Efficient Distributed Algorithms to Build Inverted Files

Abstract

We present three distinct distributed disk-based algorithms to build global compressed inverted files for very large text collections. The distributed environment we use is 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. Our algorithms consider that the total distributed main memory is considerable smaller than the inverted file to be generated. Compression is used to save memory and disk space and to save time for moving data in/out disk and across the network. We analyze our algorithms and discuss the tradeoffs among them. For practical purposes the average running time of the algorithms is $O(c)$, where c is the size of the local document collection at each machine. Using 8 processors and 16 megabytes of RAM available in each processor, the advanced variants of our algorithms are able to invert a 100 gigabytes collection (the size of the very large TREC-7 collection) in roughly 12 hours. Using 16 processors this time drops to roughly 6 hours.

1 Introduction

The potential large size of a full text collection demands specialized indexing techniques for efficient retrieval. Some examples of such indexes are suffix arrays, inverted files, and signature files. Each of them have their own strong and weak points. However, *inverted files* have been traditionally the most popular indexing technique used along the years. Inverted files are useful because their searching strategy is based mostly on the vocabulary which usually fits in main memory. Further, 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, a common type of query in IR systems and in the Web.

Despite their simple structure, the task of building inverted files for very large text collections such as the whole Web is costly. Therefore, faster indexing algorithms are always desirable and the use of parallel or distributed hardware for generating the index is an obvious alternative. In this regard, as hardware platform we consider a network of workstations because it presents performance comparable to that of a typical parallel machine but is more cost effective [1, 4].

In [11], a distributed memory-based algorithm for generation of compressed inverted files is discussed. This algorithm works only for text collections that fit in the distributed main memory. In here, we focus our study on workstations whose main memory is considerably smaller than the inverted file to be generated. The reason is that with the actual rate of growth of modern digital libraries (particularly the Web), inversion models heavily dependent on relatively large memories will eventually become unusable. Further, we consider that the inverted files to be generated are compressed. This allows fast index generation with low space consumption [7, 13] which is important in a distributed environment where data has to be moved across the network.

In a distributed environment, a clear option for generating the index is to partition the document collection among the various machines and to generate a local inverted file at each machine. However, for purpose of query processing, an organization based on local inverted files is consistently outperformed by a distributed global inverted file partitioned among the various machines [10]. In a global inverted file organization, incoming queries have to be submitted to the machines which include lists associated with the query terms only. As a result, higher concurrency is achieved which allows accelerating the query processing task by a factor which can go up to 10 for short Web-like queries [3, 10]. Thus, generation of global inverted files for distributed collections is a problem of practical interest.

In this paper we discuss new distributed algorithms for building global inverted files for large collections distributed across a high-bandwidth network of workstations. We adopt as building block a well known sequential disk-based algorithm for generating compressed inverted files described in [13]. That algorithm is modified to generate frequency-sorted inverted lists because these allow faster ranking [8, 9].

We discuss how to transform the sequential algorithm in [13] into a distributed one. We show that three distinct variations are possible which we call LL, LR, and RR. Through experimentation and analysis, we discuss the performance of these three algorithms and the tradeoffs among them. The LL algorithm is based on the simple idea of computing first the local inverted files at each machine (using a sequential algorithm) and merging them in a subsequent phase. The LR and RR algorithms are more sophisticated and also superior. In fact, they present an efficiency (the ratio between speedup and the number of processors) close to 1, whenever the network bandwidth is comparable to the disk bandwidth — a tradeoff which is in accordance with modern switching and disk technologies. Further, we argue that the LR algorithm is preferrable in most practical situations which are likely to occur nowadays because, in those situations, it presents performance comparable to that of the RR algorithm but is simpler to implement.

