Exploiting Parallelism to Support Scalable Hierarchical Clustering*

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

Abstract

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 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^3/p) 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 showing 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.

1 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 data sets 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 tradeoff between accuracy and speed. Hierarchical agglomerative clustering algorithms calculate a full document-to-document similarity matrx. 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 [1, 2]. Our algorithm has an expected time of O($\frac{n^2}{p}$) 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

*We would like to thank the National Science Foundation and the Army Research Office for their support of this effort under contract numbers NSF EIA-0119469, NSF EIA-0130673, and ARO DAAD19-01-1-0432.

in a reasonable amount of time. Since we use an optimized serial hierarchical agglomerative clustering algorithm, our actual speedup is O($\frac{n^2}{2p}$), half of the expected speedup. That is, only half of the symmetric matrix is used by an optimized serial algorithm. In our parallel approach, to reduce inter-node communication, we process a complete matrix. hence, our two node instances require roughly the same processing as the optimized serial version. The speedup, however, is consistent and does not decay. Namely, from two nodes onwards, 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 highly accurate clustering subroutine in the buckshot algorithm to facilitate clustering of larger document collections. The buckshot algorithm results in near optimal speedup.

2 Background and Prior Work

Although many clustering techniques are currently available [3], there are two main categories of approaches: *partitioning* and *hierarchical* clustering. *Partitioning* assigns every document to a single cluster iteratively [4, 5] in an attempt to determine k partitions that optimize a certain criterion function [6]. 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 [7]. *Agglomerative* algorithms typically use a *stored matrix* or *stored data* approach [8]. 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 are also updated. The total time complexity for the similarity matrix method is O(n^3) 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 [9]. Both the similarity matrix and priority queue methods require a memory complexity of O(n^2). It is important to note, however, that since the priority queue method must also store document identifiers, it requires over double the memory of the similarity matrix method. *Stored data* approaches require the recalculation 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 α, the time complexity is O(αn^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 [8].

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 [10, 11, 12]. 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 [13, 14]. 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 [15]. This reduces the computational complexity to $O(kn)$ time [16]. 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 [17].

There has been work done to develop scalable algorithms for clustering. A scalable clustering approach has three main aspects [19]. 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, et al minimizes the number of scans by using the k-means algorithm with a limited memory buffer to store summaries of the documents already scanned [20]. Ordonez, et al uses a relational database to store the data set, generally reducing the number of disk scans to three [21]. 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 [6, 22, 23, 24]. 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 [25, 26, 27]. He, et al represent the document as a low dimensional vector from a compact representation subspace [28]. A δ tree index where the number of dimensions increases towards the leaf level has been used to speed up processing of high-dimensional k-nearest neighbor queries [29]. Orlandic, et al use a data reduction method that represents the data space as a set of dense cells [30].

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, et al used a parallel k-means algorithm to create up to 16 clusters from generated test collections of documents having 8-128 terms in length, the largest of which was 2GB [31]. In addition, Dhillon, et al multi-threaded the spherical k-means partitioning algorithm and achieved near linear speedup and scaleup when running on 113,716 NSF award abstracts averaging 72 terms in length after term-filtering [32]. There exists some work on parallel hierarchical agglomerative clustering, but most of these algorithms have large computational overhead or have not been evaluated for document clustering [6, 22, 33, 34]. 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 examines a parallel hierarchical agglomerative clustering algorithm based on dividing the data into partially overlapping partitions [8]. Experiments showed 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 [33, 35, 36]. Ruocco and Frieder's single-pass partitioning algorithm showed near linear speedup on subsets of the *Tipster* document collection, the largest of which contained 10,000 documents [18]. 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 performed experiments on changing the spatial density of a document collection using clustering with the vector space model [37, 38]. Xu and Croft described a method for improving distributed retrieval effectiveness using language models of clustered collections [39]. More recently, models were presented by which retrieval effectiveness might be improved through modified term weighting in clustered document collections [40]. Query-time efficiency can also 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.

3 Sequential Methods

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

3.1 Hierarchical Agglomerative Clustering

For hierarchical agglomerative clustering, each document is initially a separate cluster. Then the clusters are merged in stages until the desired number of clusters are found. We use the sequential hierarchical agglomerative algorithm

[41] 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 are also stored.

Algorithm	Time Complexity
Preconditions: n = document size, k = desired number of clusters	
Phase 1: Build Similarity Matrix	O(n^2)
1. Assign each document d_i to cluster c_i	O(n)
2. For every cluster pair (c_i, c_j) where $i \neq j$ calculate the similarity between c_i and c_j, place in Similarity Matrix D	O(n^2)
3. Find the nearest neighbor and corresponding similarity for each cluster c_i, place in nn_{array} and max_{array}	O(n^2)
Phase 2: Create the Clusters	O(αn^2)
4. Repeat $n - k$ times	
4.1. Search nn_{array} and max_{array} for the clusters i and j with the maximum similarity	O(n)
4.2 Replace clusters i and j by an agglomerated cluster h	O(1)
4.3 Update D to reflect revised similarity between h and all other clusters	O(n)
4.4 Update α elements in nn_{array} and max_{array}	O(αn)
4. Output k clusters	

Figure 1: Hierarchical Agglomerative Clustering Algorithm

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 [42]. Although any similarity measure can be used, in our experimentation, we use a cosine similarity measure [37, 38] that includes document and query length normalization factors estimated from their number of unique terms [43] coupled with a modern term-weighting scheme [44]. 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, nn_{array}, and the corresponding maximum similarity values, max_{array}. The nn_{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 nn_{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 nn_{array}. Similarly, the first position of max_{array} contains the similarity score between clusters 1 and 6, in this case 10.

