On the Parallel Implementation of Sparse Matrix Information Retrieval Engine

Ankit Jain and Nazli Goharian
Information Retrieval Laboratory
Illinois Institute of Technology
Chicago, Illinois
{ajain, nazli@ir.iit.edu}

Abstract

We demonstrate a parallel implementation of a sparse matrix information retrieval engine. We use a shared nothing PC cluster. We perform our experiments with TREC disk 4 and 5 data, a NIST 2 Gigabytes standard benchmark text collection on 2, 4, 6, 8, 10, 12 and 14 processing nodes with different queries. We compare the results with the results of sequential inverted index, a conventional and common indexing and query processing method. The experimental results are promising and show a significant speedup.

1 Introduction and Prior Work

There are terabytes of text web data and every day the amount of information present on the World Wide Web (WWW) is growing. With the large volume of data, the task of query processing to identify relevant documents can be significantly time consuming. Information explosion demands scalable retrieval systems, which motivates selecting indexing and search algorithms that can support effectiveness and efficiency of the search. A concern with the development of scalable information retrieval (IR) is the design of algorithms that yield acceptable processing speeds when faced with large data sets. Parallel Processing of large volume of data in IR has several advantages such as improving response time and capability of searching larger collections [1].

Inverted Index is the most popular approach for text retrieval search engines. However, the parallization of inverted index is not a trivial task. Thus, parallel processing is not taken advantage of in the processing of large volume of data by search engines. In this paper, we evaluate the result of our implementation of a parallel information retrieval engine as the application of sparse matrix-vector multiplication. The application of sparse matrix-vector multiplication in an IR System was described in the prior work of Goharian, et al. [2] and [3]. They showed that using sparse matrix IR, with compressed sparse row format (CSR), a storage reduction of 35%-40% could be achieved over the storage requirement of the conventional inverted index while the accuracy of the search would remain the same. The readers are referred to [4] for sparse matrix description and formats, and to [5], [6], and [7] for information retrieval topics.

In the remaining part of this paper, we present the parallel implementation, experiments and results. We evaluate our results based on the best sequential approach in IR, i.e., inverted index.

2 Experimental Framework

2.1 Text Collection

We used a standard text collection, TREC disk 4 and 5 data, benchmark data provided by TREC (Text Retrieval Conference) sponsored by National Institute of Standards and Technology (NIST) for our experiments [8]. The text collection is a SGML tagged collection. A sample document from TREC text collection with the SGML tag is shown below.

</DOC>
<DOC>
<DOCNO> FT911-10 </DOCNO>
<HEADLINE>
FT 14 MAY 91 / World News in Brief: Brussels rioting
</HEADLINE>
<DATELINE> BRUSSELS </DATELINE>
<TEXT>
Almost 200 North African immigrants were arrested in Brussels at the weekend during the worst racial rioting ever seen in the normally placid Belgian capital.
</TEXT>
</DOC>
<DOC>

A parser as described in [9] is used to parse the SGML tagged TREC collection. The Parser tokenizes the text and eliminates the stop words. Stop word elimination is a technique to effectively

remove the frequently used words such as *"a"*, *"an"*, and *"the"*. We used a stemmer, i.e., Porter stemmer [10], to stem the words. The stemming process removes the prefixes and suffixes such as *"ing"*, *"ion"*, *" kilo"*, and *"mega"*. These techniques result in a more effective indexing and retrieval system. The parsing and indexing process create the vector representation of text for query processing. Table 1 shows the document collection size, number of documents in the collection, total number of terms in the collection, and the number of distinct terms. The total number of terms in the collection corresponds to the non-zero elements in the sparse matrix.

Table 1- TREC Disk 4 and 5 Text Collection

Document Collection Size	2GB
Number of Files Parsed	2,295
Number of Documents Parsed	524,939
Total Number of Terms, distinct in a document (excluding Stop Terms)	78,896,566
Number of Distinct Terms (excluding Stop Terms)	820,581

2.2 Queries

We used 7 queries from the list of 50 *Topic* TREC queries. *Topic* queries are the short queries that contain 1-3 terms. Each query is numbered. We based our decision for choosing these seven queries, i.e., queries 362, 364, 376, 377, 382, 383, and 389 on the number of documents retrieved by each of the 50 *Topics* queries. We believe that the number of retrieved results for a query could be an indicator of the behavior of the parallel system, as the retrieved results are proportional to the amount of work for query processing. Table 2 shows the number of retrieved results for each of these three queries. The following are two sample queries used in our experiments:

<top>
<num> Number: 362
<title> human smuggling
</top>

<top>
<num> Number: 383
<title> mental illness drugs
</top>

Table 2: Query and Number of Retrieved Documents

Query	Retrieved Results	
364	131	Minimum
377	5628	
382	19208	
362	28617	Average
383	34152	Median
389	56631	
376	114785	Maximum

2.3 Hardware and software requirements:

Tests were conducted on a High Performance 16 nodes Ethernet 10/100 LAN cluster. Each node in the cluster is Dual Pentium III (Coppermine) Intel 1GHz CPU and 1024 Megabyte RAM machine. We used JAVA Remote method invocation (RMI) for the implementation [11].

3 Parallel Implementation

In parallel CSR information retrieval the vectors that hold the indexing information of the text collection can be broken into smaller parts that can be evenly distributed among the nodes. All nodes take part in the query processing, which leads to a good speedup. Figure 1 shows how the CSR vectors can be divided into smaller parts, so that the vector elements belonging to each document are kept together at one node. In this example, the distinct terms of document 1 with term identifiers 1, 2 and 3 in column vector and the corresponding term weight 2, 1, and 1 in non-zero vector can be located on one node. Load balancing in case of inverted index is not a trivial process. The load balancing in Compressed Sparse Row (CSR) matrix results in balanced nodes and can be done at the time of building the index.

Non-zero vector	2	1	1	1	1	1	1	1
Column vector	1	2	3	4	5	6	3	7
Row Vector	3	6	8					

Belongs to one document

Figure 1: Vectors of CSR and the Elements of a Document

In our approach to parallel implementation of CSR information retrieval engine, we describe the following components of our system: load balancing of index for the query processing nodes, architecture of the system, and the query processing and result generation:

3.1 Load Balancing

Now the challenge is to distribute the load equally among the participating nodes. The assumption is that if equal number of terms is on each node, then the workload can be almost equally distributed as all the terms are to be processed in CSR approach. To achieve this, we used the Greedy load allocation algorithm [12] that almost equally distributes the load among all the processors. Figure 2 shows an example of Greedy algorithm load balancing, taking a collection of 9 documents and 3 nodes with different number of distinct terms per node. As the result, node 1 has 56 terms, node 2 has 542 terms and node 3 has 54 terms. Table 3 demonstrates the result of the Greedy load balancing on our 2GB collection, using 15 nodes. The *retrieved results* column shows the number of documents retrieved by each node during processing query 376, which retrieves a total of 114,785 documents. Each node has an average of 5,260,025 terms, a median of 5,269,918 terms, and the standard deviation of 59.126 terms. Note that the number of retrieved results per node is directly proportional to the load on the nodes. The mean, median and standard deviation of retrieved results on each node is 7,652, 7642, and 81.8, respectively.

Document	Distinct terms per document
D1	20
D2	30
D3	10
D4	15
D5	30
D6	11
D7	22
D8	18
D9	6

Node 1	Node 2	Node 3
D1, 20	D2, 30	D3, 10
D5, 30		D4, 15
	D7, 22	D6, 11
D9, 6		D6, 18
56	52	54

Figure 2: Greedy Load Allocation on a Sample Collection

3.2 System Architecture

The parsing of the data collection, the indexing, and the load balancing are done at the server. These processes are sequential and done prior to the query processing. The major tasks during query processing are parsing the query, processing the query and sorting the results, i.e., ranking the computed scores. Figure 3 shows the different components of the architecture for our parallel Sparse Matrix IR that are described further:

Table 3: Greedy Load Allocation of TREC6-7 data on 15 Nodes

Nodes	Load	Retrieved Results
1	5,259,939	7,528
2	5,259,959	7,555
3	5,259,963	7,571
4	5,259,982	7,576
5	5,259,990	7,583
6	5,259,990	7,629
7	5,259,997	7,633
8	5,260,018	7,642
9	5,260,029	7,657
10	5,260,033	7,674
11	5,260,068	7,709
12	5,260,070	7,717
13	5,260,089	7,752
14	5,260,111	7,767
15	5,260,138	7,792
Mean	5,260,025	7,652
Median	5,269,918	7,642
Standard Deviation	59.126	81.8