This paper is organized as follows. In Section 2 we introduce the environment of our distributed text collection and the terminology we use throughout the paper. In Section 3 we discuss three design decisions for distributed inverted files: numbering of index terms according to a lexicographical ordering, frequency-based sorting of inverted lists, and compressing the inverted lists. In Section 4 we detail a sequential disk-based algorithm for generating inverted files which we use as a building block for our distributed algorithms. In Section 5 we present and analyze our three distributed algorithms named LL, LR, and RR. In Section 6 we present experimental and analytical results which show the practical performance of the proposed algorithms. Finally, in Section 7 we present our conclusions.

2 Distributed Text Collection and Terminology

We consider a high-bandwidth network of workstations with a shared-nothing memory organization as illustrated in Figure 1. In this network, we assume that the documents of our collection are distributed among the various machines and stored on their local disks. For simplicity, throughout this work we assume that each machine holds nearly the same number of documents. However, all our algorithms are equally suitable for a non-uniform distribution of the document collection.

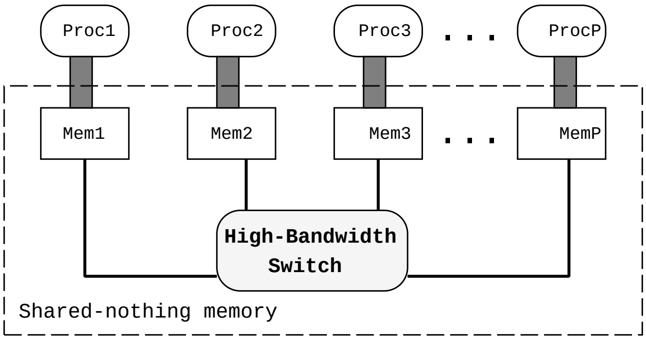

Figure 1: Network of workstations with shared-nothing memory.

A global inverted file is generated for the whole collection. Since such index is quite large (despite being compressed), it also has to be distributed among the machines in the network. As done for the document collection, we assume a uniform distribution of the index such that any two processors hold a portion of the inverted file which is roughly of the same size in bytes.

Various variables affect the performance of the distributed algorithms for index generation here discussed. To refer to them, we use the terminology below. All sizes are in bytes while all times are measured in seconds.

- B: local memory buffer for storing inverted lists
- b: size of local buffer B
- c: size of local collection at each processor
- d_j: the jth document
- f: size of local compressed inverted file
- f': size of local temporary compressed inverted file
- $f_{i,j}$: frequency of occurrence of k_i in d_j
- k_i: the ith index term
- R: number of runs for generating an inverted file
- p: number of processors
- t_1: time spent at phase 1 of an algorithm
- t_a: time spent at step a of an algorithm
- t_{ci}: time to compare two integers
- t_{cs}: average time (per byte) to compare two strings
- t_n: average network communication time (per byte)
- t_p: average time (per byte) for parsing index terms
- t'_p: average time (per byte) for parsing index terms a second time
- t_r: average disk transfer time (per byte)
- t_s: average seek time
- t_z: average time (per byte of compressed file) to compress (or decompress)
- tt_X: total time spent by an algorithm named X
- v: size of local vocabulary (distinct set of index terms)
- x: number of indexing points (non-distinct) in local collection
- w: total number of words (non-distinct) in local collection
- w_s: average size (in characters) of an English word

The parameters c, f, f', v, x, and w are relative to the local collection at each processor. To obtain their values, we multiply the respective size by the number p of processors (except for the vocabulary size v whose value is a non-linear function of the collection size as detailed in Section 5.1). For instance, $c * p$ is the size (in bytes) of the whole document collection.

The time t_z to compress the index depends on the size of the source uncompressed file and on the size of the compressed file. Here, as in [13], we approximate t_z by a function of the size of the resultant compressed file only. Such approximation is reasonably accurate for our purposes.

3 Design Decisions for Distributed Inverted Files

In this section, we briefly review a distributed memory-based inversion algorithm which illustrates some of the tradeoffs involved in the generation of distributed inverted files. These tradeoffs motivate the adoption of three design decisions which affect our algorithms for distributed inverted files: numbering of index terms according to a lexicographical ordering, frequency-based sorting of inverted lists, and compressing the inverted lists.

