Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Parallel Generation of Inverted Lists on a Network of Workstations

Berthier Ribeiro-Neto
João Paulo Kitajima
Gonzalo Navarro
Nivio Ziviani

Relatório Técnico 009/97[1]

April 1997

[1]Available from http://www.dcc.ufmg.br/~nivio

Parallel Generation of Inverted Lists on a Network of Workstations

Berthier A. Ribeiro-Neto † ‡
João Paulo Kitajima * §
Gonzalo Navarro ¶
Nivio Ziviani * ‖

Abstract

We present an algorithm for the parallel computation of inverted lists for large texts. The algorithm operates in the environment of a high-bandwidth network of workstations whose local memories compose a large distributed shared memory. Our idea is to store the inverted list in the large shared memory as it is generated. The algorithm cost is $O(n/r)$ where n is the text size and r is the number of available processors. Thus, the algorithm scales up very well because doubling both the text size and the number of available workstations does not affect its execution time. Using real text data, we simulated the crucial steps of our algorithm and confirmed our analytical predictions. Preliminary estimates on the algorithm expected running time are also drawn.

1 Introduction

The traditional model of text in information retrieval (IR) is that of a collection of documents indexed by keywords. In this model, the user specifies his information need by providing sets of keywords and the information system retrieves the documents which best approximate the user query. Further, the information system might attempt to rank the retrieved documents using some measure of relevance.

Representing the content of a document and of a user query by a set of keywords was necessary in the early seventies due to performance constraints. Nowadays, the advent of powerful workstations has allowed the consideration of alternative models for IR. One such model which is gaining popularity is the *full text* model. In this model, documents are represented by either their complete full text or extended abstracts. The user expresses his information need by providing strings to be matched and the information system retrieves those documents which contain the user specified strings. The cost of searching the full text is usually high but the method presents important advantages [8]. First, no structure in the text is needed which broadens the scope of applications of full text search. Second, no keywords are used which broadens the domain of queries the user might specify. Third, the model is simpler and can be easily grasped by a common user.

The potential large size of a full text collection demands specialized indexing techniques for efficient retrieval. A number of index types for text retrieval exist in the literature and have been implemented under different scenarios [7]. Some examples are suffix arrays, inverted

†Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Brazil.
‡This author has been partially supported by CNPq Project 300188/95-1.
§This author has been partially supported by CNPq Project 300815/94-8.
¶Departamento de Ciencias de la Computación, Universidad de Chile, Chile.
‖This author has been partially supported by CNPq Project 520916/94-8 and Project Ritos/Cyted.

lists, and signature files. Each of them have their own strong and weak points. However, *inverted lists* have been traditionally the most popular indexing technique used along the years.

Inverted lists 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. Further, inverted lists are simple to update and perform well when the pattern to be searched for is formed by conjunctions and disjunctions of simple words. This is probably the most common type of query in IR. For example, inverted lists are widely used to index the World Wide Web (which we refer to by Web simply).

Despite their simple structure, the task of building inverted lists for very large text (such as the whole Web) is costly. Even the best possible sequential algorithm for generating inverted lists requires computing time proportional to the text size (since it has to read the whole text at the very least). This might be prohibitive if the large text keeps changing and the index needs to be updated frequently (as in the Web). Faster indexing algorithms are highly desirable and an obvious alternative is the use of parallel hardware for generating the index.

As a platform for parallel computing, we opt for a high-bandwidth network of workstations. This is an attractive alternative nowadays due to the following reasons. First, the emergent fast switching technology provides very fast message exchanges among the various machines and consequently less parallelism overhead [5]. Second, a network of workstations provides computing power comparable to that of a typical parallel machine but is more cost effective [2, 6]. Third, these workstations are already available elsewhere and most of them are completely idle most of the time [2, 6].

In this paper we investigate a new algorithm for the distributed parallel generation of large inverted lists in the context of a high-bandwidth network of workstations. We present the algorithm, analyze its performance, and draw some experimental results. The algorithm cost is $O(b)$ where b is the size of the text at each processor. Using real text data, we simulated the crucial steps of our algorithm and confirmed our analytical predictions. We also did some estimates on the expected execution time of our algorithm when it runs on 16 workstations with 64Mbytes of free local memory each. These estimates suggest that, in the context of a real distributed shared memory system, our algorithm should be able to invert a text of size 1Gbyte in less than 20 minutes.

Besides this very low expected running time in practical situations, our algorithm presents a couple of nice properties. First, it scales up very well. In fact, if one doubles the text size and also doubles the number of available workstations, the running time does not increase. This is because the amount of local text remains constant. Second, higher parallelism improves performance. In fact, with a constant text size, increasing the number of available workstations reduces the expected running time. This is because a higher number of workstations implies that the amount b of local text at each machine is smaller. This improvement, however, is limited by the size of the vocabulary (i.e., the set of distinct words in the text) because the vocabulary is maintained in memory throughout the computation. Further, increasing the number of workstations excessively partitions the inverted list in too many pieces which might compromise querying operations.

