Parallelizing the Buckshot Algorithm for Efficient Document Clustering[*]

Eric C. Jensen, Steven M. Beitzel, Angelo J. Pilotto, Nazli Goharian, Ophir Frieder
Information Retrieval Laboratory
Illinois Institute of Technology
Chicago, IL 60616
{jensen,beitzel,pilotto,goharian,frieder}@ir.iit.edu

ABSTRACT

We present a parallel implementation of the Buckshot document clustering algorithm. We demonstrate that this parallel approach is highly efficient both in terms of load balancing and minimization of communication. In a series of experiments using the 2GB of SGML data from TReC disks 4 and 5, our parallel approach was shown to be scalable in terms of processors efficiently used and the number of clusters created.

1. INTRODUCTION

We describe a parallel implementation of a classical clustering approach based on partitioning called Buckshot Clustering [2]. Since the volume of data stored and searched in today's information systems is vast, high performance computing, in this case parallel processing, is needed to sustain acceptable clustering times. Some recent parallel text clustering efforts include the works by Dhillon, et. al [4] and Ruocco & Frieder [6].

Dhillon, et. al parallelized the spherical k-means partitioning algorithm and achieved near linear speedup and scaleup when running on test collections of documents from 8-128 terms in length, the largest of which was 2GB [3]. Ruocco and Frieder [6] developed a near linear speedup parallel single-pass partitioning algorithm but only evaluated their approach on subsets of the *Tipster* document collection, the largest of which contained 10000 documents. We evaluate our parallel document clustering on a standard, modern document collection to support future comparisons.

The original buckshot algorithm presented in [2] only defines two stages to the algorithm: selecting a random sample of documents of size $\sqrt{kn}$ and using a cluster subroutine to find initial centers from this random sample. The cluster subroutine can be any hierarchical agglomerative approach. The buckshot algorithm itself gives no information on what to do after the initial centers are created, although techniques for assigning the remaining documents to appropriate centers are given. Our parallel approach uses an Assign-To-Nearest algorithm similar to the one discussed in [2].

2. PARALLEL APPROACH TO BUCKSHOT

Our parallel approach to the buckshot clustering algorithm is described in 3 phases. Phases 1 and 3 of the buckshot algorithm are fairly straightforward to parallelize, as the data can be easily partitioned among nodes, and there is no need for communication or coordination. The main effort involves parallelizing the creation of the initial 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. Our algorithm is designed to produce the same results as a serial implementation. We describe our parallel approach for each phase of the Buckshot algorithm in the following three sections.

2.1 Build Similarity Matrix for $\sqrt{kn}$ Sample

Each row in the document-to-document similarity matrix represents a document in the random sample and the similarity scores relating it to every other document. By using row-based partitioning, we are able to assign each node approximately $\frac{\sqrt{kn}}{p}$ rows of the matrix to "manage". 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 1, we illustrate our sample similarity matrix after partitioning it among three nodes. As shown, the data and the computational load are evenly partitioned over the available nodes in the system.

		1	2	3	4	5	6
N1	1	-	7	8	9	10	11
	2	7	-	12	13	14	15
N2	3	8	12	-	16	17	18
	4	9	13	16	-	19	20
N3	5	10	14	17	19	-	21
	6	11	15	18	20	21	-

Figure 1: A partitioned similarity matrix

[*]We thank the National Science Foundation and the Army Research Office for their support under contract numbers NSF EIA-0119469, NSF EIA-0130673, and ARO DAAD19-01-1-0432.

Efficient serial implementations of the Buckshot algorithm only require the storage of one half of the symmetrical similarity matrix, requiring $\frac{\sqrt{kn}^2-\sqrt{kn}}{2}$ matrix entries. Our parallel approach requires the storage of the complete matrix, so that each node can find similarities between its managed clusters and the newly formed clusters with a minimum of communication, which is of significant importance for any vast amount of data.