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 in the vocabulary, an *inverted list* of the documents in which that word occurs (the frequency of the word in each of those documents is also normally included). Such indexing structure usually occupies space which is roughly 40% of the collection size. For small to medium size collections, it is then possible to generate the whole distributed index in the aggregate memory of the various machines in the network as done in [11].

The distributed memory-based algorithm proposed in [11] is composed of three phases. In phase 1, each processor builds an inverted file for its local text. In phase 2, the global vocabulary and the portion of the global inverted file to be held by each processor are determined. In phase 3, portions of the local inverted files (as determined in phase 2) are exchanged to generate the global inverted file. Clearly, this algorithm can be modified to operate as a disk-based algorithm by storing the local inverted files generated in phase 1 on disk. But then, for successful execution of phase 3, the now disk-resident inverted file must be traversed sequentially. This requires (a) storing the inverted file on disk in alphabetical order of its index terms and (b) assigning disk-contiguous slices of the global inverted file to each processor (for instance, index terms initiated with characters from a to c are assigned to processor 0). This leads to our first design decision for distributed inverted files.

> Decision 1: The index terms which compose our inverted files (both local and global) will be (a) ordered lexicographically and (b) numbered in increasing order according to the lexicographical order previously established.

The inverted lists generated in [11] follow the ordering of documents naturally existent within the collection (which determines the ordering for parsing the documents). In [8, 9], however, it is proposed to sort the inverted lists in the inverted file by the frequency of occurrence of terms in documents. The motivation is that, in case a vectorial ranking is adopted (which is the preferable choice nowadays), much more efficient ranking can be obtained at query processing time. The idea is to filter the inverted list such that only a portion of it, composed of the documents with higher $f_{i,j}$ frequencies, needs to be retrieved from disk. Reducing the sizes of the inverted lists retrieved from disk is particularly important in a distributed environment because it reduces considerably the sizes of the matching lists which have to be moved across the network at query processing time. This leads to our second basic decision which is as follows.

Decision 2: The pairs $[d_j, f_{i,j}]$ which compose the inverted list for each index term k_i must be sorted according to the decreasing values of the frequencies $f_{i,j}$.

In [11], compression is used in the inverted lists, both during construction and querying times. The motivation is that, if compression is used, inverted lists are smaller and can be moved faster accross the network. Furthermore, this reduction in size improves also the time spent reading and writing the lists to disk. This leads to our third design decision for distributed inverted files.

Decision 3: The inverted list for each index term k_i will be compressed.

4 A Sequential Disk-Based Algorithm

Disk-based sequential algorithms (i.e., which run in a single processor) for generating compressed inverted files have been studied extensively in the literature [7, 13]. However, the algorithms usually fail to consider the first two design decisions we just established in the previous section. To enforce them, we adapt the *multiway merging* algorithm proposed in [13].

The sequential algorithm we propose is composed of three phases a, b, and c. In phase a, all documents are read sequentially from disk and parsed into index terms. This allows creating a perfect hashed vocabulary [5] in which the terms are sorted lexicographically and numbered according to our decision 1. In phase b, all documents are again read sequentially from disk and again parsed into index terms. Triplets composed of an index term number k_i, a document number d_j, and a frequency $f_{i,j}$ are then formed and inserted into a buffer B in main memory. Whenever this buffer fills (i.e., whenever a run is completed), the partial inverted lists are sorted in decreasing order of the frequencies $f_{i,j}$ (to fulfill our decision 2), compressed (to fulfill our decision 3), and stored in a temporary file F. In phase c, a disk-based multiway merge is done to combine the partial inverted lists into final lists. The details are shown in Figure 2.

```
a. (a1) read all documents sequentially, parse into index terms and
        create perfect hashed vocabulary
   (a2) create memory buffer B   /* with dynamic allocation */
   (a3) R = 0   /* initialize number of runs */
b. foreach d_j do
   begin
   (b1) read d_j, parse into index terms
   (b2) foreach  k_i ∈ d_j  do  B = B + [k_i, d_j, f_i,j]
   (b3) if  'buffer B is full'  then
        begin
        (b3.1) do  quicksort on the triplets   /* keys = k_i, f_i,j */
        (b3.2) F = compress-write-disk(B)
        (b3.3) B = φ;   R = R + 1
        end
   end
c. multiway-disk-merge(F,R)
```

Figure 2: The sequential (single processor) algorithm.

