Parallel Generation of Inverted Files for Distributed Text Collections

Abstract

We present a scalable algorithm for the parallel computation of inverted files for large text collections. The algorithm takes into account an environment of a high bandwidth network of workstations with a shared-nothing memory organization. The text collection is assumed to be evenly distributed among the disks of the various workstations. Compression is used to save space in main memory (where inverted lists are kept) and to save time when data have to be moved across the network. The algorithm average running cost is $O(t/p)$ where t is the size of the whole text collection and p is the number of available processors. We implemented our algorithm and drew experimental results. In a 100 Mbits/s switched Ethernet network with 4 PentiumPro 200 megahertz, 128 megabytes RAM on each processor, we were able to invert 2 gigabytes of TREC documents in 15 minutes. Further, we also proposed an analytical model for the algorithm execution time.

1 Introduction

The potential large size of a full text collection demands specialized indexing techniques for efficient information retrieval (IR). Different index types for text retrieval exist in the literature and have been implemented under different scenarios [4]. Some examples are suffix arrays, inverted files, and signature files. Each of them has strong and weak points. However, *inverted files* have been traditionally the most popular indexing technique used along the years.

Inverted files are useful because, although the size of the index is proportional to the size of the text, their searching strategy is based mostly on the vocabulary (i.e., the set of distinct words in the text) which usually fits in main memory[1]. Inverted files are simple to update and perform well when the pattern to be searched for is formed by conjunctions and disjunctions of simple words which are probably the most common type of query in information retrieval systems. For instance, inverted files are widely used to index the World Wide Web (which we refer to by Web simply).

The best possible sequential algorithm for generating inverted files requires computing time proportional to the text size (since it has to read the whole text at the very least) and might require expensive hardware if fast inversion of large texts is demanded. Since current text collections keep growing both in number and in size, faster indexing algorithms are highly desirable and an alternative is the use of parallel hardware for generating the index.

In this paper we investigate a new scalable algorithm for the distributed parallel generation of large inverted files in the context of a high bandwidth network of workstations. We present the algorithm, analyze its performance, and draw experimental results. A network of workstations provides computing power comparable to that of a typical parallel machine but is more cost effective [1].

Our algorithm assumes that the documents in the text collection are evenly distributed among the processors of the network. The algorithm operates by inverting the local text collections in parallel, computing a global vocabulary, and then exchanging inverted lists among the processors to generate a global inverted file for the whole text collection. The Golomb compression strategy [10] is used to save space in main memory (where inverted files are kept) and to save time when data have to be moved across the network. Using a 100 Mbits/s switched Ethernet network with 4 PentiumPro 200 megahertz, we are able to invert 2 gigabytes of TREC documents in 15 minutes. We also developed an analytical model for the execution time of our parallel program which is fairly accurate. Using this model, we show that the algorithm average running cost is $O(t/p)$, where t is the size of the whole input text collection and p is the number of processors in the network. Besides this low running time with a practical collection, our algorithm scales up well. Due to the high parallelism of computation and communication and to smart algorithms for data exchange, performance is improved.

The paper is organized as follows. We first discuss related work. Following, we present the organization of our distributed text collection. We then discuss the motivation for generating global inverted files from the point of view of query processing. The discussion proceeds with a presentation of our parallel algorithm and its associated analytical model, followed by our results and conclusions.

[1] In a text of size t it is reasonable to expect a vocabulary of size proportional to $\sqrt{t}$ [7].

2 Related Work

The size of a non-compressed inverted file ranges from 30% to 100% of the text size, depending on the implementation. This implies that, for a very large text, the corresponding inverted file has to be stored in disk. Thus, sequential algorithms for generating large inverted files try to minimize disk accesses. To cope with this reality, many approaches have been proposed [6].

One typical technique to minimize disk accesses is to reduce the size of the lists by modifying the *granularity* of the pointers. That is, instead of pointing to every document in which a word w occurs, the list may point only to *blocks* in which that word occurs. Typically, a block may contain many documents specially with collections of many small documents. This technique reduces the space requirements not only because there are less blocks than documents (and hence the pointers can be smaller) but also because all the occurrences of a word in a block are represented once in the inverted list. Using such indices, we need to search sequentially inside blocks if the exact document in the text collection is requested (which is the case with the most common queries). An example of application of this technique is Glimpse [9].

The main problem with reducing the granularity of the pointers is that the index cannot be reduced too much (by increasing the block size) without degrading the performance. This is because it is expensive to search large blocks sequentially for the exact position of a given word. Therefore, the approach does not work well for texts above 200 megabytes in size [3].

Another thread of research is compressing the inverted file as done in [10]. The key idea is to always compress any portion of the inverted file which has to be stored in disk. This reduces the amount of disk space consumed and thus, the number of disk accesses. Fast index generation is reported (e.g., indexing 2 gigabytes of text in 4 hours) with low space consumption (the size of the compressed index is only 10% of the text size). The main reason for such performance is the reduction of the time spent reading/writing data from/to the disk. In our work, we also compress inverted files as in [10].