In step one, a single node selects the random sample of $\sqrt{kn}$ documents and sends their identifiers to all participating nodes. Each node then identifies its unique subset of the similarity matrix to manage. Once the sub-matrix is identified, each node calculates similarity scores for each of its managed documents to every other document in the random sample. For this process, we load term vectors representing each document in the sample into memory.

A key requirement of our parallel algorithm is that each node in the system must have sufficient main memory available to hold the term vectors for all $\sqrt{kn}$ documents in the random sample. This condition is necessary to prevent prohibitive I/O and communication costs that would be required if term vectors had to be continually shuffled between disk, main memory, and various nodes. This is the dominating memory cost in our parallel algorithm.

2.2 A Parallel Clustering Subroutine

Each node is only responsible for maintenance of its partition of the similarity matrix. Therefore, the first phase in the cluster subroutine is for each node to scan its portion of the matrix for the clusters with the highest similarity. Single documents are viewed as clusters of size one. Once a node identifies its 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 15, along with the two cluster identifiers that correspond to that similarity. Node 2 broadcasts 20 and its component cluster identifiers, and node 3 broadcasts 21, etc. Once each node has discovered the clusters that have highest similarity over the entire matrix, it updates its 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, this is done by keeping a count of how many clusters are currently being managed by each node, and selecting the node with the smallest load as the "managing node" for the new cluster. 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 its portion of the similarity matrix. There are several methods of updating the similarity scores when a new cluster is formed. We used the group-average method proposed in [2]. When a new cluster α is formed from components α_1 and α_2, its similarity to an arbitrary cluster β can be found by the following:

$$sim(\alpha,\beta) = \frac{(sim(\alpha_1,\beta)+sim(\alpha_2,\beta))}{2}$$

In our example, each node updates the scores between the new cluster, created by merged clusters 5 and 6, and each existing cluster. The matrix is updated as shown in figure 2.

		1	2	3	4	5	6	5,6
N1	1	-	7	8	9	-	-	11
	2	7	-	12	13	-	-	15
N2	3	8	12	-	16	-	-	18
	4	9	13	16	-	-	-	20
N3	5	-	-	-	-	-	-	-
	6	-	-	-	-	-	-	-
	5,6	11	15	18	20	-	-	-

Figure 2: A modified similarity matrix

Node 3, as the most under-utilized node due to the merge of clusters 5 and 6, is selected to manage the new cluster. 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 of the matrix in 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 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 1 sends$\{1, 10.5\}$ and $\{2, 14.5\}$ to node 3 to populate the similarities. Node 2 sends $\{3, 17.5\}$ and $\{4, 19.5\}$ to node 3. Once node 3 collects the scores from other nodes and updates its partition of the matrix, the entire matrix has been updated and ready to repeat this process. Each pass of this algorithm results in the merging of two existing clusters into one, and thus requires $\sqrt{kn} - k$ steps to form k clusters.

2.3 Group Remaining Documents in Parallel

After the completion of the clustering subroutine, k initial clusters are created from the random sample of $\sqrt{kn}$ documents. The third phase of the Buckshot algorithm assigns the remaining documents according to their similarity to the centroids of the initial clusters. This step of the algorithm is trivially parallelized via data partitioning. First, the initial cluster centroids are calculated on every node. This is done in favor of communication due to the relatively large cost of communication versus a simple calculation on each node.

After centroid calculation is complete, each node is assigned approximately $\frac{n-\sqrt{kn}}{p}$ 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. Once this process is completed, the document identifiers for each final cluster are gathered onto the root node for writing out to disk.