2 Basic Concepts

An *inverted list* 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, a list of all positions in the text in which the word w occurs. Additionally, the vocabulary is sorted in lexicographical order. Since a position in the text is normally indicated by an *offset* counted from the beginning of the text, the list of all positions in which the word w occurs is simply a list of offsets sorted by their increasing values. Figure 1 illustrates a small portion of text and its corresponding inverted list.

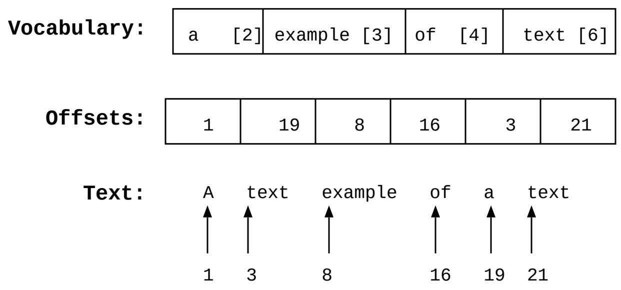

Figure 1: A small text and its inverted list. Each word in the vocabulary includes a pointer to the end of its corresponding inverted list.

The whole inverted list occupies space which is proportional to the text size, although the vocabulary is small and fits in main memory. For instance, in a text with n words it is reasonable to expect a vocabulary of size proportional to $\sqrt{n}$ [3, 11]. Further, with large texts, some restrictions are imposed on the inverted list to keep it smaller [10]. Examples of these restrictions are: (a) filtering of text characters and separators, (b) use of a controlled vocabulary in which not all words in the text are indexed, and (c) exclusion from the vocabulary of *stop words* such as articles and prepositions. The main disadvantage of these restrictions is that words excluded from the index but present in the text are not searchable.

Generation of the inverted list takes time at least proportional to the text size due to the need to read the entire text. Thus, for very large text, parallel algorithms for accomplishing the task are attractive. This is particularly true whenever computing power and memory are highly available. While this is not the case when one looks at individual workstations, the scenario changes in a network of modern workstations. In fact, recent experiments showed that, even during the daytime hours, more than 60% of the workstations in a local network are available 100% of the time [2]. Therefore, it is quite reasonable to expect much greater availability of computing power and large memory space with the emerging high-bandwidth networks of workstations.

In a high-bandwidth network, exchanging messages among workstations is fast which makes it attractive to look at the aggregate memory in all workstations as a large distributed shared memory [1, 2]. Our idea is to use this large distributed shared memory to store the whole inverted list and use the attached workstations as processing nodes of a distributed parallel machine. Figure 2 illustrates this distributed shared memory environment.

At first glance, one might think that the main advantage of the environment in Figure 2 is that our algorithm operates solely on one primary aggregate memory. However, accessing

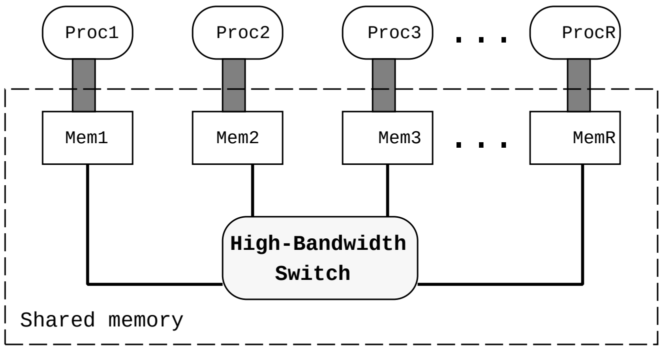

Figure 2: Distributed shared memory.

remote data in current distributed shared memory systems is expensive due to network and operating system overhead [2]. In fact, with commercially available ATM network technology, it can be shown that the time to access remote data in the aggregate memory is still similar to the time to access data sequentially from a local disk at the very best [12, 17]. The advantages provided by the distributed shared memory environment come from two distinct features instead. First, the distributed memory allows random data accesses at uniform cost which we do not have with the local disk. Second, the availability of multiple workstations allows splitting our problem in smaller parts and working on these in parallel.

3 Related Work

The size of an inverted list ranges from 30% to 100% of the text size, depending on the implementation. This implies that, for very large text, the list has to be stored in disk. Thus, sequential algorithms for generating large inverted lists try to minimize disk accesses. To cope with this reality, many approaches have been proposed [10].