Parallelization and use of remote primary memory (instead of local disk) also offer a possibility to speed up the generation of indexing structures. Mergesort-based parallel algorithms to generate suffix arrays have been studied in [8]. Recently, a new quicksort-based distributed algorithm for generating suffix arrays has been proposed [11].

The work discussed in this paper uses parallelization to generate a distributed inverted file.

3 Distributed Text Collection

In this section we describe the distributed text collection which operates in our network of workstations.

3.1 Distribution of the Text Collection

For clarity, we focus our attention on identifying the main tradeoffs involved with the distributed generation of inverted files and consider only the case in which the documents in the collection are evenly distributed across the network. Despite the fact that we consider only the case of a homogeneous distribution of documents, our parallel algorithm for generating inverted files can be directly applied to the case of a non-homogeneous distribution.

Let p be the number of machines in the network and t the size (in bytes) of the whole collection. Define,

$$b = \frac{t}{p} \tag{1}$$

Then, considering that the documents are evenly distributed across the network, each machine holds (in its local disk) a subcollection whose size (in bytes) is roughly given by b.

3.2 Distribution of the Inverted File

An *inverted file* is an indexing structure composed of: (a) a list of all distinct words in the text which is usually referred to as the *vocabulary* and (b) for each word w in the vocabulary, an *inverted list* of documents in which the word w occurs. Additionally, the vocabulary is sorted in lexicographical order.

With large texts, some restrictions are imposed on the inverted file to keep it smaller [6]. Examples of these restrictions are: (a) filtering of text characters and separators, and (b) use of a controlled vocabulary in which not all words in the text are indexed (such as *stop words* – e.g., articles and prepositions).

In our distributed text collection, each machine holds a subcollection whose size (in bytes) is roughly b. This implies that, for each subcollection, the corresponding inverted file has size $O(b)$. Thus, for p machines, the size of the index for the whole collection is given by $p \times K_i \times b$ where K_i is a proportionality constant. There are two fundamental basic organizations for this index.

The first organization for the index is to have each machine with its own local inverted file [14, 18]. In this *local index organization*, generating and maintaining the indexes are simple because everything can be done locally without interaction among the machines.

The second possibility is to have a global inverted file for the whole collection. To the best of our knowledge, this is the approach taken by most search engines in the Web. These engines maintain a global library (composed of copies of Web documents) for which a global index is periodically recomputed. This global index is normally maintained in a single, large, central machine. Here, we consider the situation in which this global index is distributed among networked machines [14, 18]. This is called a *global index organization*.

There are many possibilities to distribute the index (for instance, the distribution might be based on a criteria of load balancing). For simplicity, we consider that the global index should be distributed among the machines in lexicographical order such that each machine holds roughly an equal portion of the whole index. According to this strategy, machine 1 might end up holding the global inverted lists for all the keywords which start with the letters A, B, or C, machine 2 might end up holding the global inverted list for all the keywords which start with the letters D, E, F, or G, and so on. The important issue is that each machine holds a portion of the global index of size proportional to the size of the local document collection (b).

4 The Querying Issue

One might wonder whether it is worth worrying about the generation of a global inverted file when local inverted files (which are easier to generate and maintain) can be used for distributed query processing.

Consider that there is a *central broker* machine to which all the queries are first directed. This broker inserts the query requests in a queue and process them from there. Further, consider that there are always enough queries to fill a minimum size query processing queue (as expected in the Web).

In the local index organization, the broker takes a query out of the queue and sends this query to all the machines in the network. Each machine then processes the whole query locally and obtains the set of documents related to that query. Besides, it is also necessary to rank the answer set which can be done, for instance, with the vector space model [15]. After ranking, each machine selects a certain number of documents from the top of the ranking and returns these to the broker as the local answer set. The central broker them collects the r sorted local answer sets and combines them (through a merging sort procedure) into a global (and final) ranked set of documents. By selecting a set of documents from the top of the ranking, each machine reduces the amount of data which has to be sent (to the broker) through the network. However, such reduction in network traffic must not affect the precision of the global answer set. This can be assured through the adoption of a simple cutting strategy as discussed in [12, 13, 14].

Since the indexes are local to each machine in the local index organization, global information on the occurrence of keywords in the collection is missing. Without this information, the estimates for the *inverse document frequency* (*idf*) weights (associated to each term by the vector space model [15]) are slack and, as a result, the generated global ranking suffers from a drop in precision figures [14].

To avoid this, it is necessary to provide each machine in the local index organization with access to global information on keyword distribution. Thus, we notice that, even with a local index organization, computation of global information is unavoidable. This is our first reason for the need to compute (at least in part) a global indexing structure.

In the global index organization, the central broker takes a query out of the queue and first determines which machines hold inverted lists relative to the query terms. Notice that, in this case, not all machines might be involved. The situation here is quite distinct from that with the local index organization [14].

