td>
</tr>
<tr>
<td>4</td>
<td>33</td>
<td>86</td>
<td>163</td>
<td>67</td>
<td>337</td>
</tr>
<tr>
<td>6</td>
<td>21</td>
<td>56</td>
<td>110</td>
<td>45</td>
<td>113</td>
</tr>
<tr>
<td>8</td>
<td>18</td>
<td>45</td>
<td>87</td>
<td>34</td>
<td>86</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>38</td>
<td>72</td>
<td>30</td>
<td>71</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>32</td>
<td>62</td>
<td>24</td>
<td>60</td>
</tr>
</tbody>
</table>

**Table 6. Phase 1 and total buckshot speed-up on TREC Disks 4 and 5.**

<table>
<thead>
<tr>
<th>Number of clusters</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.77</td>
<td>1.72</td>
<td>0.69</td>
<td>1.68</td>
<td>0.71</td>
</tr>
<tr>
<td>4</td>
<td>1.48</td>
<td>3.33</td>
<td>1.39</td>
<td>3.39</td>
<td>1.38</td>
</tr>
<tr>
<td>6</td>
<td>2.33</td>
<td>4.94</td>
<td>2.07</td>
<td>4.96</td>
<td>2.09</td>
</tr>
<tr>
<td>8</td>
<td>2.72</td>
<td>6.24</td>
<td>2.74</td>
<td>6.58</td>
<td>2.67</td>
</tr>
<tr>
<td>10</td>
<td>3.27</td>
<td>7.54</td>
<td>3.10</td>
<td>7.94</td>
<td>3.17</td>
</tr>
<tr>
<td>12</td>
<td>3.77</td>
<td>8.76</td>
<td>3.88</td>
<td>9.32</td>
<td>3.91</td>
</tr>
</tbody>
</table>
A collection such as we have done provides a fair approximation to a homogeneous collection of like size, as the documents themselves are unaltered and comparisons between them are comparable to those we might expect to find. As with scaling the number of clusters, we see that the algorithm scales to $O(n^2)$ run time. In one example from our experiments, when we double the collection size on 12 nodes including I/O, our system takes 2.03 times as long to execute in contrast to the two times predicted by the theoretical analysis.

### Cluster Quality

To evaluate the quality of our clusters, we compared against a bisection variant of the $k$-means algorithm generated using the vcluster program included in the CLUTO package (Karypis, 2002). We evaluate the tightness of a cluster by measuring the average internal similarity between each document in a cluster with that cluster’s centroid. This comparison was made to validate our approach since the $k$-means algorithm is commonly thought of as an efficient scalable algorithm of choice.

Table 8 gives the average internal normalized cosine measurements. The fourth column shows the results of a paired $t$ test using the similarity measurements from each document to the cluster centroid. The rows with a check mark show that there is a statistically significant difference with a 99% confidence. Our results show that the clusters generated using the buckshot algorithm have significantly better quality than those generated using the $k$-means algorithm. Thus, our approach provides a credible alternative to parallel $k$-means.

### Summary and Future Work

We designed, implemented, and thoroughly evaluated a parallel version of the hierarchical agglomerative clustering algorithm, which is optimized for parallel computation with reduced interprocess communication on semi-large datasets. In addition, we showed how our parallel hierarchical agglomerative clustering algorithm can be used as the clustering subroutine of our parallel buckshot clustering algorithm to facilitate clustering of large document collections.

We focused on showing the scalability of our parallel hierarchical agglomerative algorithm in terms of the number of nodes and collection size. Our results showed that our algorithm scaled linearly as the number of nodes increased. As the collection size increased, our algorithm performs at near theoretical expectations. In addition, the $O(n^2)$ memory complexity allows larger collections to be clustered as the number of nodes increases. Cluster quality was evaluated and determined to be tighter than clusters generated by a bisection variant of the $k$-means algorithm.
In addition to scalability in terms of number of nodes and collection size, we showed the scalability of our parallel buckshot algorithm as the number of clusters increased. In all three scalability requirements, we saw performance near theoretical expectations, indicating that our parallel algorithm could scale to much larger numbers of nodes and collection sizes. When scaling collection size, we saw a scaling of execution time near to $O(nk)$.) Our results showed that our algorithm scaled linearly as the number of nodes increased. Informally, we have used this system to cluster a filtered version of the 18-GB TREC collection of government Web pages into 256 clusters in approximately 1 day on 32 processors.

There are two high-level categories for future work: clustering efficiency and clustering effectiveness. We plan to address efficiency by experimenting with an even larger corpus on more nodes. We will examine a memory-bounded version of our algorithms, which allows for a flexible balance of memory footprint and speed of execution. Also planned are experiments with load-balancing and communication-balancing techniques geared toward a heterogeneous execution environment, perhaps residing on a grid of computers where communication costs can vary greatly. Effectiveness will be tested by attempting to integrate the clusters into the retrieval process to improve average precision.

References