In this simple example, the columns and rows are labelled 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 zero and one. For simplicity, these scores are represented here as integers. A memory-efficient sequential implementation of the hierarchical agglomerative clustering algorithm requires only approximately $\frac{n^2-n}{2}$ entries (rounding to whole numbers is left out for simplicity throughout) in the similarity matrix, as the matrix is symmetrical over the diagonal.

(a)

	1	**2**	**3**	**4**	**5**	**6**
1	-	9	8	7	8	10
2	9	-	12	9	2	7
3	8	12	-	6	4	11
4	7	9	6	-	10	2
5	8	2	4	10	-	9
6	10	7	11	2	9	-

(b)

nn_{array}
6
3
2
5
4
3

(c)

max_{array}
10
12
12
10
10
11

Figure 2: A sample (a) document-to-document similarity matrix, (b) nearest neighbor array, and (c) maximum similarity array

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 method [45] 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 3.1.1. 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 nn_{array} and max_{array}. Assuming α updates are performed, this step runs in O(αn) 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.

$$sim(h,c) = \frac{(\mid i \mid sim(i,c) + \mid j \mid sim(j,c))}{\mid i \mid\mid j \mid} \tag{3.1.1}$$

The computational complexity of the sequential hierarchical agglomerative clustering algorithm is both O(n^3) and Ω(n^2) [9]. In a worst-case scenario, when $\alpha = n$, the algorithm runs in O(n^3) time, however, Anderberg theorizes that α averages a constant number of updates per iteration [41]. In our experiments, we found that α 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.

3.2 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 to a rectangular runtime of kn where k is much smaller than n [16]. The initial centers generated from the hierarchical agglomerative clustering subroutine 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 [15].

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 is then assigned to the most similar cluster. This is repeated until every document

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

Figure 3: Buckshot Clustering Algorithm

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 only iterated a small fixed number of times [15]. Our algorithm performs two iterations of the second phase.

4 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 [46]. The time for broadcast and gather are given in Equations 4.0.1 and 4.0.2 [47].

$$\texttt{broadcast} : O((C_{latency} + N_{bytes}C_{transfer})lg\ p) \quad (4.0.1)$$

$$\texttt{gather:} O(C_{latency}lg\ p + N_{bytes}C_{transfer}) \quad (4.0.2)$$

- $C_{latency}$ - startup cost of communicating
- N_{bytes} - number of bytes to be communicated
- $C_{transfer}$ - time required to transmit a single byte

4.1 Parallel Hierarchical Agglomerative Clustering Algorithm

The first phase of the Hierarchical agglomerative clustering algorithm is fairly straightforward to parallelize, as the data can be easily partitioned 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

Algorithm	Time Complexity
n = collection size, k = # of clusters, p = # of nodes	
Phase 1: Build Similarity Matrix	O($\frac{n^2}{p}$)
1. Broadcast document IDs to all processors using the **MPI_Bcast** collective operation	O($N_{bytes}\ log\ p$)
2. **In Parallel**: Partition the document collection C into p collections of $\frac{n}{p}$ documents each.	O(1)
3. **In Parallel**: Load term vectors for all sample documents in partition p_i from disk into memory on each processor	O($\frac{n}{p}$)
4. **In Parallel**: For each document $d_i \in \{C - p_i\}$	O($\frac{n^2}{p}$)
4.1. **In Parallel**: Load term vector for d_i into memory	O(1)
4.2. **In Parallel**: Calculate $sim(d_i, d_j)$ for each $d_j \in p_i$, place in submatrix D	O($\frac{n}{p}$)
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 nn_{array} and max_{array}	O($\frac{n^2}{p}$)
6. Gather the size of each processor's matrix portion on all processors using the **MPI_Allgather** collective operation	O($log\ p$)
Phase 2: Create k clusters	O($\frac{\alpha n^2}{p}$)
7. Repeat $n - k$ times	
7.1. **In Parallel**: Each processor searches the respective partition of nn_{array} and max_{array} for clusters i and j with maximum similarity	O($\frac{n}{p}$)
7.2 The **MPI_Allgather** collective operation is used to gather the maximum similarity from each processor	O(p)
7.3 **In Parallel**: Each processor determines clusters i and j with the maximum similarity.	O(p)
7.4 **In Parallel**: Each Processor determines the managing processor, $P_{manager}$, responsible for the new cluster and updates the load count for each processor	O(p)
7.5 $P_{manager}$ searches through all rows to find an empty row and broadcasts the row number to all other processors via the **MPI_Bcast** collective operation	O($\frac{n}{p}$)
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.	O($\frac{n}{p}$)
7.7 Each Processor sends new similarities to $P_{manager}$ via **MPI_Gather** collective operation	O($log\ p$)
7.8 $P_{manager}$ updates the respective partition of D with the new similarities	O($\frac{n}{p}$)
7.9 **In Parallel:** Each processor updates the respective partition of nn_{array} and max_{array}	O($\frac{\alpha n}{p}$)
8. Output k clusters	

Figure 4: Parallel Hierarchical Agglomerative Clustering Algorithm

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.

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

4.1.1 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.

(a)									(b)	(c)
		1	**2**	**3**	**4**	**5**	**6**		nn_{array}	max_{array}
N1	**1**	-	9	8	7	8	10		6	10
	2	9	-	12	9	2	7		3	12
N2	**3**	8	12	-	6	4	11		2	12
	4	7	9	6	-	10	2		5	10
N3	**5**	8	2	4	10	-	9		4	10
	6	10	7	11	2	9	-		3	11

Figure 5: A partitioned (a) similarity matrix, (b) nearest neighbor array, and (c) maximum similarity array

By distributing the similarity matrix and nearest neighbor arrays in this fashion, the data and computational load are 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 in section 3.1, efficient sequential implementations of the hierarchical agglomerative clustering algorithm only require the storage of one half of the symmetrical similarity matrix, consisting of $\frac{n^2-n}{2}$ 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. This is done 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 one, node zero 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 sub-matrix 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 $O(\frac{n^2}{p})$, allowing us to cluster increasingly large document collections as the number of nodes increases.