Let us briefly discuss the querying performance for those two organizations. For that, consider again the 50 TREC queries numbered ¿from 101 to 150 and the documents in the disk 1 of the TREC collection [5]. Figure 1, obtained from [14], compares the 50 queries total processing time for the global index (GI) and the local index (LI) organizations at a network speed of 80 Mbits/s. As it can be seen, the global index organization consistently outperforms the local index organization at this network speed. Further, the relative improvement in performance increases with the number of machines, the network bandwidth, and the disks transfer rate [14]. Thus, our second reason for the need to generate global inverted files is that a global index organization outperforms a local index organization in an environment of high bandwidth networks.

Number of machines	Response time (s)		GI as percentage of LI (%)
	LI	GI	
2	21.26	19.64	92.37
4	17.13	11.13	64.98
8	14.58	8.86	60.78
16	13.11	7.50	57.23
32	12.23	7.00	57.21
64	11.78	6.93	58.83

Figure 1: Global (GI) versus local (LI) index: estimated total time for 50 TREC queries.

5 Parallel Algorithm For Generating Inverted Files

In this Section we describe and analyze our parallel algorithm for the distributed generation of large inverted files. For the purpose of explaining the algorithm, the processors are numbered arbitrarily, ¿from 0 to $p-1$.

The algorithm proceeds in three phases:

- Phase 1: *Local Inverted Files.* In this phase, each processor builds an inverted file for its local text.
- Phase 2: *Global Vocabulary.* In this phase, the global vocabulary and the portion of the global inverted file to be held by each processor are determined.
- Phase 3: *Global Distributed Inverted File.* In this phase, portions of the local inverted files (as determined in phase 2) are exchanged to generate the global inverted file.

In the following, we discuss and analyze each of these phases in detail.

5.1 Phase 1: Local Inverted Files

In this first phase, each processor reads its b bytes of text from the local disk and builds the corresponding inverted file. This needs no communication among processors which work in parallel. The main steps in this phase are as follows. In step S1, data are read from disk into a buffer in main memory. In step S2, the data in the buffer (in SGML - *Standard Generalized Markup Language*) are processed by a lexical analysis task (which identifies and marks the words) and by a filtering task (which cuts out stop words). Following, the words are inserted into a hash table whose entries point to the inverted lists for each word. The elements in the inverted list for a word w are pairs (d,f) where d is a document in which the word w occurs and f is the frequency of occurrence. After all the local collection has been processed, the inverted file is consolidated. The inverted lists are compressed, but the local vocabulary is kept uncompressed and unsorted in the hash table. There is no need to sort it in this moment because, in phase 2, new entries are added to the hash table (which avoids duplication of data structures). Everything, but the text input, is done in main memory.

The cost of the algorithm for phase 1 is given roughly by:

$$\begin{array}{rll} t_1 = & b \times ts1 + & (S1) \\ & b \times ts2 & (S2) \end{array} \tag{2}$$

where

b: size (in bytes) of local text collection
$ts1$: average time (in seconds, disk) per byte (S1)
$ts2$: average time (in seconds, cpu) per byte (S2)

$ts1$ and $ts2$ can be derived experimentally.

Notice that our analytical model is based on a linearity assumption and thus, is fairly simple. This linearity assumption is clearly valid for disk accesses whenever the number of seeks is small. Also, the linearity assumption is acceptable for step S2 because we use a hash table with constant access cost per entry (the hash table presents low occupancy and the Golomb compression algorithm has constant cost per byte [10]).

5.2 Phase 2: Global Vocabulary

Once phase 1 is concluded, each processor knows the local vocabulary and the sizes of the inverted lists (relative to the local text collection) for each word. The processors then engage in a vocabulary merging process to determine the global vocabulary and the size of the inverted lists (in the global text) for each word.

The size v in English words of the vocabulary (for a text of size t) can be computed as

$$v = Kt^{\beta} = O(t^{\beta}) \tag{3}$$

where $0 < \beta < 1$ and K is a constant [7].

In step S3, the processors are coupled pairwise. The odd numbered processors transfer all their vocabulary information to the even numbered processors which merge the vocabularies and update the sizes of the inverted lists. This pairing process is then applied recursively until the processor with number 0 is left with all the global vocabulary. This is the only significative step of phase 2.

Recalling that a text of size b has a vocabulary of size v given by Kb^{β} English words we can write that the cost for phase 2 is given roughly by:

$$t_2 = K \sum_{i=0}^{(\log_2 p)-1} S_w (2^i b)^{\beta} \times (ts3 + ts4) \quad (S3) \tag{4}$$

where

S_w: average size in bytes of English words
$ts3$: average time (in seconds, network) per byte (S3)
$ts4$: average time (in seconds, cpu) per byte (S3)
p: the number of processors

The factor 2^i in step S3 accounts for the factor that, at the *ith* step, the text size is roughly $2^i b$. $ts3$ is the effective communication time and $ts4$ is the time spent inserting words into the hash. S_w is used to convert vocabulary size from number of English words to number of bytes.

From [2], we take that $K = 4.8$, $\beta = 0.56$, and $S_w = 6$. The first two values were obtained considering only the disk 1 of the TREC collection [5] but should suffice.