Apparently, reading and parsing all the documents twice is a bad choice. However, this is not so. In fact, the expensive portion of this step is the parsing. Since the tasks executed at the first parsing (building the hashed vocabulary) and the tasks executed at the second parsing (building the triplets and inserting them into the buffer B) are complimentary and have to be executed anyway (i.e., even if only a single reading and parsing is done), the double reading and parsing does not add a great overhead. Further, it allows building a perfect hashed vocabulary which provides for direct access to any inverted list with no need to lookup at a vocabulary entry [5]. Thus, once the perfect hash has been built, it is no longer necessary to keep the vocabulary in memory (all significative memory consumption is now represented by the buffer B which stores the inverted lists).

Analytical Costs

Our analytical model is based on the discussion in [13]. The sequential algorithm detailed in Figure 2 is composed of the steps a, b, and c. Let t_a, t_b, and t_c be the respective times (in seconds) for each of these steps.

For the step a, we can write

$$t_a = ct_r + wt_p + (1.2\ v \log v)\ w_s t_{cs}$$

which accounts for reading all the documents in the collection, parsing all the words, and sorting the vocabulary to generate a perfect hash. The constant 1.2 is the proportionality constant for a well engineered implementation of the quicksort.

For the step b, we can write

$$\begin{aligned} t_b &= ct_r + wt_p^{'} + & /*\ b1 + b2\ */ \\ &\ R\left(1.2\ \frac{x}{R}\log\frac{x}{R}\right) t_{ci} + & /*\ b3.1\ */ \\ &\ f^{'}\ (t_z + t_r) & /*\ b3.2\ */ \\ &= ct_r + wt_p^{'} + 1.2x\log\frac{x}{R}t_{ci} + f^{'}\ (t_z + t_r) & \qquad (1) \end{aligned}$$

which accounts for a second reading of all the documents in the collection and a second parsing of all words. Each time a run is completed (i.e., each time the buffer area B fills up), the roughly x/R triplets $[k_i, d_j, f_{i,j}]$ in the buffer B must be sorted using the term number k_i as primary key and the frequency $f_{i,j}$ as secondary key. This makes the triplets for a given term k_i contiguous and sorts its corresponding inverted list (which is partial at this point) in decreasing values of $f_{i,j}$. These lists are then compressed and stored on disk. The compression scheme we use at this phase is the one described in [9].

For the step c, we can write

$$t_c = f^{'}\ \left(\frac{R}{b}t_s + t_r + t_z\right) +\ x\ \lceil \log R \rceil\ t_{ci} +\ f\ (t_r + t_z)$$

which accounts for (1) reading and uncompressing the partial inverted lists generated by the R runs (from each run, a portion of size b/R is read into memory at a time), (2) R-way merging them using a heap structure (which requires $x \log R$ comparisons), and (3) compressing the merged lists and writing them back to disk.

The total time tt_{seq} for our sequential algorithm is then given by

$$tt_{seq} = t_a + t_b + t_c \qquad (2)$$

In section 6 we demonstrate that such analytical model is quite accurate and yields predictions which are very close to the total time measured in a real execution.

5 Distributed Disk-Based Algorithms

In this section, we discuss three distributed disk-based algorithms for generating global inverted files.