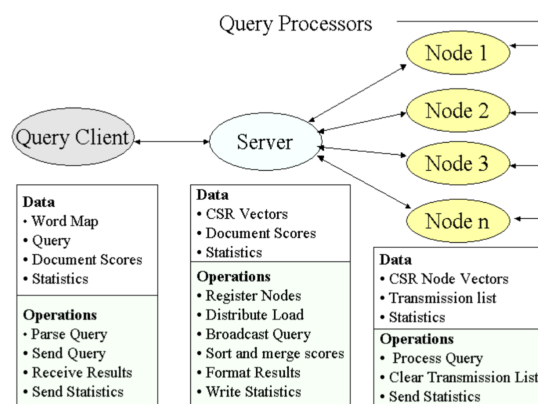

Figure 3: Parallel System Architecture

<u>*Query Client:*</u> Query client receives the query from the user and sends it to the server. Query client waits until the query processors process the query and then receives the document scores form the server and prints the formatted results.

<u>*Server or Master Node*</u>: Server distributes the data, i.e., the elements of the vectors, as determined by Greedy load allocation algorithm to the query processor nodes. Upon receiving the query from the query client, the server sends the query to each query processor node and delegates the nodes to process the query. The server receives the retrieved documents from the query processor nodes. It sorts and merges the results and sends the formatted document scores to the query client.

<u>*Query Processors*</u>: Each query processor node receives the query from the server. The index, i.e., vector elements resides in the memory of each node. Each node processes the query by using the CSR sparse matrix-vector multiplication algorithm described in [2] and [3]. The result of the

multiplication computation is the relevance ranking score demonstrating the relevance of each document to the query. Each document has a score associated with the document. The query processor nodes sort their document scores and send the scores to the server.

3.2.1 Search Protocol and Events on the Time Line

The events that happen during the query processing are shown on the time line, and the algorithm is highlighted in the following five steps:

Step 1) Initialization Step

Java RMI registry starts at the server. It acts as a directory and stores the address of each query processor node. Then the server starts and allocates memory for the recourses and initializes. Query processor nodes start one by one. Each query processor node connects itself to the server and requests for the registration to the server by sending its IP address and host name to the server. The server receives an IP address and host name from a node and returns an identification number, i.e., a node id, to that node. The server stores all IP addresses and host names in a hash table called the registration table (Figure 4).

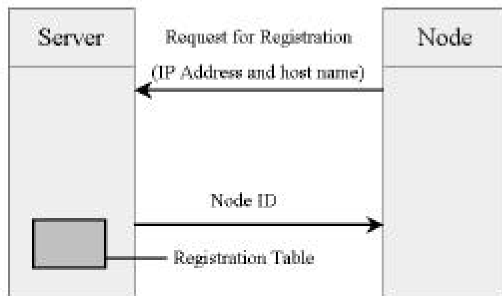

Figure 4: Initialization Step

Step 2) Load Distribution Step

Upon receiving the identification number from the server, each query processing node requests the server for indexed data, i.e., part of the vectors. After receiving the request from the node, the server reads the portion of the vectors that belongs to the respective node from the disk and sends it to the node. The query-processing node receives the part of the vectors and requests the server for a query. Until this point server does not have the query; server puts the node into wait state (Figure 5).

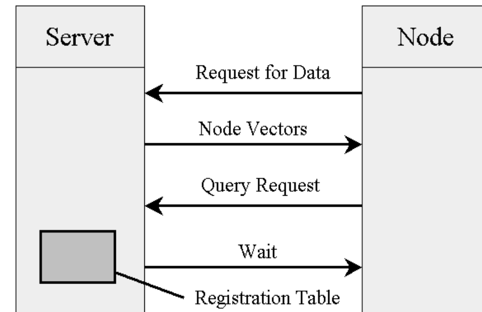

Figure 5: Load Distribution Step

Step 3) Query Parsing and Distribution Step

At this time, the query client starts and requests the mapping of term tokens to term identifiers (word map) from the server to parse the query. The server responds and sends the required data. Query client gets the query from the user, parses it and sends the query vector to the server. The server broadcasts the query to all the query processors. All nodes get back to ready status and start to process the query (Figure 6).