The complete algorithm including phase one 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 nearest neighbor and corresponding maximum similarity for each cluster is given by Equation 4.1.1.

4.1.2 Phase 2: A Parallel Clustering Subroutine

Each node is only responsible for maintaining 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 one. Once a node identifies the two most-similar clusters, it notifies all other nodes in the system.

$$O((C_{latency} + N_{bytes}C_{transfer})lg\, p + \frac{n}{p} \cdot C_{read} + n \cdot \frac{n}{p} \cdot C_{sim} + n \cdot \frac{n}{p} \cdot C_{compare}) = O(\frac{n^2}{p}) \quad (4.1.1)$$

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

As the result of phase one 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, etc... 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 3.1.1.

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.

(a)

		1	**2**	**2,3**	**3**	**4**	**5**	**6**
N1	**1**	-	-	17	-	7	8	10
	2	-	-	-	-	-	-	
	2,3	17	-	-	-	15	6	18
N2	**3**	-	-	-	-	-	-	-
	4	7	-	15	-	-	10	2
N3	**5**	8	-	6	-	10	-	9
	6	10	-	18	-	2	9	-

(b)

nn_{array}
2,3
-
1
-
2,3
4
2,3

(c)

max_{array}
17
-
17
-
15
10
18

Figure 6: A modified (a) similarity matrix, (b) nearest neighbor array, and (c) maximum similarity array

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 under-used 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 sub-matrix is selected to hold the row for the new cluster. Consequently, all the similarity values between each 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 sub-matrix 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 arrays are updated. In this example the nearest neighbor is updated for the new cluster and clusters 1, 4, and 6. Each pass of this algorithm results in the merging of two existing clusters into one, and thus requires $n - k$ steps to form k clusters. The total time taken in each step to find each node's maximum similarity, gather those similarities onto every node, scan those similarities for the global maximum, find an open matrix row,

broadcast that row's identifier, merge document identifiers for the new cluster, calculate the group averages, gather them onto the managing node, and update the nearest neighbor array is given in Equation 4.1.2.

$$
\begin{aligned}
O((n-k)(\frac{n}{p} \cdot C_{compare} + (C_{latency} lg\, p + p \cdot C_{transfer}) + p \cdot C_{compare} + \frac{n}{p} \cdot C_{compare} + (C_{latency} \\
+ C_{transfer}) lg\, p + C_{union} + \frac{n}{p} \cdot C_{groupavg} + (C_{latency} lg\, p + C_{transfer}) + \alpha \cdot \frac{n}{p} \cdot C_{compare})) \\
= O(\frac{\alpha n^2}{p})
\end{aligned} \tag{4.1.2}
$$

- $C_{compare}$ - cost of comparing two numbers
- C_{union} - cost of putting the merged document identifiers into the set of identifiers in the merging cluster
- $C_{groupavg}$ - cost of calculating a group average similarity

Combining both phases of the parallel algorithm, the total time taken is shown in Equation 4.1.3. Note that in a worse case scenario $\alpha = n$ increasing the complexity to $O(\frac{n^3}{p})$, however, since we assume that each iteration changes a constant number of items, the expected complexity becomes O($\frac{n^2}{p}$).

$$
O(\frac{n^2}{p}) + O(\frac{\alpha n^2}{p}) = O(\frac{\alpha n^2}{p}) = O(\frac{n^2}{p}) \tag{4.1.3}
$$

Algorithm	Time Complexity
Precondition: n = collection size, k = # of clusters, p=# of nodes, s=# of sample documents	
Phase 1: Run Cluster Subroutine	O($\frac{\alpha s^2}{p}$)
1. Run parallel hierarchical agglomerative clustering algorithm (See Fig 4) with s documents.	O($\frac{\alpha s^2}{p}$)
Phase 2: Group remaining documents	O($\frac{kn}{p}$)
2. **In Parallel**: Each processor calculates the centroids of all k initial clusters.	O(s)
3. **In Parallel**: Each processor is assigned $\frac{n-s}{p}$ of the remaining documents to be clustered	O($\frac{n}{p}$)
4. **In Parallel**: Each processor does the following for each document d in the respective set of remaining documents:	O($\frac{kn}{p}$)
4.1. **In Parallel**: Load term vector for document d into memory	O(1)
4.2. **In Parallel**: Find similarity between d and each cluster centroid c_i	O(k)
4.3. **In Parallel**: Assign d to cluster i where $sim(d, c_i) > sim(d, c_j)$ for all $i \neq j$	O(1)
5. Gather the cluster assignments for all remaining documents onto the root processor, P_0 via **MPI_Gather** collective operation.	O($log\ p$)
6. Repeat Phase 2	O($\frac{kn}{p}$)
7. Output k clusters	

Figure 7: Parallel Buckshot Clustering Algorithm

4.2 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, $\frac{s}{k}$ term vectors in size, making transmitting them a significantly larger cost than calculating all of them. It should be noted that the effectiveness of load-balancing in phase one of our parallel hierarchical agglomerative clustering algorithm and phase two of our parallel buckshot algorithm depends to some degree on each node being assigned documents of roughly similar length. The documents in the 2GB 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 standard deviation 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 one standard deviation 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 $\frac{n-s}{p}$ 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 three; the nodes only synchronize at completion to combine their clusters onto node zero. 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 zero is shown in Equation 4.2.1. Combined with phase 1 of the buckshot algorithm results in a time complexity of O($\frac{\alpha kn}{p}$).

$$O(t(s \cdot C_{vectadd} + \frac{n-s}{p}(C_{read} + k \cdot C_{sim}) + (C_{latency} lg\, p + C_{transfer}))) = O(\frac{ts^2}{p}) = O(\frac{kn}{p}) \qquad (4.2.1)$$

- $C_{vectadd}$ - cost of summing two document vectors during centroid calculation
- C_{read} - cost of reading a document from disk
- C_{sim} - cost of calculating the similarity coefficient between a document and a centroid
- t - number of iterations, usually very small

4.3 Necessity to Maintain Fixed Memory

The serial version of our algorithms optimally store only $\frac{n^2}{2}$ 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 $\frac{n}{p}$ 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 $\frac{n}{p}$ 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 $\frac{n}{p}$ 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.