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 each occurrence of a word, the list may point only to the files which contain that word. Also, fixed blocks of a given size can be used instead of files. This technique reduces the space requirements not only because there are less blocks or files than words (and hence the pointers can be smaller) but also because all the occurrences of a word in a given block or file are represented only once in the inverted list. Indices of this kind need to search sequentially inside blocks or files if the exact position of a word in the text is requested. Examples of application of this technique are Glimpse [13] and the work in [4]. Glimpse gives good examples of the number of sophisticated queries that can be solved by manipulating the vocabulary.

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 [4]. 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 200Mbytes in size.

Another thread of research is compression, where probably the best exponent is [14]. The key idea in [14] is to always compress any portion of the inverted list which has to be stored in disk. This reduces the amount of disk space consumed and thus, also reduces the number of disk accesses. Further, the pointers point to documents in which a word occurs and not to the positions in the text (i.e., inverted files, not inverted lists, are generated). Since the documents considered are quite small (e.g., 1Kbyte), they can be efficiently searched sequentially and present no serious inconvenience. The authors report very fast index generation (e.g., indexing 2 Gbytes of text in 4 hours) and low space consumption (the size of the compressed index is only 10% of the text size). The main reason for such impressive performance is the reduction of the time spent reading/writing data from/to the disk. Another recent work on the use of compression for indexing, oriented towards suffix arrays instead of inverted lists though, is [15].

Parallelization also offers a possibility to speed up the generation of indexing structures. Mergesort-based parallel algorithms to generate suffix arrays have been studied in [12, 17]. Recently, a new quicksort-based distributed algorithm for generating suffix arrays has been proposed [16]. Such algorithm is $O(b \log^2 r)$ where r is the number of processors required and b is the amount of local memory (in each processor) required to hold the whole text of size n (i.e., $n = b * r$). This is quite better, for instance, than the sequential algorithm discussed in [8] whose time complexity is $O(n^2/m)$, where m is the size of the memory. The work discussed in this paper also uses parallelization to obtain a distributed index. However, here the focus is on the generation of inverted lists.

4 Parallel Algorithm For Generating Inverted Lists

In this Section we describe and analyze a parallel algorithm for the distributed generation of large inverted lists. The algorithm operates on a high-speed network of workstations whose memories compose a distributed shared aggregate memory (as in Figure 2). Let r be the number of processors (one per workstation) available for our computation and let n be the size (in bytes) of the text to be indexed. Define,

$$b = \frac{n}{r} \tag{1}$$

The algorithm we describe can be generalized to consider non-homogeneous sizes for the local texts, although we describe here only the homogeneous case for simplicity.

Our parallel algorithm operates on the distributed shared memory and only accesses disk to read the text in the beginning (if the local memory is insufficient to hold the respective local inverted lists, it uses the disk also to hold the resulting local index). The algorithm assumes that the text is spread across the disks of the r processors with each local disk storing a text of b bytes. Further, the inverted list for the whole text is spread across the memories of the r processors with each local memory storing nearly b list bytes. Since the inverted list is divided among the various processors, each processor holds a lexicographical interval of the vocabulary at the end. This turns querying into a fast operation because of three reasons. First, each processor knows which processor to contact to solve a given query. Second, for each queried word only one processor has to be contacted. Third, disjoint queries can be performed completely in parallel.

In the following we describe the algorithm and analyze its cost. The analysis is divided in two parts: internal CPU times (which are identified by the label **I**), and communication times (which are identified by the label **C**). For the purpose of explaining the algorithm, the processors are numbered arbitrarily, from 0 to $r - 1$.

The algorithm proceeds in three steps:

- Step 1: *Local Inverted Lists.* In this step, each processor builds an inverted list for its local text.
- Step 2: *Global Vocabulary.* In this step, the global vocabulary and the portion of the global inverted list to be held by each processor are determined.
- Step 3: *Global Distributed Inverted List.* In this step, portions of the local lists are exchanged to distribute the global inverted list as determined in step 2.

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

4.1 Step 1: Local Inverted Lists

In this first step, each processor reads its b bytes of text from the local disk and builds the corresponding inverted list. This needs no communication among processors and is done in parallel. The positions of the words in the text are translated to reflect their positions in the global text.

In a high-bandwidth network, accessing local disk takes time similar to accessing remote data in the aggregate memory [12, 17]. Thus, disk access time can be computed as if it were communication time. As a result, reading text from the local disk takes time $O(b)\mathbf{C}$. By using a hash table of words, the building of the inverted list takes $O(b)\mathbf{I}$ on average, while the worst case can also be $O(b)\mathbf{I}$ (using a trie).

Therefore, the total cost for this step is $O(b)(\mathbf{I} + \mathbf{C})$ on the average and worst cases.

4.2 Step 2: Global Vocabulary

Once step 1 is concluded, each processor knows the local vocabulary and the sizes of the list of occurrences (in the local text) for each word. The processors then engage in a vocabulary merging process to determine the global vocabulary and the size of the list of occurrences (in the global text) for each word. This process is illustrated in Figure 3 and is done as follows.