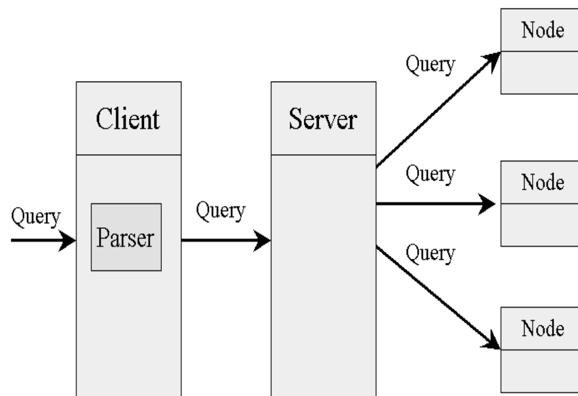

Figure 6: Query Parsing and Distribution Step

Step 4) Query Processing Step

Each query-processing node processes the query on its portion of data and keeps on collecting the relevance ranking scores on its transmission queue. The query processor checks the size of the transmission list and if the transmission list is bigger than the specified transmission limit, it sorts the scores of the transmission list and transmits them to the server and clears the transmission list. The server receives the results, i.e., scores, from the query processor nodes and appends the scores to its score list (Figure 7).

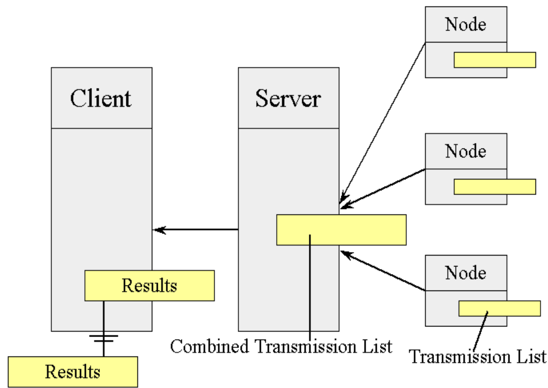

Figure 7: Query Processing Step

Step 5) Final Relevance Ranking Scores Compilation Step

During the whole process, the query client, server and the query processor nodes keep collecting the various statistics. Once a query processor completes the query processing on its node, it sends the final transmission list to the server along with its task completion signal. At this time the query processor node goes in the wait state. When the server receives the done signal from all the query processor nodes, it sorts the scores and formats the results. The server sends the results to the query client along with a done signal. Upon receiving the done signal from server, the query client finally writes the results to the disk and goes to the wait state. Query processors and query client send their statistics to the server and server writes the statistics on the disk. The total time for query processing recorded in every experiment includes the timing for query parsing, processing, results merging, sorting and formatting (Figure 8).

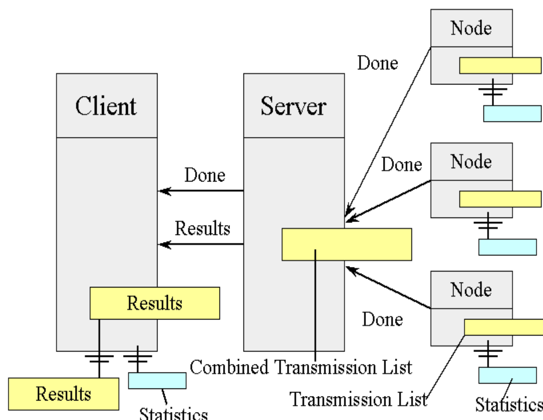

Figure 8: Final Relevance Ranking Scores Compilation Step

4 Experimental Results

We performed our experiments using a 2 GB text collection and different number of nodes, i.e., 2, 4, 6, 8, 10, 12 and 14 nodes to process the queries. Each configuration of the nodes is tested with all seven queries listed earlier in the section of experimental framework. Each experiment with different number of nodes and different queries are also done with five combinations of transmission list sizes. Thus, we gathered the performance results for 245 different experimental runs on the system. As demonstrated in figure 9 the query processing time reaches to a minimum and then increases again if we keep on increasing the number of nodes. The optimum number of nodes for this data collection is 12 nodes.