5.3 Phase 3: Global Distributed Inverted File

The first part of phase 3 (step S4) takes place only in processor 0 and corresponds to a global vocabulary sorting and the computation of the lexicographical boundaries of p equal-sized stripes of the global inverted file – one for each processor. This striping information is broadcasted to all processors with negligible cost. After this broadcast, each processor knows the stripe of the global inverted file which will be held by any other processor. Therefore, each processor knows to which node it has to transfer each portion of its local inverted file. Once every processor knows which part of its local inverted file must be transferred to which processor, an exchange sequence is planned. Since each processor needs to talk to all others, we can adopt a clever step-by-step all-to-all communication procedure devised in [17]. Each time two processors are paired together, they exchange local inverted files relative to the stripes determined in phase 2.

Looking more carefully at the last steps of phase 3, each machine receives from processor 0 information concerning the final partition of the global inverted file. Each processor then sorts its local vocabulary (step S5). A binary search, with negligible cost, is then performed in order to identify the stripes that should be exported. In step S6, stripes are exchanged with the current peer processor. Communication parallelism is exploited at maximum. The final local inverted file can be left in primary memory or written to disk.

Notice that, once two processors are paired together, they exchange compressed stripes of inverted files in both directions sequentially. Thus, each processor needs to be paired with each other only once. If the network is based on, for example, a switch, pairs of processors exchange data independently one from each other with *no network contention.*

The total number of pairing rounds in the all-to-all communication is equal to $p-1$ (cf. [17]). The cost of phase 3 is then given by

$$\begin{aligned} t_3 = \; & K_q \times v_g \log v_g \times ts5 + && (S4) \\ & K_q \times v_l \log v_l \times ts5 + && (S5) \\ & (p-1) K_c \times \tfrac{2(K_i b)}{p} \times (ts6 + ts7) && (S6) \end{aligned} \tag{5}$$

where

v_l: size (in English words) of the local vocabulary
v_g: size (in English words) of the global vocabulary
K_q: proportionality constant for quicksorting
K_c: compression factor which accounts for the reduction in size due to compression
K_i: ratio between the inverted list size and the corresponding text size
$ts5$: average time (in seconds, cpu) per English word (S4 e S5)

$ts6$: average time (in seconds, network) per byte (S6)
$ts7$: average time (in seconds, cpu) per byte (S6)

The factor K_c accounts for the reduction in space due to the fact that we move compressed inverted lists. The factor 2 which multiplies $\frac{K_i b}{p}$ accounts for the fact that each pairing involves a bidirectional communication. At each pairing, the amount of unidirectional communication is approximately equal to $K_c \frac{K_i b}{p}$. The upper limit of the number of transferred bytes in the whole step S6 (in the perfect homogeneous case and for a given processor) is $2K_c K_i b$.

5.4 Average Total Cost

Let I stand for computation internal costs and C for communication (or network) costs (and considering that the cost of disk I/O is comparable to the cost of transferring data across the network). The total cost order for our parallel algorithm is given by

$$\begin{array}{ll} O(b)\mathbf{I} + O(b)\mathbf{C} + & \text{(Phase 1)} \\ O(t^\beta)\mathbf{I} + O(t^\beta)\mathbf{C} + & \text{(Phase 2)} \\ O(t^\beta \log t^\beta)\mathbf{I} + O(b)\mathbf{C} & \text{(Phase 3)} \end{array}$$

By observing that $b >> t^\beta$ for common English texts (see Section 6), we conclude that the average cost of our algorithm is expected to be

$$O(b)\mathbf{I} + O(b)\mathbf{C} = O(\frac{t}{p})\mathbf{I} + O(\frac{t}{p})\mathbf{C} \quad (6)$$

Such result shows that, for common English texts, our parallel algorithm is expected to scale up nicely. In fact, for a given text of size t, increasing the degree of parallelism in our network by a factor α (i.e., increasing the number of workstations to $\alpha * p$), without violating the condition that $b >> t^\beta$, leads to an expected reduction in the execution time of our algorithm by a factor of $1/\alpha$. This is because b is also reduced by a factor of $1/\alpha$. Further, if one doubles the size of the text and also doubles the number of processors available, the overall average cost of our algorithm is expected to remain constant because b remains constant. Our experiments below corroborate this observation. The only step that tends to be not scalable is phase 2 where parallelism decreases as the number of parallel processor pairs is gradually reduced. However the total communication time is $O(t^\beta)$, that is, in the order of the global vocabulary size, which becomes stable for larger input text collections.

6 Experimental and Analytical Results

In this section, we present experimental and analytical results for our algorithm.

6.1 Experimental Environment

All of our experiments were done on a network of 4 PentiumPro 200 megahertz interconnected by a 100 Mbits/s switched Ethernet. Each processor has 128 megabytes of RAM. The disks

used are Samsung IDE disks, with nominal transfer rates in the order of 50 Mbits/s. At the software level, communication is handled by PVM [16], a message passing library on top of TCP/IP. The text collection we used was composed of documents extracted from disks 1, 2, and 3 of the TREC collection [5]. All five sub-collections in those disks (i.e., AP, DOE, FR, WSJ, and ZIFF) were used.