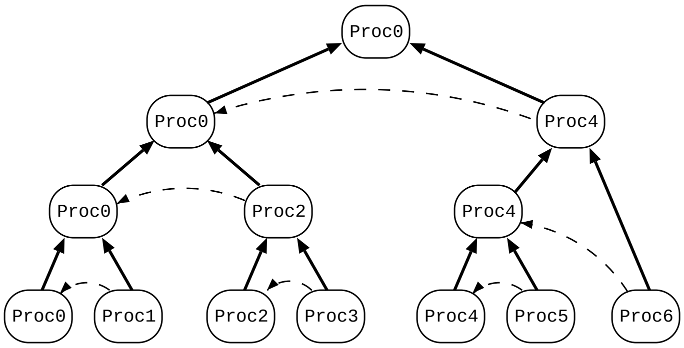

Figure 3: Construction of a global vocabulary by seven processors.

In a first stage, the processors are coupled pairwise. Processor 0 is paired with processor 1, processor 2 is paired with processor 3, and so on. Then, the largest numbered processor in each pair transfers all its vocabulary information to the smallest numbered processor in the pair which merges the two vocabularies and updates the sizes of the lists of occurrences. At the end of this stage, the even-numbered processors hold all the vocabulary information concerning the global text.

In a second stage, this pairing process is repeated using only the even-numbered processors. At the end of this second stage, only processors numbered with multiples of 4 remain. At this point, these processors together hold all the vocabulary information concerning the global text.

This pairing process is applied recursively until the processor with number 0 is left with all the global vocabulary. At this point, processor 0 has all the information to determine the boundaries of r equal-sized parts of the global inverted list — one for each processor. This information is then broadcasted to all processors. After this broadcast, each processor knows the portion of the global inverted list which will be held by any other processor, and therefore it knows to whom it has to transfer each portion of its local list.

As shown in [3, 11], a text of size n has a vocabulary of size $Kn^\beta = O(n^\beta)$, where $0 < \beta < 1$ and K is a constant. Therefore, the initial local vocabulary of each processor has Kb^β words, while the final vocabulary is $O(n^\beta)$. At the first stage, the internal and communication costs are each one Kb^β, in the second stage they are each one $K(2b)^\beta$, and so on (all the merges at each stage occur in parallel). Only $\log_2 r$ stages are necessary. The final broadcast of the partition information has cost $O(r)\mathbf{C}$, which is negligible. Further, all synchronization which is required is a barrier operation between consecutive stages (i.e., levels) of the pairing process whose cost is also negligible. Therefore, the total cost to merge the vocabulary is

$$K \sum_{i=0}^{\log_2 r} (2^i b)^\beta \quad = \quad Kn^\beta \left(1 + O\left(\frac{1}{2^\beta}\right)\right) = O(n^\beta)\ (\mathbf{I} + \mathbf{C})$$

This is an average and worst case. For common texts in the English language, one can expect β to be between 0.4 and 0.6 on the average. For instance, we analyzed extracts of the Wall Street Journal issued in the years from 1987 to 1989 (obtained from disk 1 of the TREC collection [9]) and found that $K = 43.5$ and $\beta = 0.43$. We discuss this issue in greater detail in Section 5, which covers experimental results.

4.3 Step 3: Global Distributed Inverted List

Once every processor knows which part of its local inverted list must be transferred to which processor, an exchange sequence is planned. This sequence is precomputed inside each processor, and the results are all the same because they are based on the same data. Therefore, no synchronization messages are necessary.

We describe an exchange mechanism which allows every processor to be paired with each other at some moment. When two processors are paired, they exchange the appropriate portions of their indices. The exchange mechanism progresses in stages. In the first stage, we make sure that every processor is paired to its previous and next processor (assuming that processors 0 and $r-1$ are neighbors). In the second stage we do the same for every pair of processors at "distance" two, and so on. By doing so, only $\lfloor r/2 \rfloor$ stages are needed. Figure 4 illustrates this exchange mechanism for the case of seven processors. The stages are the rows in the

Figure. In the ith stage, processors at distance i are paired. At each stage, only three rounds are required to accomplish all the required pairings.