3. EXPERIMENTATION

3.1 Setup

Our experiments were conducted on a Linux Beowulf cluster of 16 computers connected via a gigabit Ethernet switch, each with two Pentium III 1000MHz CPUs and 1GB of main

	Number of clusters					
	16		32		64	
nodes	I/O	w/o I/O	I/O	w/o I/O	I/O	w/o I/O
2	405	335	763	624	1556	1284
4	207	171	384	314	780	643
6	141	117	257	210	527	435
8	105	87	194	158	401	331
10	85	70	157	128	319	264
12	72	59	131	107	267	221
14	63	52	113	92	231	191
16	55	45	100	81	206	170

Figure 3: Execution times (minutes)

memory. We implemented our algorithm in Java 1.4, using the MPI for Java library [1] as a wrapper for the MPICH [5] message-passing library. All communication operations in our implementation make use of underlying recursive doubling collective algorithms in the MPI library. Experiments were run on idle nodes, using only one processor on each computer.

Experiments were performed using the 2GB SGML collection from TReC disks four and five [7]. Documents were parsed into term vectors prior to clustering. Term vector files were replicated onto single local UDMA33 EIDE hard drives on each node. Lexicon data needed for similarity calculations were loaded entirely into memory on each node from an NFS shared filesystem. No other significant disk I/O was necessary.

3.2 Performance Metrics

Our timings begin with the completion of process launching and communications initialization (MPI.Init) up 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 experiments are done with a varying number of computation nodes (p) and clusters(k) to show scalability in terms of processing nodes and the number of clusters. Note that 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.

The random set of sample documents was held constant for each number of clusters. 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.

3.3 Results

Figure 3 includes the timings with and without I/O costs to examine the effect of I/O on scalability. Figure 4 shows the ratio of times from the 2-node runs as the number of nodes is increased. When the number of nodes is increased the execution time decreases in a nearly linear fashion, particularly when a large number of clusters is used. Figure 5 shows time ratios from the 16-cluster runs as the number of clusters is increased. Scaling the number of clusters by a factor of 2 shows better than the doubled execution time expected by the algorithm's $O(kn)$ growth.

	Number of nodes							
clusters	2	4	6	8	10	12	14	16
16	1	.51	.35	.26	.21	.18	.16	.13
32	1	.50	.34	.25	.21	.17	.15	.13
64	1	.50	.34	.26	.21	.17	.15	.13

Figure 4: Time ratios by the number of nodes.

	Number of clusters		
nodes	16	32	64
2	1	1.86	3.83
4	1	1.84	3.76
6	1	1.79	3.72
8	1	1.82	3.80
10	1	1.83	3.77
12	1	1.81	3.75
14	1	1.77	3.67
16	1	1.80	3.78

Figure 5: Time ratios by number of clusters.

4. CONCLUSION

We designed, implemented, and optimized a parallel version of the Buckshot clustering algorithm. Experimental results on a standard 2GB document collection demonstrate that our approach is scalable in terms of both the number of processing nodes and the number of clusters.

5. ACKNOWLDEGEMENTS

We wish to thank Dr. Xian-He Sun of the Illinois Institute of Technology for his support of this project.

6. REFERENCES

[1] Baker, M., Carpenter, B., Fox, G., Ko, S., and Lim, S. mpijava: An object-oriented java interface to mpi. In *International Workshop on Java for Parallel and Distributed Computing* (April 1999), IPPS/SPDP.

[2] Cutting, D. R., Pedersen, J. O., Karger, D., and Tukey, J. W. Scatter/gather: A cluster-based approach to browsing large document collections. In *Proceedings of 15th Annual ACM-SIGIR* (1992), pp. 318–329.

[3] Dhillon, I. S., Fan, J., and Guan, Y. Efficient clustering of very large document collections. In *Data Mining for Scientific and Engineering Applications*. 2001.

[4] Dhillon, I. S., and Modha, D. S. A data-clustering algorithm on distributed memory multiprocessors. In *Large-Scale Parallel Data Mining, Lecture Notes in Artificial Intelligence* (2000), pp. 245–260.

[5] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing 22*, 6 (1996), 789–828.

[6] Ruocco, A. S., and Frieder, O. Clustering and classification of large document bases in a parallel environment. *JASIS 48*, 10 (1997), 932–943.

[7] English document collections. Tech. rep., NIST Text Retrieval Conference, 2002.