5.1 LL Algorithm: Local Buffer and Local Lists

The main idea behind our first parallel algorithm is to combine the parallel memory-based algorithm discussed in Section 3 with the sequential disk-based algorithm discussed in Section 4. In this new algorithm, the buffer for storing the $[k_i, d_j, f_{i,j}]$ triplets is *local* to each machine. Also, the inverted lists are initially stored on the *local* disk at each machine. Thus, we label this parallel inversion algorithm as the LL algorithm (Local buffer and Local lists)

The LL algorithm works as follows.

- Phase 1: *Local Inverted Files.* In this phase, each processor builds an inverted file for its local text and stores it on the local disk.
- 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. For this, the processors are paired as illustrated in Figure 3. At the end, processor p_0 results with the global vocabulary.

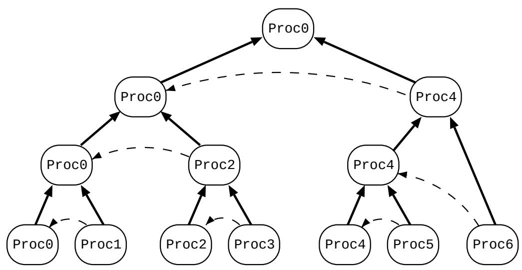

Figure 3: Global vocabulary computation with 7 processors.

- 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. For this, an all-to-all communication procedure [12] is used. The number of rounds of processor pairings required is $p-1$. Each portion received by a processor p_i (from every other processor p_j) is stored on disk as a separate run. As a result, each processor ends up with p separate runs on disk. At the end, a p-way merging on disk is executed at each processor to generate the final inverted lists which compose the global inverted file. The details are as follows.

pair each processor p_i with every other processor p_j
foreach p_i within $[p_i, p_j]$ do
begin
    (3.1) from the local disk of p_i, read the inverted lists for p_j

(3.2) send the inverted lists read to p_j
(3.3) receive remote inverted lists from p_j
(3.4) store inverted lists just received on a disk file F
end
(3.5) multiway-disk-merge(F,p)

Analytical Costs

The LL algorithm is composed of the phases 1, 2, and 3. Let t_1, t_2, and t_3 be the time spent in each of these phases, respectively. The analytical costs (in time) are as follows.

The phase 1 is exactly the sequential algorithm discussed in Section 4. Thus, the time t_1 is equal to the time tt_{seq} detailed in Eq. 2 i.e.,

$$t_1 = tt_{seq}$$

The phase 2 takes $\log_2 p$ steps to be completed (see Figure 3). Each step lasts for the time to send the accumulated vocabulary from the higher numbered processor to the lower numbered one (in each pair). This accumulated vocabulary is roughly of the same size for any pair of processors (and doubles in size each time we move up in the computation tree of Figure 3). At the end, the vocabulary is broadcast to all other processors (by processor p_0).

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

$$v = Kc^\beta$$

where $0 < \beta < 1$ and K is a constant [6, 2]. Thus, the time t_2 spent at phase 2 can be approximated by

$$\begin{aligned} t_2 &= \left(\sum_{i=0}^{(\log_2 p)-1} 2^i \ v \right) w_s \ (t_n + t_{cs}) + K(pc)^\beta \ (w_s + 4) \ t_n \\ &= (p-1) \ v \ w_s \ (t_n + t_{cs}) + v(p)^\beta \ (w_s + 4) \ t_n \end{aligned}$$

where w_s is the average size in bytes of English words and can be taken roughly as 5. Further, to illustrate, the values of K and β for the disk 1 of the TREC collection are roughly $K = 4.8$ and $\beta = 0.56$ [2].

The second factor in the expression for t_2 accounts for the broadcast of the global vocabulary to all the other processors. The collection size considered in this case is that of the global collection and is given by $p * c$. Besides the terms of the vocabulary, an integer of size 4 is also transmitted. This integer is the new number associated to each term according to a lexicographical ordering of the global vocabulary.