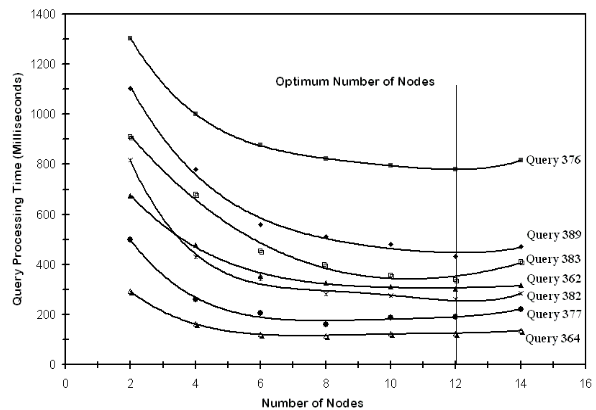

Figure 9: Query Processing Time for different Number of Nodes

Figure 10 shows query processing timing for different sizes of transmission list on node 12. A similar pattern can be noted here as well. The query processing time reaches to a minimum and then increases again if we keep on increasing the transmission list size. On the X-axis the transmission list size is the percentage of the average retrieved results per node and Y-axis is the query processing time.

Our experimental results show that with the optimum number of nodes and optimum size of transmission list, parallel CSR Information Retrieval (IR) outperforms inverted index with substantial margin. Table 4 shows the comparison of the query processing timing on inverted index and parallel CSR using the optimum number of nodes and optimum size of transmission list for our test data

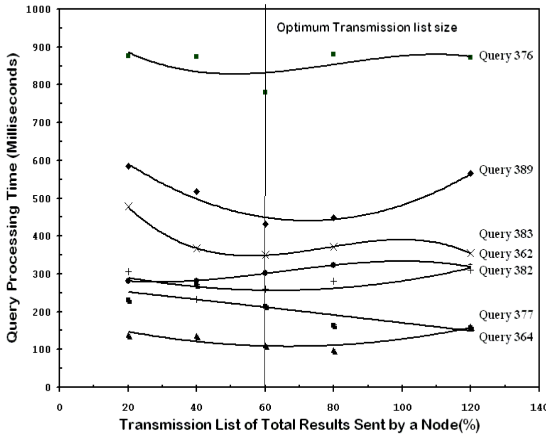

Figure 10: Query Processing time for Transmission List Size using 12 Nodes

Table 4: Query Processing time for Inverted Index and Parallel CSR

Query	Retrieved Results	Query Processing Time Inverted Index milliseconds	Query Processing Time CSR Parallel milliseconds
364	131	9499	165
377	5628	10427	214
382	19208	11460	260
362	28617	13376	302
383	34152	13738	351
389	56631	15071	431
376	114785	20468	779

Figure 11 depicts the nature of parallel CSR query processing with increasing retrieved results, i.e., retrieved documents. Parallel processing overheads, i.e., communication overhead and additional computations increase, if the number of retrieved results increases.

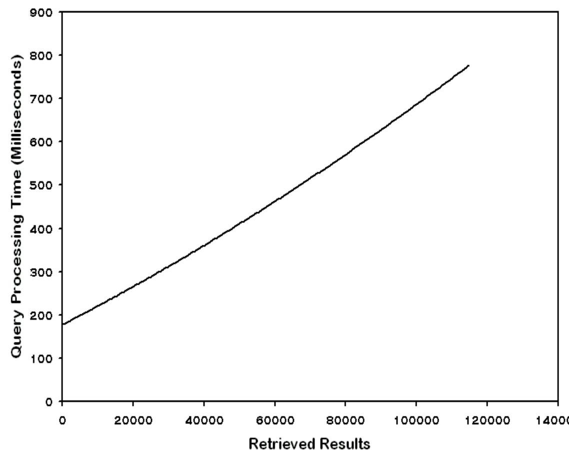

Figure 11: CSR Parallel: Query Processing time vs. Retrieved results

Table 5 shows the speedup and the efficiency achieved with the optimum settings for number of nodes and transmission list size by parallel CSR over inverted index. A super linear speedup is noticed for our experiments. In an inverted index IR system, because of the memory limitations the whole storage structure cannot be kept in the memory at a time and hence query processor reads the posting lists of the query tokens from the disk to process the query. While in the case of the CSR parallel implementation, since many nodes take part in query processing and each node has enough memory to keep it's part of index, the need of I/O at the time of query processing can be omitted. Superior performance of the parallel implementation and the fact that index is accessed from the cache contributes to the super linear speedup. We acknowledge that a fare comparison and measure of speedup requires a fully memory resident of data in the case of inverted index, i.e. running inverted index experiments on a machine with 12 GB memory, for the 12 nodes that participated in parallel experiments each with 1GB. Such a hardware requirement was not available to us. On the other hand, as mentioned earlier the parallel implementation of inverted index is not reported as a successful approach.