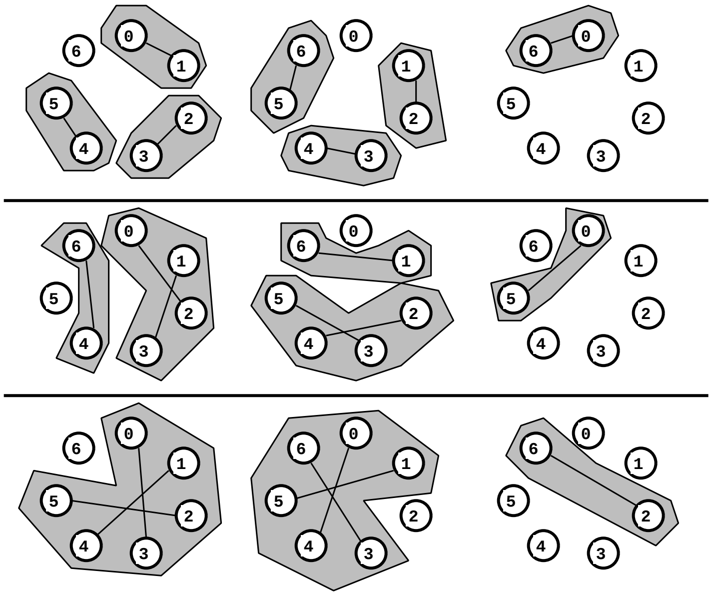

Figure 4: A redistribution process with 7 processors.

In general, the exchange mechanism works as follows. At stage i, we ensure that every processor p is paired with processors $p+i$ and $r+p-i$ (for simplicity, we refer to a processor p by either p or $jr+p$, where $j>0$). This can always be accomplished with three rounds of pairing. To show this, we distinguish groupings of pairs of processors (shown in gray in Figure 4) in each round. In the first round of stage i, the groupings of pairs are $[(0,i)$, $(1,i+1)$, ..., $(i-1,2i-1)]$, $[(2i,3i)$, $(2i+1,3i+1)$, ..., $(3i-1,4i-1)]$, and so on. In the second round of stage i, the groupings of pairs are $[(i,2i)$, $(i+1,2i+1)$, ..., $(2i-1,3i-1)]$, $[(3i,4i)$, $(3i+1,4i+1)$, ..., $(4i-1,5i-1)]$, and so on. Notice that, in general, the number of processors in a grouping is $2i$. Whenever r is a multiple of $2i$, all pairings in stage i are accomplished with only two rounds. However, if r is not a multiple of $2i$, processors might be left unpaired in the first and second rounds. In this case, the third round is required.

Let k and s be two integers such that $r=k(2i)+s$, where $k>0$, $0\leq s<2i$, and i is always a stage number. If r is not a multiple of $2i$ then $s>0$. In this case, the number of unpaired processors is s when $(s<i)$ and is $2i-1$ when $s>i$. Notice that, in the first round, the unpaired processors are near the highest-numbered processors, while in the second turn they are among the lowest-numbered. Those processors unpaired in the first two rounds need precisely to be paired among them in an additional third round. Therefore, in at most 3 rounds we complete a stage.

Hence, we need a total of $3\lfloor r/2\rfloor = O(r)$ exchange turns. In the worst case, each turn costs $O(b)\mathbf{C}$. However, as shown in the Appendix, each turn costs only $O(b/r)\mathbf{C}$ in the average (in Section 5 we verify this result for common English texts). Therefore, we have a total cost for this part of $O(n)\mathbf{C}$ in the worst case and $O(b)\mathbf{C}$ on the average.

4.4 Total Cost

In this Section, we sum up the costs of the three major steps of our algorithm to yield a total cost. From the above discussion, the worst case total cost is given by

$$O(b)\mathbf{I} + O(n)\mathbf{C}$$

While this worst case is linear on the communication costs (and thus, not superior to the linear sequential algorithm discussed in [14]), the average case is much better. In fact, on the average, the expected cost of our algorithm is

$$O(b)\mathbf{I} + O(n^{\beta} + b)\mathbf{C}$$

By observing that $b >> n^{\beta}$ for common English texts (see Section 5), we conclude that the average cost of our algorithm is expected to be

$$O(b)\mathbf{I} + O(b)\mathbf{C}$$

Such result shows that, for common English texts, our parallel algorithm is expected to scale up very nicely. In fact, for a given text of size n, increasing the degree of parallelism in our network by a factor α (i.e., increasing the number of workstations to $\alpha * r$) without violating the condition that $b >> n^{\beta}$ 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. This is a quite good and shows that, on the average, our algorithm is expected to scale up ideally.

5 Experimental Results and Estimates

In this section, we present simulation results which confirm that the amount of real text data exchanged by a pair of processors in step 3 of our algorithm (i.e., cost of a pairwise list exchange) is $O(b/r)$. We also present some statistics based on real texts which validate the assumption that $b >> n^{\beta}$. Further, we draw estimates on the algorithm expected running time for inverting a real text of size 1Gbyte.

5.1 Experimental Results

We simulated the list exchanges in step 3 of our algorithm on a single workstation. The sample text used was the Wall Street Journal (WSJ) subset contained in disk 1 of the TREC collection [9]. This WSJ subset contains extracts (or documents) of the Wall Street Journal issued in the years from 1987 to 1989.