Since the number of machines is small, we first use our experiments to validate the analytical model. Then, we use the analytical model to study the behavior of the algorithm for larger collections and larger networks.

6.2 Validating Assumptions

We have two basic assumptions which need corroboration: (1) that the amount of data exchanged during each pairing at phase 3 of our algorithm can be approximated by $\frac{K_c(K_i b)}{p}$ and (2) that $b >> t^\beta$.

In order to validate the first assumption, we executed our algorithm for different text sizes and different number of PVM virtual processors (not physical processors – one physical processor can support more than one virtual processor) and measured the number of bytes transferred in phase 3. We notice that each processor j (in the horizontal axis of Figure 2) receives roughly the same number of bytes from each other processor, that is, $\frac{K_c(K_i b)}{p}$, according to the notation of Section 5. If we consider $K_c = 0.25$ (measured experimentally in this work), $K_i = 0.6$ [2], $p = 8$, and $b = \frac{512}{8} = 64$ megabytes, the estimated block size is 1.20 megabytes (cf. Figure 2).

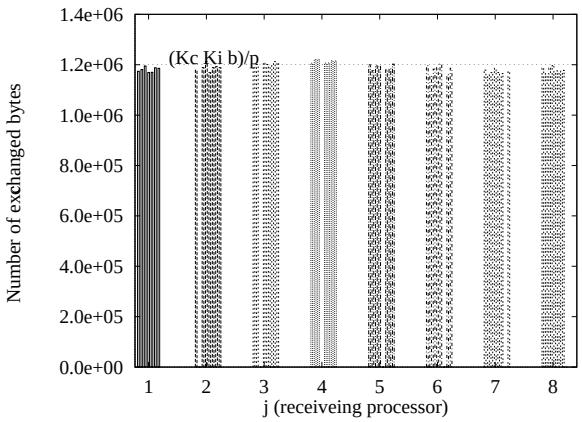

Figure 2: Data exchanged among p processors in phase 3 ($p = 8$), t=512 megabytes.

Regarding the second assumption, Table 1 shows t^β and b_i, $i = 2, 4, 8, 16$, for a portion of the WSJ collection. The data confirm that the assumption is realistic.

6.3 Experimental Results

We generated inverted files for subsets of documents of the TREC collection using 1, 2, and 4 machines. The measured execution times for different sizes of the subsets (t) are presented in Figure 3. We observe that, for this space of experiments, for a given processor, the variation

File	t^{β}	b_2	b_4	b_8	b_{16}
WSJ	4,172.62	82,238,869.50	41,119,370.75	20,559,655.38	10,279,782.19

Table 1: Validating assumption: $b >> t^{\beta}$.

of the execution time is linear, except for small input collections (less than 32 megabytes), where communication becomes a bottleneck.

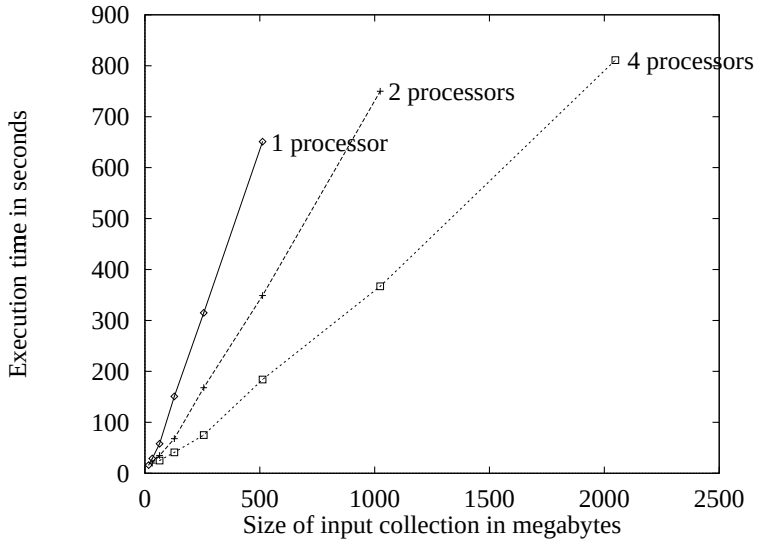

Figure 3: Execution time, in seconds, of the parallel generation of inverted files *versus* size t, in megabytes, of the input collection.

We also plotted the variation of the execution time varying the number of processors (p), but keeping constant the local text size (b) (cf. Figure 4). This means that, *for different processors* in the graph and the same b, the input collection size is pb. The curves are quite coincident, as expected. The differences are due to the communication overhead (absent when $p = 1$ and of increasing importance for larger p).

In Figure 5, we plot the memory utilization per processor, considering different b. The size of the compressed inverted list in main memory (the largest data structure during the execution) corresponds roughly to 15% of b. This matches with a K_c of 0.25 and a K_i of 0.6, presented in the Subsection 6.4.

In Figures 6 and 7, we remark that the percentages of execution times of each phase (when compared to the sum of the execution times of each phase) tend to become stable when increasing b. For large collection local blocks, in this scenario, *phase 1* tend to dominate. Also, for larger number of processors, communication becomes more and more representative. Considering our limited experimental environment, only the analytical model can tell us what happens when we have more processors available.