5 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 the parallel buckshot algorithm is also scalable as the number of clusters increases.

5.1 Setup

Our experiments were run using a Linux Beowulf cluster consisting of 12 total computers, each with two AMD Athlon 2GHz CPUs and 2GB 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 [48] as a wrapper for the MPICH [46] implementation of MPI. All communication operations in our implementation make use of underlying recursive doubling collective algorithms in the MPICH library [47]. 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 *Data Set 1*, a 73MB collection consisting of 20,000 documents to test the scalability of our hierarchical agglomerative clustering algorithm. *Data Set 1* is a subset of the 2GB SGMLcollection from TREC disks four and five [49] The entire TREC disks four and five were used to test the scalability of our parallel buckshot algorithm. We used our information retrieval engine, AIRE [50], 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 [44]. 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 was loaded in its entirety into memory on each node from an NFS shared filesystem. No other significant disk I/O was necessary.

5.2 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.

6 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(αkn), our experiments using the buckshot algorithm also vary k. For the buckshot algorithm, increasing k is computationally similar to an increase in the data set 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 data sets. 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. Then we 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.

	Number of nodes						
	1	*2*	*4*	*6*	*8*	*10*	*12*
total w/o IO	540	715	360	242	184	148	125

Table 1: Execution times (minutes) to cluster *Data Set 1*

	Number of nodes					
	2	*4*	*6*	*8*	*10*	*12*
sequential	0.76	1.50	2.23	2.93	3.65	4.32
two node	1.00	1.99	2.95	3.89	4.83	5.72

Table 2: Speedup calculated using the sequential and two node hierarchical runs on *Data Set 1*

6.1 Hierarchical Agglomerative Clustering Results

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

In Figure 8, we plot the speedup corresponding to the values in Figures 2. The sequential run uses a version of the program optimized for sequential execution. Most significant is that the serial version only performs the necessary comparisons between distinct pairs of documents 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($\frac{1}{2}\alpha n^2$) time is equal to the time complexity using two nodes of O($\frac{\alpha n^2}{2}$) 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 time to cluster decreases. Our goal is to demonstrate scalability as the nodes increase. Because the scalability is offset by a factor of two, we expect our algorithm to exhibit a speedup of $\frac{p}{2}$ rather than p. In Figure 8, we show the speedup calculated using the sequential and two node runs. In addition the theoretical speedup of $\frac{p}{2}$ is also 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 $O(\frac{\alpha n^2}{p})$ time. For 128 clusters, scaling from the two node parallel run to the 12 node one provides a speedup of 5.72 out of the theoretically optimal 6.0.

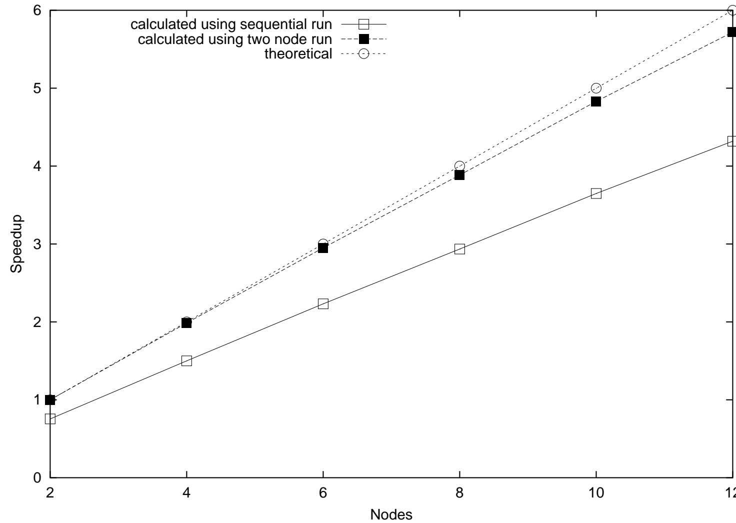

Figure 8: Speedup on *Data Set 1*

nodes	*Subsets of Data Set 1*			
	Data Set 1 20,000	*Data Set 2* 15,000	*Data Set 3* 10,000	*Data Set 4* 5,000
1	540	303	139	34
2	715	378	158	37
4	360	191	80	19
6	242	129	54	13
8	184	100	41	10
10	148	79	34	8
12	128	68	29	7

Table 3: Execution times (minutes) for creating 128 clusters on collections varying in size. Ideal performance is a quadratic increase as the collection size increases and linear decrease as the number of nodes increases

In Figures 3 and 9, we examine scaling the collection size. Three separate collections are examined, one consisting of 5,000 documents (*Data Set 2*), one of 10,000 documents (*Data Set 3*), and one of 15,000 documents (*Data Set 4*). *Data Set 2-4* are all subsets of *Data Set 1*. As with scaling the number of clusters, we see that the algorithm scales close to O($\frac{n^2}{p}$) 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 4 times predicted by the theoretical analysis.