We first merged all documents in a single large file of size nearly 250 Mbytes. This file was then broken in r (we used $r = 2$, $r = 4$, $r = 8$, and $r = 16$) text blocks. For each text block, the corresponding inverted list was generated applying a fast sequential program for list inversion [3]. This algorithm was also applied to the whole text. At this point, the following information is known: (a) the whole text, (b) the global vocabulary, and (c) the global inverted list. Further, for each text block, the following information is known: (a) the vocabulary and (b) the corresponding inverted list. Using the global vocabulary, we determine which portion of the global inverted list should be held by any given processor. Each of these portions is roughly of size b. At this point, it is known, for example, that processor p_i should receive the portion of the global index from A to C.

We proceeded by pairing each processor with all the others and by simulating the pairwise list exchanges. Let m_{ij} be the number of bytes sent from processor i to processor j when they are paired. Also, let M be the average value of m_{ij}. Then,

$$M = \frac{\sum_{i \neq j} m_{ij}}{(r-1) * r}$$

Figure 5 plots both M and b/r for r equal to 2, 4, 8, and 16. It shows experimentally that the amount of communication exchanged between any pair of processors is very close to b/r.

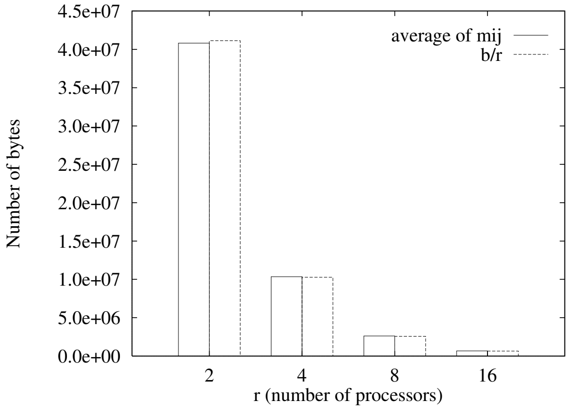

Figure 5: Comparison of M with $\frac{b}{r}$.

Intuitively, this result shows that, for a given large conventional text T, pieces of T have the same distribution of words than that of T itself (as assumed in the analysis). This is the case for the WSJ subset, but if we consider, for example, concatenated subsets (WSJ and a text composed of HTML documents), this may be not true. In this case, an unbalanced distribution occurs with some pairs of processors communicating much more than others. Figure 6 shows the amount of communication in each possible pairing for $r = 8$. It is interesting to verify that each processor j (in the horizontal axis) receives roughly the same number of bytes from each other processor. Differences among processors are due to large pointer lists for a given word which occur at the end of the available memory of a processor. A pointer list for a given word can not be spread over 2 processors. This situation causes some unbalance in the index distribution over the processors (for example, in the Figure 6, processor 8 receives a smaller part of the index).

Another important experimental result is validating the assumption that $b >> n^\beta$ (cf. Section 4). Table 1 shows n^β and b_i, $i = 2, 4, 8, 16$, for the WSJ collection and confirms that the assumption is realistic.

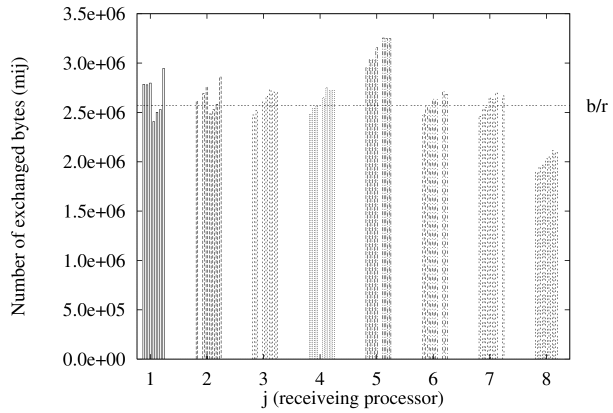

Figure 6: Amount of exchanged index parts for $r = 8$.

File	n^{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: Comparison of n^{β} and b.

5.2 Estimates on the Execution Running Time

We estimate the execution time of our algorithm for inverting a text of size nearly 1 Gigabyte obtained from the disk 1 of the TREC collection. Our estimates consider the availability of 16 machines with 64 Megabytes of free memory in each of them. The parameters of relevance are:

$n = 1.14 * 10^9$ bytes (disk1 of the TREC collection)
$b = 67.11 * 10^6$ bytes (64 Megabytes of available memory)
$r = 16$ processors (number of processors)
$w = 175.65 * 10^6$ words (number of words in the text)
$v = 4.63 * 10^6$ bytes (size of the vocabulary)
$t_p = 20.0 * 10^{-6}$ seconds/word (time per word to parse and stem terms, from[14])
$t_r = 0.5 * 10^{-6}$ seconds/byte (time to read/write from/to disk, from[14])
$t_m = t_r$ (time to access remote data in the aggregate memory, from [17])