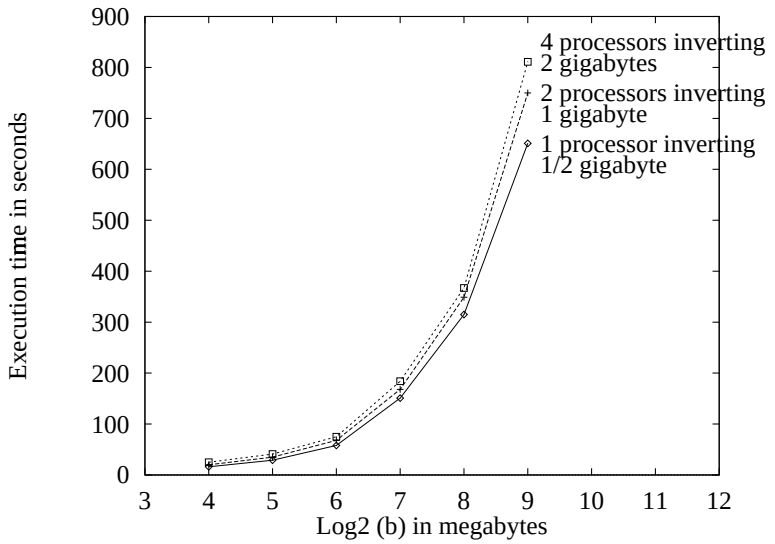

Figure 4: Execution time *versus* local text size b for different number of processors p.

6.4 Analytical Results

In Figures 8 and 9, we compare the measured execution times with those calculated by the analytical model developed in Section 5. The following constants are used (cf. Section 5 for notation).

given a text size t, the vocabulary size is $4.8t^{0.56}$ [2] English words
$ts1$ values 3.1×10^{-7} s/byte (or a IDE disk bandwidth of 3.2 megabytes/s)
$ts2$ values 7.7×10^{-7} s/byte
$ts3 = ts6$ values 1.18×10^{-7} s/byte (network bandwidth of around 65 Mbits/s)
$ts4$ values 1.56×10^{-6} s/byte
$ts7$ values 2.5×10^{-7} s/byte
$ts5 \times K_q$ values 5.43×10^{-7} s/word
K_c values 0.25
K_i values 0.6
$S_w = 10$ in S3 (6 bytes plus 4 control bytes)

We verify that the model estimates are close of the measured data, even taking into account that some model parameters are computed analytically (e.g., the vocabulary size). Other model constants like K_c and K_i are reasonable values, but they can vary, depending on the input text type and size. Therefore, the analytical model should be used carefully. The two tables below present some figures derived ¿from the model. In the first table, we vary p keeping t in 32 gigabytes. In the second table, we also vary p, but we keep b in 512 megabytes.

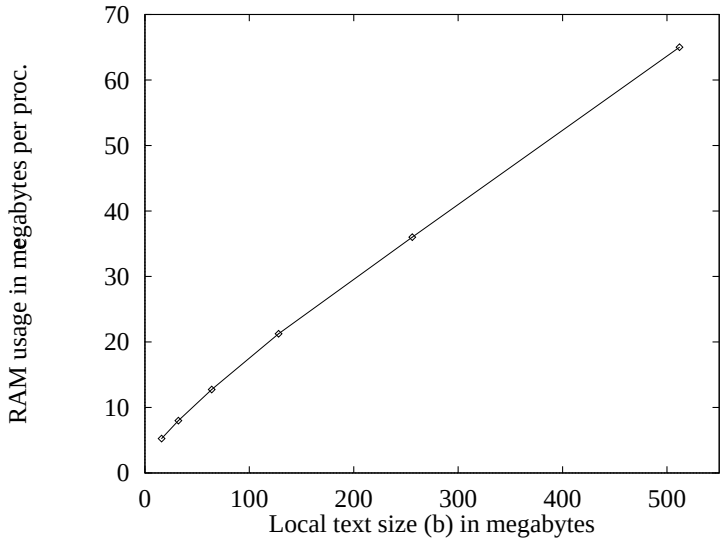

Figure 5: Primary memory utilization per processor *versus* b.

p	b (gigabytes)	RAM utilization (gigabytes)	estimated exec time (min)	perc. phase 1 (%)
2	16	2.4	327.02	94.56
4	8	1.2	168.77	91.61
8	4	0.6	86.74	89.12
16	2	0.3	45.03	85.84
32	1	0.15	24.02	80.47
64	0.5	0.075	13.48	71.67

p	t (gigabytes)	estimated exec time (min)	perc. phase 1 (%)
2	1	10.42	92.76
4	2	10.87	88.92
8	4	11.29	85.59
16	8	11.80	81.91
32	16	12.49	77.38
64	32	13.48	71.67

In the first table, increasing the number of processors from 2 to 32, the speedup in execution time is 24, that is 75% of the ideal (32). With larger p, communication increases and phase 1 becomes less important. However, due to the scalability of the algorithm, this reduction is not linear with p. In the second table, keeping the same local b, estimated execution time increase (from $p = 2$ to $p = 64$) is around 30%. As in the first table, the percentage of phase 1 decreases, due to the more communication overhead for larger p.