Table 5: Speedup and Efficiency

Query	Speedup (S)	Efficiency = S/N
364	57.57	4.8
377	48.72	4.1
382	44.07	3.7
362	44.29	3.7
383	39.13	3.3
389	34.97	2.9
376	26.27	2.2

4.1 Analysis

Query performance is very system specific. Among the machines we used in our experiments, the server had to be of a good configuration. A faster LAN reduces the communication overheads. The queries that retrieve larger number of documents take longer, as the nodes have to clear the transmission list several times, which results in a higher parallel processing overheads. If the transmission list is bigger than the optimum size, the degree of parallelism reduces. At the same time, if transmission list size is smaller than the optimum size, communication overheads deteriorates the performance. The number of nodes participating in the query processing also

affects the performance. For a specific text collection and type of queries, using only the optimum number of nodes is the most advantageous.

The sequential processing component that includes parsing the query and final compilation of the results also affects the total timing. Based on our experiments on different collection sizes, we observed that the sequential processing time remains constant and almost independent of the collection size. As the result of this observation, we expect a better speedup for larger collections and larger number of processing nodes. The significance of optimum transmission list size is a factor to be considered. This threshold can be defined as fixed or determined dynamically on the run time based on the size of collection on which the tests are to be performed. Our experimental results suggest that for our collection the optimum transmission list size should be 60% of the average results retrieved on a node.

One other advantage of our parallel query processing approach is the process of adding a new node to the system takes very minimal time and effort. Based on our experiments, a new node initialization takes 150 seconds. One issue can be raised in our architecture is the repercussions of a node failure. The server will endlessly keep on waiting for the done signal from the failed node and will never be able to complete the processing. One approach to this problem is relaxing the requirements of processing all the documents with the notion that it does not effect the retrieval accuracy as the search engines today only index a fewer percentage of the web. However, a timer can be used on the server to handle this situation. If the timer expires and server receives nothing from the node, server will compile the received document relevance ranking scores and conclude.

5 Conclusion and Future Work

We presented a parallel information retrieval engine based on compressed sparse row matrix-vector multiplication algorithm approach. Since the majority of the current information retrieval systems are based on the inverted index and the proposed approach is an alternative approach to inverted index, we compared our experimental results with Inverted Index. Our experimental results on parallelizing CSR information retrieval system showed a significant speedup compare to inverted index. Another concern regarding inverted index is the complexity of the update. We believe that CSR vectors can be updated easily for adding new documents to the collection without the need of rebuilding the whole index.

References

[1] E. Rasmussen, Parallel Information Processing,. American Society of Information Science,1992.

[2] N. Goharian, T. El-Ghazawi, D. Grossman, A. Chowdhury, On the Enhancements of a Sparse Matrix Information Retrieval Approach, PDPTA'2000.

[3] N. Goharian, T. El-Ghazawi, D. Grossman, "Enterprise Text Processing: A Sparse Matrix Approach", Proc.of the IEEE Int. Conf. on Information Technology: Computing and Coding(ITCC01), LV, 2001.

[4] BLAST Forum, "Documentation for the Basic Linear Algebra Subprograms", http://www.netlib.org/blast/blast-forum,1999.

[5] G. Salton, "Automatic Text Processing", Addison Wesley, Massachusetts, 1989.

[6] D. Grossman and O. Frieder, "Information Retrieval: Algorithm and Heuristics", Kluwer Academic Publishers, 1998.

[7] ACM SIGIR Proceedings, http://sigir.acm.org

[8] Text Retrieval Conference, http://trec.nist.gov

[9] D. Grossman and O. Frieder, "Anatomy of a Search Engine: A Java-based Introduction to Scaleable Information Retrieval",manuscript 2002.

[10] Porter, M. F. An algorithm for suffix stripping. Program Automated Library and Information Systems, 14 (3), 130-137. 1980.

[11] J. Maassen, R. Nieuwpoort, R. Veldema, H. Bal, A. Plaat, An Efficient Implementation of Java's Remote Method Invocation, Seventh ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, Atlanta,1999.

[12] S. Nastea, O. Frieder and T. El-Ghazawi, Load Balanced Sparse Matrix-Vector Multiplication on Parallel Computers, Journal of Parallel and Distributed Computing (JPDC), 46:180-193, 1997.

[13] O. Sornil, Parallel Inverted Index for Large-Scale, Dynamic Digital Libraries, Virginia Tech Computer Science, Blacksburg, Ph. D. Dissertation Draft, 2000.