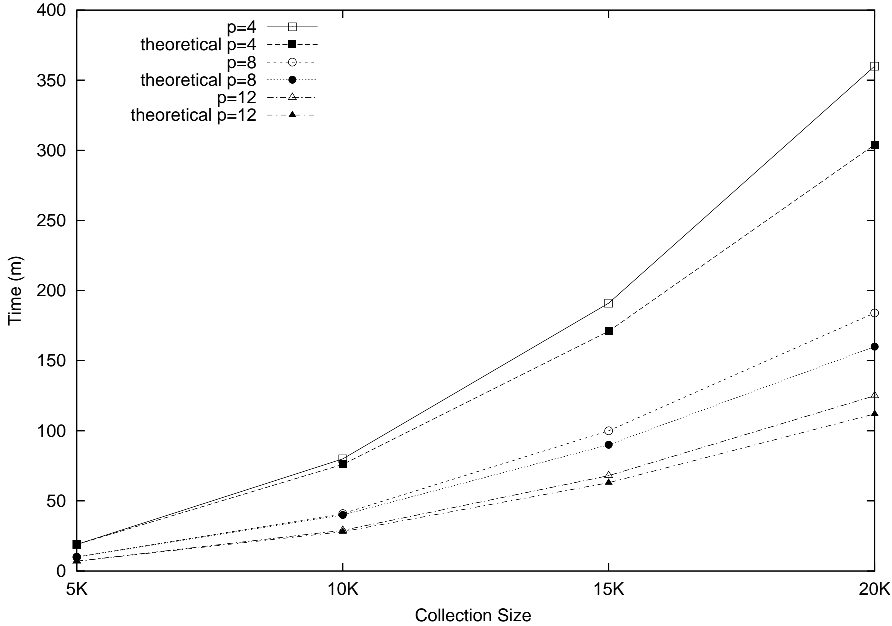

Figure 9: Scaling Collection Size for 128 Clusters

6.2 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 [51]. 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 in [52].

In Figure 4, we show 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 significance test using the similarity measurements from each document to the cluster centroid.The rows with a checkmark 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 the k-means clusters.

clusters	k-means	Hierarchical	95% significance
32	3.87×10^{-3}	3.13×10^{-3}	$\surd$
64	4.29×10^{-3}	4.17×10^{-3}	$\surd$
128	4.77×10^{-3}	5.34×10^{-3}	$\surd$
256	5.44×10^{-3}	6.88×10^{-3}	$\surd$
512	6.16×10^{-3}	8.22×10^{-3}	$\surd$

Table 4: Average Internal Normalized Cosine Measurements where larger measurements are preferred

6.3 Parallel Buckshot Clustering Algorithm Results

We show scalability of our parallel buckshot clustering algorithm by performing experiments on the entire 2GB TREC Disks four and five 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.

In Figure 5, we provide the raw timings with and without input/output cost (reading documents from disk, etc.) for clusterings of the 2GB of SGML data on TREC disks 4 and 5 with varying numbers of nodes and clusters. Also shown are the timings for phase one of the buckshot clustering algorithm, the hierarchical agglomerative clustering subroutine. Figure 6 shows the speedup of the hierarchical agglomerative clustering subroutine and the entire buckshot algorithm corresponding to the timings in Figure 5. Our results show that although our parallel hierarchical agglomerative clustering subroutine runs in half of the expected time $O(\frac{n^2}{2p})$, our parallel buckshot algorithm results in near optimal speedup. This is clearly due to the dominance of the latter stage in terms of processing time. In Figure 10, we plot the speedup corresponding to the timings in Figure 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 $O(\frac{\alpha kn}{p})$ time. Since the second phase of the parallel buckshot algorithm is evenly distributed among the nodes, the two node buckshot run is expected to exhibit near optimal speedup. For 512 clusters, scaling from the optimized sequential (one node) run to the 12 node run including IO time provides a speedup of 10.02 compared to the theoretical speedup of 12. Furthermore, the speedup of phase 1 is 4.49 compared to the theoretical 6. The improved performance as k increases shows that the increased cost of our parallelization when increasing p is offset by the scaling of k.

In Figure 11, we examine scaling the number of clusters based on the same runs on TREC Disks four and five. These experiments show that scaling the number of clusters by a factor of two results in the less than the doubled execution time expected by $O(\alpha kn)$ 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 2 times increase projected by the theoretical analysis.

In Figures 7 and 12, we give timings for and examine scaling the collection size by beginning with the 484MB subset of 131,890 LA Times documents and duplicating it to achieve collections of 968MB and 1452MB contcluaining 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 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 $O(\frac{\alpha kn}{p})$ run time. In one example from our experiments, when we double the collection size on 12 nodes including IO our system takes 2.03 times as long to execute, in contrast to the 2 times predicted by the theoretical analysis.

	Number of clusters														
	32			64			128			256			512		
	w/o I/O		I/O	w/o I/O		I/O	w/o I/O		I/O	w/o I/O		I/O	w/o I/O		I/O
nodes	Phase 1	total	total	Phase 1	total	total	Phase 1	total	total	Phase 1	total	total	Phase 1	total	total
1	49	251	543	93	493	1072	184	996	2152	358	1962	4241	840	4170	8737
2	64	167	316	134	339	639	258	671	1261	504	1327	2497	1028	2759	5067
4	33	86	163	67	167	316	133	337	630	266	689	1274	534	1378	2542
6	21	56	110	45	113	216	88	223	423	174	446	844	360	921	1714
8	18	45	87	34	86	163	69	172	323	134	346	640	262	693	1282
10	15	38	72	30	71	135	58	141	265	112	277	521	220	553	1027
12	13	32	62	24	60	115	47	115	218	89	227	427	187	478	872