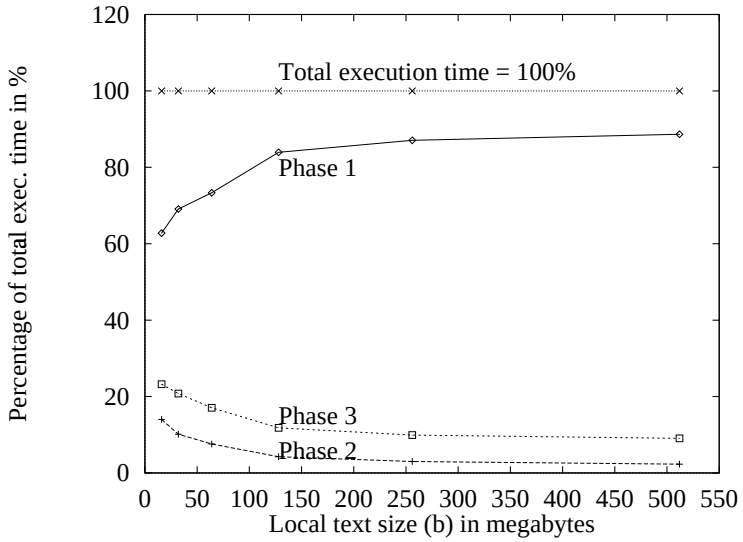

Figure 6: Percentage of the total execution time *versus* local collection block size (b) for $p = 4$.

7 Conclusions

In this paper we investigated a new scalable algorithm for the distributed parallel generation of large inverted files in the context of a high bandwidth network of workstations. The algorithm cost is $O(\frac{t}{p})$ where t is the input collection size and p is the number of processors. We designed, implemented, and evaluated our parallel algorithm using real text data and a fast network. An example of the power of the proposed algorithm is the inversion of 2 gigabytes in 15 minutes.

Our experiments confirm that: (1) our analytical model is appropriate and provides accurate performance predictions and (2) the cost of our algorithm varies linearly with the size of the local text.

The good performance of our algorithm is due to two main reasons. First, it works solely on main memory which is used to store the whole inverted file (the only I/O operations are the reading of the initial text and the writing of the final inverted lists). Second, it is parallel. Clearly, this implies that large memory and many workstations should be available in order to invert larger collections. While this is not the case in an individual basis, the scenario is quite different with the emergence of high bandwidth networks of *cheap* workstations. With such networks, computing power and large memory should be highly available.

We intend to proceed our study in two basic directions. First, we would like to modify our algorithm to also operate in disk. This would allow inverting text collections whose size far exceeds the space available in main memory. Second, we plan to run our algorithm in an IBM SP with 32 nodes in order to evaluate our algorithm in larger parallel systems.

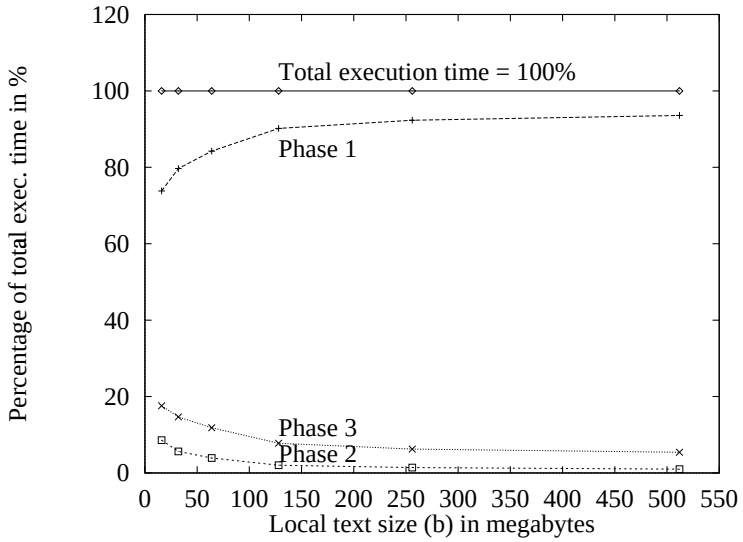

Figure 7: Percentage of the total execution time *versus* local collection block size (b) for $p = 2$.

References

[1] T. Anderson, D. Culler, and D. Patterson. A case for NOW (network of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] M.D. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In Ricardo Baeza-Yates, editor, *IV South American Workshop on String Processing - WSP97 - International Informatics Series*, volume 8, pages 2–20, Valparaíso, Chile, November 1997. Carleton University Press.

[3] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. In F. Golshani and K. Makki, editors, *Proceedings of the Sixth ACM International Conference on Information and Knowledge Management (CIKM'97)*, pages 1–8, Las Vegas, USA, 1997.

[4] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval – Data Structures & Algorithms*. Prentice Hall, 1992.