The time t_3 for phase 3 is as follows.

$$\begin{aligned} t_3 = \ & (p-1) \ 2 \ \frac{f}{p} \ t_r + & /*3.1+3.4*/ \\ & (p-1) \ 2 \ \frac{f}{p} \ t_n + & /*3.2+3.3*/ \\ & f \ \left(\frac{p}{b} \ t_s + t_r + t_z \right) + \ x \ \lceil \log p \rceil \ t_{ci} + \ f \ (t_r + t_z) & /*3.5*/ \end{aligned}$$

5.2 LR Algorithm: Local Buffer and Remote Lists

In the LL algorithm, when the memory buffer B fills the partial inverted lists are stored on a local disk. As a result, those inverted lists have to merged locally through an R-way merging procedure (at the end of phase 1). Later on, at the end of phase 3, a second multiway merging procedure is executed (this time a p-way merging) to merge the lists received from other processors. The first one of these two multiway merging procedures can be avoided entirely if we store the partial inverted lists at a remote disk (instead of at a local disk). This is the main idea behind the LR algorithm which we now discuss.

The LR parallel algorithm uses a *local* memory buffer B as before but sends the partial inverted lists (at the time the buffer B fills) to other processors for storage at *remote* disks. This means that information on the global vocabulary must be available early on (because a processor p_i needs to know the destination of each inverted list to be sent out). As a result, the LR algorithm is in Figure 4.

```
a. (a1)  each processor reads its local documents sequentially;
         each processor creates a perfect hashed vocabulary
   (a2)  B = ∅;    R = 0 /* initialize B and R */
b. (b1)  compute global vocabulary in p_0 which renumbers all index terms
   (b2)  p_0 broadcasts global vocabulary to all processors
c. foreach d_j do    /* all processors in parallel */
   begin (c1) read d_j, parse into index terms
         (c2) foreach k_i ∈ d_j do B = B + [k_i, d_j, f_{i,j}]
         (c3) if 'buffer B is full' then
              begin
              (c3.1) do quicksort on the triplets   /* keys = k_i, f_{i,j} */
              (c3.2) pair each processor p_i with every other processor p_j
                     foreach p_i in a pair [p_i, p_j] do
                     begin let B_i[j] be portion of B in p_i destined to p_j.
                           (c3.2.1)  CB_i[j] = compress(B_i[j])
                           (c3.2.2)  send-to(p_j, CB_i[j])
                           (c3.2.3)  receive-from(p_j, CB_j[i])
                           (c3.2.4)  F = F + write-disk(CB_j[i])
                     end
              end
   end
d. multiway-disk-merge(F,p*R)
```

Figure 4: The LR algorithm.

Analytical Costs

The total execution time tt_{LR} of the LR algorithm can be estimated as follows.

$$\begin{aligned} tt_{LR} \;=\;& ct_r + wt_p + (1.2\; v \log v)\; w_s t_{cs} + && /*a*/ \\ & \Big[(p-1)\; v\; w_s\; (t_n + t_{cs}) + v(p)^{\beta}\; (w_s+4)\; t_n\Big] + && /*b*/ \\ & ct_r + wt_p^{'} + 1.2x \log \frac{x}{R} t_{ci} + && /*c1+c2+c3.1*/ \end{aligned}$$

$$f^{'}\ (t_z+t_r)+\left[2\ \frac{p-1}{p}\ f^{'}\ t_n\right]+ \qquad /*c3.2*/$$
$$f^{'}\ \left(\frac{pR}{b}\ t_s+t_r+t_z\right)\ +\ x\ \lceil\log pR\rceil\ t_{ci}+\ f\ (t_r+t_z) \qquad /*d*/ \qquad (3)$$

We observe that the expression for tt_{LR} is similar to the expression for tt_{seq} in Eq. 2. The main differences are (1) the terms between square brackets in the second and fourth lines of Eq. 3 (corresponding to the vocabulary computation and the exchange of inverted lists) and (2) the pR-way merging (instead of an R-way merging) in the last line of Eq. 2. Since the vocabulary computation is very fast [11] and the difference in time between an R-way and a pR-way merging is usually small, the difference in time between the sequential and the LR algorithm (which works with a text p times larger) is fundamentally the term $[2*f^{'}*t_n*(p-1)/p]$. Thus, with a fast enough network, the LR algorithm presents an efficiency (ratio between speedup and number of processors) which is close to 1.