Consider first that the whole text is stored on a single disk, accessible by all the processors. The expected running time of our algorithm is then given by

Step 1 time: $(n * t_r) + (w * t_p)$ (text read plus word filtering)
Step 2 time: $V * (2 * t_m)$ (gathering+scattering global vocabulary)
Step 3 time: $b * t_m$ (merging)

Internal times (like parallel computation of partial lists in local memory) are $O(b)$ and are completely negligible. Replacing the terms by their values, we have:

$$T = ((1.14 * 10^9) * (0.5 * 10^{-6})) + ((175.65 * 10^6) * (20 * 10^{-6})) +$$
$$((4.63 * 10^6) * (2 * 0.5 * 10^{-6})) + (67.11 * 10^6 * 0.5 * 10^{-6})$$
$$= 4121.19$$

The estimated execution time is 4121.19 seconds, approximately equivalent to 1.15 hours.

Notice that the time of the step 1 completely dominates all other times. This is the time to read the text from disks, stem the words, and assemble the vocabulary.

Consider now that we have r disks working in parallel, i.e. one disk per processor, and that the text is spread across the disks. Then, the execution time of step 1 decreases by a factor of r. With $r = 16$ (for this example), the final execution time is 4,89 minutes.

It was not considered that the final index may be written to disk. For the one disk situation this will cost 570 extra seconds (i.e., $n * t_r$). For the parallel disks case, the additional time is $\frac{570}{3} = 190$ seconds.

Let us stress network and operating system overhead. In [1], a real implementation of a distributed shared memory is discussed. Their system runs on a ATM network which operates at 100 Mbps. They report a time of 500 microseconds to send and receive a 48 bytes message. Overestimate this time by a factor of 2. Then,

$$t_m = \frac{1000 * 10^{-6}}{96} = 10,41 * 10^{-6} sec/byte \tag{2}$$

and the expected execution time of our algorithm with parallel disks becomes 17,48 minutes.

6 Conclusions

In this paper we investigated a new algorithm for the distributed parallel generation of large inverted lists in the context of a high-bandwidth network of workstations. The algorithm cost is $O(b)$ where b is the size of the local memory (i.e., the memory in any given processor) used in our computation. Using real text data, we simulated the crucial steps of our algorithm and confirmed our analytical predictions.

Our algorithm presents a couple of nice properties. First, it scales up very well. In fact, if one doubles the text size and also doubles the number of available workstations, the running time does not increase. Second, higher parallelism improves performance. In fact, with a constant text size, increasing the number of available workstations reduces the expected running time (as long as there is enough space in the local memory to store both the vocabulary and a portion of the index).

The good performance of our algorithm is due to two main reasons. First, it works solely on the distributed shared memory which is used to store the whole inverted list. Second, it takes advantage of the multiple workstations to parallelize the computation. Clearly, this implies that large memory and many workstations should be available. While this is not the case in an individual basis, the scenario is quite different with the emergence of high-bandwidth networks of modern workstations. With such networks, computing power and large memory should be highly available. Additionally, one should consider that large texts can be inverted in relatively short time and that the resultant index can be stored in disk to free memory. Thus, in many practical situations, the resources would be not tied up for long periods of time.

Interesting possibilities arise if one consider combining compression techniques, storage in local disks, and parallel algorithms. For instance, pieces of text could be inverted in parallel and the resultant partial indexes could be compressed (similarly to [14]). As in [12, 17], a parallel mergesort algorithm could be used to merge the partial indexes. If we always move compressed pointers, communication costs should be kept small even for very large indexes. More important, by maintaining the index compressed, the requirements of memory space could be considerably reduced (as in [14]). We are currently evaluating such possibilities.

References

[1] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

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

[3] M.D. Araujo, G. Navarro, and N. Ziviani. Large text searching allowing errors. Submitted for publication.

[4] R. Baeza-Yates and G. Navarro. Block addressing indices for approximate text retrieval. Submitted for publication.

[5] S. Chang, D.H. Du, J. Hsieh, R.P. Tsang, and M. Lin. Enhanced PVM communications over a high-speed lan. *IEEE Parallel & Distributed Technology*, 3(3):20–32, 1995.

[6] D.E. Culler, R.M. Karp, D. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R. Subramonian, and T.v. Eicken. Logp: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

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

[8] G.H. Gonnet, R.A. Baeza-Yates, and T. Snider. *New Indices for Text: Pat Trees and Pat Arrays*, chapter 5, pages 66–82. Prentice Hall, 1992.

[9] 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.

[10] 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.