Table 5: Execution times on TREC Disks 4 and 5 (minutes)

	Number of clusters									
	32		64		128		256		512	
nodes	Phase 1	Total	Phase 1	Total	Phase 1	Total	Phase 1	Total	Phase 1	Total
2	0.77	1.72	0.69	1.68	0.71	1.71	0.71	1.70	0.82	1.72
4	1.48	3.33	1.39	3.39	1.38	3.42	1.35	3.33	1.57	3.44
6	2.33	4.94	2.07	4.96	2.09	5.09	2.06	5.02	2.33	5.10
8	2.72	6.24	2.74	6.58	2.67	6.66	2.67	6.63	3.21	6.82
10	3.27	7.54	3.10	7.94	3.17	8.12	3.20	8.14	3.82	8.51
12	3.77	8.76	3.88	9.32	3.91	9.87	4.02	9.93	4.49	10.02

Table 6: Phase 1 and Total Buckshot Speedup on TREC Disks 4 and 5

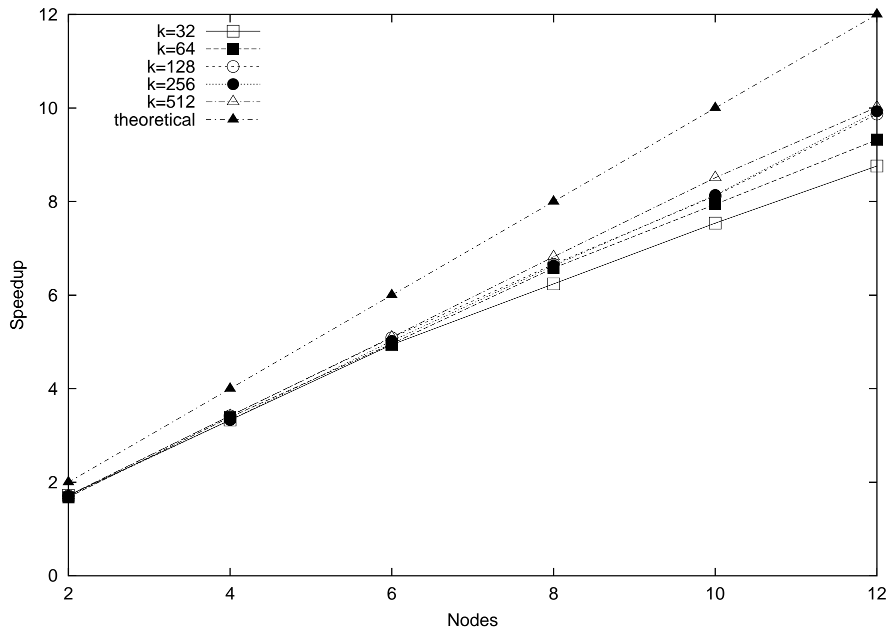

Figure 10: Speedup on TREC Disks 4 and 5

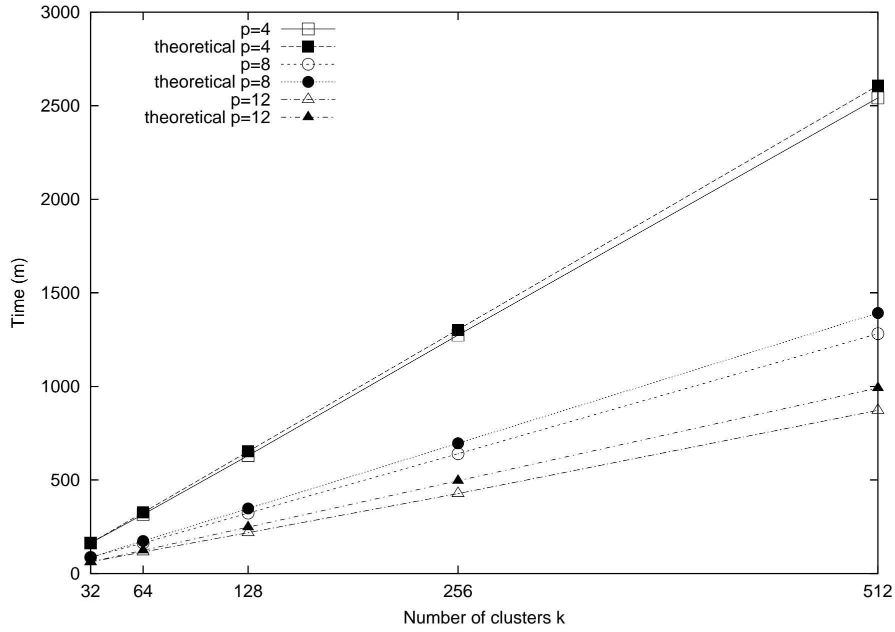

Figure 11: Scaling Number of Clusters on TREC Disks 4 and 5

	Multiple of LA Times Collection					
	1X: 484MB		2X: 968MB		3X: 1452MB	
nodes	I/O	w/o I/O	I/O	w/o I/O	I/O	w/o I/O
1	286	131	568	259	839	379
2	169	87	332	176	498	263
4	84	45	169	89	255	135
6	58	31	115	61	172	91
8	43	23	88	46	131	68
10	35	18	71	37	107	56
12	30	16	61	32	91	46

Table 7: Execution times for clustering multiples of LA Times collection into 64 clusters (minutes)