5.3 RR Algorithm: Remote Buffer and Remote Lists

The LR algorithm stores the triplets $[k_i, d_j, f_{i,j}]$ in a local buffer B, assembles partial inverted lists in this buffer, and then sends these lists out. A potential improvement is to collect the triplets in messages early on and send these messages out very soon to avoid storage at the local buffer B. By doing so, the buffer B is assembled remotely. A distinct message for every other remote processor p_j is assembled separately. These messages should be large enought to avoid penalties due to excessive network overhead. Through proper programming of two threads (one for disk access and other for network access), it is possible to overlap the transmission through the network with the sequential reading of local documents from disk. This implies that transmition of the inverted lists through the network becomes almost free.

In the RR algorithm, the buffer B is *remote* and the partial inverted lists are stored on *remote* disks. The algorithm is a slight variation of the LR algorithm and is not detailed here.

Analytical Costs

The total execution time tt_{RR} of the RR algorithm can be estimated as follows.

$$\begin{aligned} tt_{RR} \;=\; & ct_r+wt_p+(1.2\ v\log v)\ w_s t_{cs}+ \\ & \left[(p-1)\ v\ w_s\ (t_n+t_{cs})+v(p)^{\beta}\ (w_s+4)\ t_n\right]+ \\ & ct_r+wt_p^{'}+1.2x\log\frac{x}{R}t_{ci}+f^{'}\ (t_z+t_r)+ \\ & f^{'}\ \left(\frac{R}{b}\ t_s+t_r+t_z\right)+x\ \lceil\log R\rceil\ t_{ci}+f\ (t_r+t_z) \end{aligned}$$

which, except for the vocabulary computation accounted for in the second line above, is the expression for tt_{seq}. Since the vocabulary computation accounts for less than 2% of the total computation time, we can say that the RR algorithm is as fast as the sequential algorithm (while inverting a global collection which is p times larger)!

The main disadvantage of the RR algorithm is that it requires a far more complex implementation. Further, since the triplets are sent out very soon, good compression of the inverted lists at this point is not possible which implies that network traffic is higher.

6 Results

In this section we discuss experimental and analytical results for our distributed disk-based algorithms. Unless explicitly told otherwise, the parameters we consider in our experiments and analysis are as follows.

b	48 Mbytes	t_p	$0.769\mu s$/byte
c	8 Gbytes	$t_p^{'}$	$2.82\mu s$/byte
f	$0.1 * c$	t_r	$12.2\mu s$/byte
$f^{'}$	$1.1 * f$	t_s	11ms
l	x/v	t_z	$4.88\mu s$/byte
p	8	v	$4.8c^{0.56}$
t_{ci}	88ns	x	$w/2$
t_{cs}	61ns	w	$c/(w_s + 1)$
t_n	10μs	w_s	5

Thus, we assume a local collection size (per processor) of 8 Gbytes and a local buffer area of 48 Mbytes (per processor). The disk transfer time t_r corresponds to a transfer rate of 8.2 Mbytes per second which is typical nowadays. The network communication time t_n corresponds to a net bandwidth of 80 Mbits per second as observed for an ATM switch operating at a raw bandwidth of 155 Mbits per second. The standard number p of processors in our analysis is 8 which, considering PC hardware, can be assembled at relatively low cost.