[5] D. Harman. Overview of the third text retrieval conference. In *Proceedings of the Third Text Retrieval Conference - TREC-3*, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology. NIST Special Publication 500-225.

[6] D. Harman, E. Fox, R. Baeza-Yates, and W. Lee. Inverted files. In W.B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 3. Prentice Hall, 1992.

[7] J. Heaps. *Information Retrieval - Computational and Theoretical Aspects*. Academic Press, NY, 1978.

[8] J. P. Kitajima, M. D. Resende, B. Ribeiro-Neto, and N. Ziviani. Distributed parallel generation of indices for very large text databases. In A. Goscinski, M. Hobbs, and

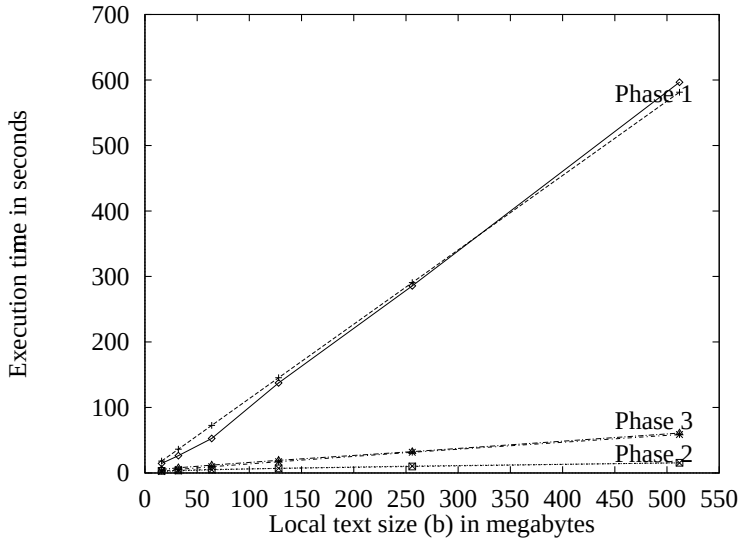

Figure 8: Comparison between measured data and outputs of analytical model for $p = 4$.

W. Zhou, editors, *Proceedings of the 1997 3rd International Conference on Algorithms and Architectures for Parallel Processing – ICA3PP*, pages 745–752, Melbourne, Australia, December 1997. World Scientific.

[9] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. Technical Report 93-34, Dept. of Computer Science, Univ, of Arizona, October 1993.

[10] A. Moffat and T.A.H. Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, 1995.

[11] G. Navarro, J. P. Kitajima, B. Ribeiro-Neto, and N. Ziviani. Distributed generation of suffix arrays. In A. Apostolico and J. Hein, editors, *Lecture Notes in Computer Science*, volume 1264, pages 102–115, Aarhus, Denmark, June 1997. Springer. Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM).

[12] National Institute of Standards and Technology. Proceedings of the text retrieval conference (TREC), November 1992.

[13] M. Persin. Document filtering for fast ranking. In *Proc. of the 17th ACM SIGIR Conference*, pages 339–348. Springer Verlag, July 1994.

[14] B. Ribeiro-Neto and R. Barbosa. Query performance for tightly coupled distributed digital libraries. *Submitted* to the ACM Digital Libraries Conference, 1998.

[15] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Co., New York, 1983.

[16] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: evolution, experiences, and trends. *Parallel Computing*, 20(4):531–546, April 1994.

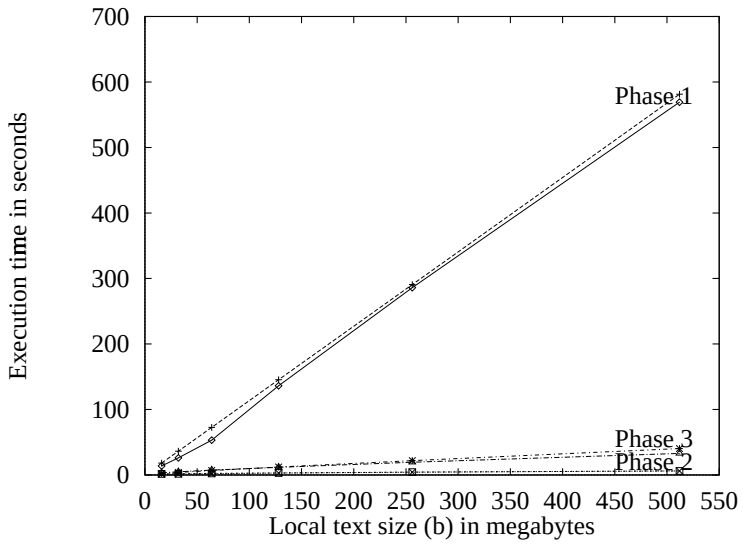

Figure 9: Comparison between measured data and outputs of analytical model for $p = 2$.

[17] T.B. Tabe, J.P. Hardwick, and Q.F. Stout. Statistical analysis of communication time on the IBM SP2. *Computing Science and Statistics*, 27:347–351, 1995.

[18] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, 1993.