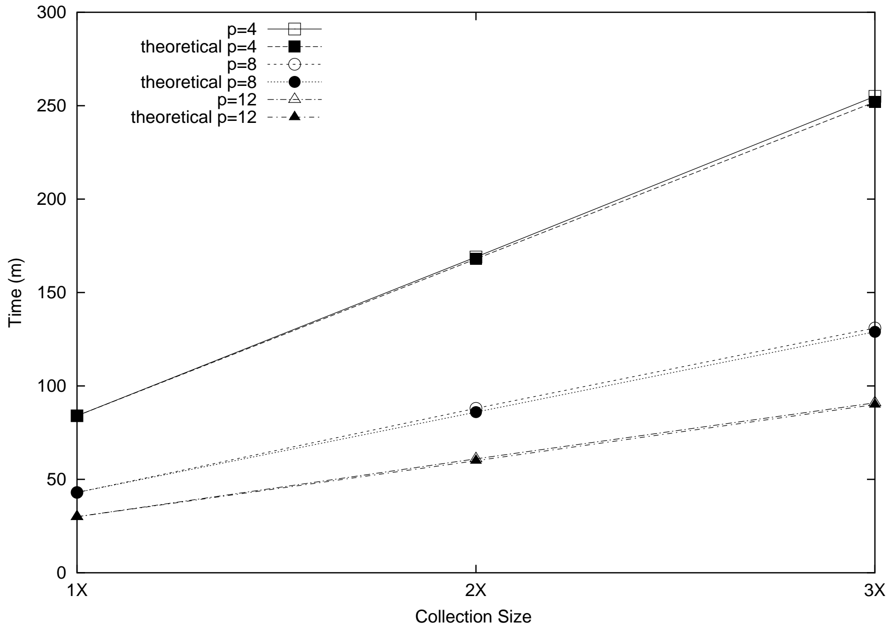

Figure 12: Scaling Collection Size for 64 Clusters

6.4 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 [51]. 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.

Figure 8 gives the average internal normalized cosine measurements. The fourth column shows the results of a paired t-test significance test using the similarity measurements from each document to the cluster centroid. The rows with a checkmark 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.

clusters	k-means	buckshot	99% significance
32	3.86×10^{-3}	4.42×10^{-3}	$\surd$
64	4.30×10^{-3}	5.50×10^{-3}	$\surd$
128	4.75×10^{-3}	6.66×10^{-3}	$\surd$
256	5.38×10^{-3}	7.90×10^{-3}	$\surd$
512	6.04×10^{-3}	8.83×10^{-3}	$\surd$

Table 8: Average Internal Normalized Cosine Measurements where larger measurements are preferred

7 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 data sets. 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($\frac{n^2}{p}$) 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(αkn). 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 18GB TREC collection of government web pages into 256 clusters in approximately one 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 towards 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

[1] W. Gropp, E. Lusk, A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface.* The MIT Press, 1996.

[2] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, *MPI: The Complete Reference.* The MIT Press, 1997.

[3] D. Fasulo, An Analysis of Recent Work on Clustering Algorithms. *Technical Report UW-CSE01 -03-02, University of Washington*, 1999.

[4] J.A. Hartigan, *Clustering Algorithms.* Wiley, 1975.

[5] R.O. Duda, P.E. Hart, *Pattern Classification and Scene Analysis.* Wiley, 1973.

[6] S. Guha, R. Rastogi, K. Shim, CURE: An Efficient Clustering Algorithm for Large Databases. *Proceedings of the 1998 ACM-SIGMOD*, pp. 73-84, 1998.

[7] N. Jardine, C.J. van Rijsbergen, The Use of Hierarchical Clustering in Information Retrieval. *Information Storage and Retrieval*, 1971.

[8] M. Dash, S. Petrutiu, P. Sheuermann. Efficient Parallel Hierarchical Clustering. *In International Europar Conference (EURO-PAR'04)*, 2004.

[9] W. H. E. Day and H. Edelsbrunner, Efficient Algorithms for Agglomerative Hierarchical Clustering Methods, *Journal of Classification*, 1, pp.7-24.

[10] B. Larsen, C. Aone, Fast and effective text mining using linear-time document clustering. *Proceedings of the 5th ACM-SIGKDD*, pp. 16-22, 1999.

[11] P. Willet, Recent trends in hierarchical document clustering: A critical review. *Information Processing and Management*, Vol 24:5 1988, pp. 577-597.

[12] R.C. Dubes, A.K. Jain, *Algorithms for Clustering Data*, Prentice Hall, 1988.

[13] Y. Zhao and G. Karypis, Evaluations of Algorithms for Obtaining Hierarchical Clustering Solutions *Proceedings of the 2002 ACM International Conference on Information and Knowledge Management (ACM-CIKM), Washington D.C.*, November 2002.

[14] M. Steinbach, G. Karypis, V. Kumar, A Comparison of Document Clustering Techniques. *Proceedings of the KDD-2000 Workshop on Text Mining*, 2000.

[15] D. Cutting, D. Karger, J. Pedersen, J. Tukey, Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections. *Proceedings of the Fifteenth Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, 1992.

[16] O. Zamir, O. Etzioni, Web Document Clustering: A Feasibility Demonstration. *21st Annual ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 46-54, 1998.

[17] C. Schütze, C. Silverstein, Projects for Efficient Document Clustering. *Proceedings of twentieth ACM-SIGIR, pp. 74-81, 1997*.

[18] A. Ruocco, O. Frieder, Clustering and Classification of Large Document Bases in a Parallel Environment. *Journal of the American Society of Information Science*, 48 (10), pp. 932-943, 1997.

[19] J. Ghosh, Scalable Clustering Methods for Data Mining. Chapter 10 in *Handbook of Data Mining*, 2003, pp. 247-277.

[20] P.S. Bradleey, U. Fayyad, and C. Reina, Scaling Clustering Algorithms to Large Databases,

[21] C. Ordonez, E. Omiecinski, Efficient Disk-Based K-Means Clustering for Relational Databases. *IEEE Transactions on Knowledge and Data Engineering*, Vol 16:8 August 2004, pp. 909-921.