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

[12] J.P. Kitajima, B. Ribeiro, and N. Ziviani. Network and memory analysis in distributed parallel generation of PAT arrays. In *Proceedings of the 8th Brazilian Symposium on Computer Architectures - High-Performance Processing*, pages 193–202, Recife, PE, August 1996. Brazilian Computer Society - SBC.

[13] 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.

[14] 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.

[15] E. Moura, G. Navarro, and N. Ziviani. Indexing compressed text. Technical report, Dept. of Computer Science, Univ. Federal de Minas Gerais, Brazil, 1996.

[16] G. Navarro, J.P. Kitajima, B. Ribeiro-Neto, and N. Ziviani. Distributed generation of suffix arrays. Submitted for publication.

[17] B. Ribeiro, J. P. Kitajima, and N. Ziviani. Distributed parallel generation of pat arrays. Technical Report 019/96, Universidade Federal de Minas Gerais - Departamento de Ciência da Computação, Belo Horizonte, Brazil, June 1996.

Appendix: Analysis of Pairwise List Exchange

In this Section, we show that the maximum amount of data exchanged by a pair of processors in the step 3 of our algorithm is $O(b/r)$.

Since the global index is divided so that an equivalent portion is assigned to each processor, we have r equal-sized portions in the index. The part of the local index to transfer to each processor can be taken as a random sampling over the whole set of words. Therefore, the number of elements of the local index of processor i corresponding to processor j has a Binomial distribution with parameters $B(b, 1/r)$, since it comes from randomly taking b elements of the global index and observing how many of them correspond to processor j (which happens with probability $1/r$).

The amount to exchange between every pair of processors can be seen as r independent random variables with the same Binomial distribution. The maximum amount of data exchanged in a stage corresponds therefore to the maximum of r independent random variables with distribution $B(b, 1/r)$. Let $X_1, ..., X_r$ be those random variables.

We first show that, for $j > b/r$, $P(X_i \geq j) = O(P(X_i = j))$, i.e. the first term of the summation of probabilities dominates the rest once we passed the mean of the distribution. If we call $p_j = P(X_i = j)$, we have

$$P(X_i \geq j) \quad = \quad \sum_{k=j}^{b} p_k \quad = \quad \sum_{k=j}^{b} \binom{b}{k} \frac{(1-1/r)^{b-k}}{r^k}$$

and we observe that

$$\frac{p_{k+1}}{p_k} \quad = \quad \frac{b-k}{(k+1)r(1-1/r)} \quad \leq \quad \frac{b}{(k+1)r} \quad \leq \quad C \quad < \quad 1$$

where the inequalities come from the fact that $k \geq b/r$. $C = b/(b+r)$ is a new constant introduced to indicate that there is a fixed upper bound for all p_{k+1}/p_k which is independent of k and smaller than 1. Therefore, the terms of the summation decrease at least by a multiplicative constant, what makes their sum a constant proportion of the first summand, i.e. Dp_j, where the constant D is bounded above by $D = 1/(1-C)$.

We now consider the probability of $Y = \max(X_1, ..., X_r) \geq k$. This is equivalent to some X_i being $\geq k$. Bounding again, we have

$$P(Y \geq k) \quad \leq \quad P(X_1 \geq k) + ... + P(X_r \geq k) \quad = \quad rP(X \geq k) \quad \leq \quad Drp_k$$

We find out now how must k be in order for the above probability to be $\leq 1/r$ (we use that result later). That is

$$Drp_k \quad = \quad Dr\binom{b}{k}\frac{1}{r^k} \quad \leq \quad 1/r$$

where we pessimistically discarded the factor $(1-1/r)^{b-k}$. Taking logarithms we have

$$b\log b - k\log k - (b-k)\log(b-k) + O(\log b) \quad \leq \quad k\log r$$

(an additional term $O(\log r)$ is discarded because $r << b$).

We substitute now $k = \alpha b/r$, for constant α, in the above equation and simplify, to obtain

$$-\frac{\alpha}{r}\log\alpha - \log(1-\alpha/r) + \frac{\alpha}{r}\log(1-\alpha/r) + O\left(\frac{\log b}{b}\right) \quad \leq \quad 0$$

which by expanding logarithms yields

$$\log\alpha + \frac{\alpha}{r} \quad \geq \quad 1 + O\left(\frac{\log b}{b} + \frac{1}{r^2}\right)$$

which is clearly achieved by some constant α.

Therefore, we have proved that for $k = O(b/r)$, the probability of the maximum Y among all the random variables X_i being $\geq k$ is $\leq 1/r$. We use it to bound the mean of Y, which is the value we are seeking for:

$$E(Y) \quad = \quad \sum_{j=0}^{b} jP(Y=j) \leq k + (b-k)P(Y > k) \quad \leq \quad k + b/r \quad = \quad O(b/r)$$

what completes the proof.