[22] T. Zhang, R. Ramakrishnan, M. Livny, BIRCH: an Efficient Data Clustering Method for Very Large Databases. *Proceedings of 1996 ACM-SIGMOD, pp. 103-114, Montreal, Canada*, 1996.

[23] K. Chen and L. Liu, ClusterMap: Labeling Clusters in Large Datasets via Visualization. In *Proceedings of the thirteenth ACM conference on Information and knowledge management*, Washington, D.C., 2004.

[24] S. Nassar, J. Sander, and C. Cheng, Incremental and effective data summarization for dynamic hierarchical clustering. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, Paris, France, 2004.

[25] J. Mao, A. Jain. Artifcial neural networks for feature extraction and multivariate data projection. *IEEE Transactions on Neural Networks*, Vol 6:2, pp. 296-317.

[26] R.O.Duda, P.E. Hart, and D.G. Stork, *Pattern Classification (2nd Ed)*, Wiley, 2001.

[27] A. Globerson and N. Tishby, Sufficient dimensionality reduction. *The Journal of Machine Learning Research*, Vol. 3, pp.1307-1331, March, 2003.

[28] X. He, D. Cai, H. Liu, and W. Ma, Locality preserving indexing for document representation. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, Sheffield, United Kingdom, 2004.

[29] B. Cui, B. C. Ooi, J. W. Su, and K. L. Tan, Contorting high dimensional data for efficient main memory KNN processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, San Diego, CA, 2003.

[30] R. Orlandic, Y. Lai, W. Yee, Clustering High-Dimensional Data Using an Efficient and Effective Data Space Reduction. *ACM Fourteenth Conference on Information and Knowledge Management (CIKM)*, Bremen, Germany, 2005.

[31] I.S Dhillon, D.S. Modha, A Data-Clustering Algorithm On Distributed Memory Multiprocessors. *Large-Scale Parallel Data Mining, Lecture Notes in Artificial Intelligence*, Volume 1759, pages 245-260, 2000.

[32] I.S. Dhillon, J. Fan, Y. Guan, Efficient Clustering of Very Large Document Collections. invited book chapter in *Data Mining for Scientific and Engineering Applications*, 2001.

[33] X. Li. Parallel Algorithms for Hierarchical Clustering and Cluster Validity. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12, pp.1088-1092, 1990.

[34] S. Rajasekaran. Efficient Parallel Hierarchical Clustering Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16:6, 2005.

[35] C. Olson. Parallel Algorithms for Hierarchical Clustering. *Journal of Parallel Computing*, 21, pp.1313-1325, 1995.

[36] C. Wu, S. Horng, H. Tsai. Efficient Parallel Algorithms for Hierarchical Clustering on Arrays with Reconfigurable Optical Buses. *Journal of Parallel and Distributed Computing*, 60, pp.1137-1153, 2000.

[37] G. Salton, A Vector Space Model for Automatic Indexing. *Communications of the ACM*, 18 (11), pp. 613-620, November 1975.

[38] G. Salton, M.J. McGill, *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, 1983.

[39] J. Xu and B. Croft. Cluster-based Language Models for Distributed Retrieval, *22st Annual ACM Conference on Research and Development in Information Retrieval (SIGIR)*, 1999.

[40] Y. Zhao, G. Karypis, Improve Precategorized Collection Retrieval by Using Supervised Term Weighting Schemes. *IEEE Conference on Information Technology Coding and Computing, Information Retrieval Session*, April 2002.

[41] M. Anderberg, *Cluster Analysis for Applications*, Academic Press, Inc., New York, NY, 1973.

[42] D. Grossman and O. Frieder, *Information Retrieval: Algorithms and Heuristics. Second Edition* Springer Publishers, 2004.

[43] D. Lee, H. Chuang, K. Seamons. *Document Ranking and the vector-space model.* IEEE Software, Vol 14:2, pp. 67-75, 1997.

[44] A. Chowdhury, S. Beitzel, E. Jensen, M. Saelee, D. Grossman, O. Frieder, IIT-TREC-9 - Entity Based Feedback with Fusion. *Proceedings of the Ninth Annual Text Retrieval Conference, NIST*, November 2000.

[45] E. Voorhees, Implementing Agglomerative Hierarchic Clustering Algorithms for User in Document Retrieval. *Information Processing & Management*, 22:6, pp.465-476, 1986.

[46] W. Gropp, E. Lusk, N. Doss, A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Journal of Parallel Computing*, 22:6, pp.789-828, September, 1996.

[47] R. Thakur and W. Gropp, Improving the Performance of Collective Operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number LNCS2840 in Lecture Notes in Computer Science, Springer Verlag, pp. 257-267, 2003.

[48] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim. mpiJava: An Object-Oriented Java interface to MPI. *Presented at International Workshop on Java for Parallel and Distributed Computing*, IPPS/SPDP 1999, San Juan, Puerto Rico, April 1999. *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*, Menlo Park, CA, 1998.

[49] NIST Text Retrieval Conference. *English Document Collections* http://trec.nist.gov/data/docs_eng.html

[50] T. Infantes-Morris, P. Bernhard, K. Fox, G. Faulkner, K. Stripling, Industrial Evaluation of a Highly-accurate Academic IR System. *Proceedings of the ACM Conference on Information and Knowledge Management*, November 2003.

[51] G. Karypis, *CLUTO - A Clustering Toolkit*, Dept. of Computer Science, University of Minnesota, May 2002, `http://www-users.cs.umn.edu/~karypis/cluto/`.

[52] S. Zhong, Efficient Online Spherical K-Means Clustering. *IEEE International Joint Conference on Neural Networks*, Vol. 5 August 2005, pp. 3180-3185