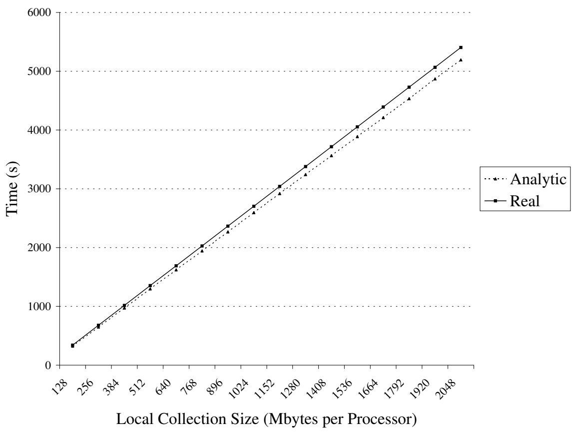

Figure 5: Analytical and experimental total execution time for our sequential disk-based algorithm considering varying collection sizes.

We implemented the sequential algorithm described in Section 4 and ran it for distinct collection sizes with the purpose of validating our analysis of that algorithm. Figure 5 illustrates the results for collection sizes ranging from 128 Mbytes to 2 Gbytes. We observe that the analytical model provides a good prediction for the overall execution time. The small difference in the graph is due to operating system overhead which is not accounted for. We also compared the experimental and predicted time for each of the factors in the expression for

tt_{seq} (Eq. (2)) and again observed a pretty good match. Therefore, Eq. (2) allows analyzing the behavior of the sequential algorithm in generic situations, as we now do.

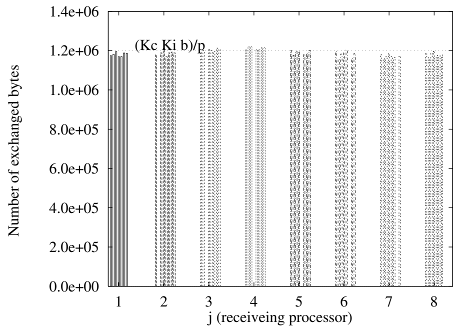

Figure 6: Network traffic among 8 processors considering a local collection of size 96 Mbytes.

Regarding the LL, LR, and RR distributed algorithms, we notice that their total execution times include factors related to disk reading/writing, parsing, sorting, comparing, compressing, multiway merging, and network shipping. With the exception of network shipping (i.e., moving bytes across the network), all these factors are also present in the sequential algorithm and have been confirmed experimentally.

With regard to network shipping, the factor t_n is as measured. Further, we observe that we have assumed uniformity of traffic among the various pairs of processors. To validate this assumption, we computed compressed inverted files for distinct local collections (of size 96 Mbytes each), computed the respective global vocabulary, partitioned lexicographically the compressed inverted lists, and shipped each portion to the proper remote processor. Figure 6 illustrates the results. As can be observed, with large inverted files, an even partition of the global inverted file among the various processors translates into an even distribution of the network traffic, which validates our modeling assumption.

Figure 7 compares the total execution times for the sequential and the distributed RR algorithms considering collection sizes which vary from 8 Gbytes to 128 Gbytes. In this case the RR algorithm is always inverting a collection 8 times larger (because $p = 8$) than the sequential algorithm. The two total execution times are basically the same. The reasons for this extremely good efficiency are the facts that transmission across the network is overlapped with disk access operations and that computation of the global vocabulary can be done very fast.

In Figure 8 we compare the execution times of our three distributed algorithms for collection sizes which vary from 8 Gbytes to 128 Gbytes. We first observe that the LR and RR algorithms are roughly 40% faster than the LL algorithm. Thus, investing time in a more sophisticated distributed algorithm is worthwhile. With 12.5 Gbytes of text data per processor and 8 processors (for a total collection size of 100 Gbytes), the LR algorithm takes roughly 12 hours to generate a global distributed inverted file. Using 16 processors and 6.25 Gbytes of text data per processor, this time drops to roughly 6 hours. We also notice that at this network speed, the LR algorithm presents behavior very close to the RR algorithm. Since the LR algorithm is simpler to implement than the RR algorithm, it should be the algorithm of choice in most practical situations.

In Figure 9 we compare the execution times of our three distributed algorithms for a constant local collection size ($c = 8$ Gbytes in each processor) and various number of processors.

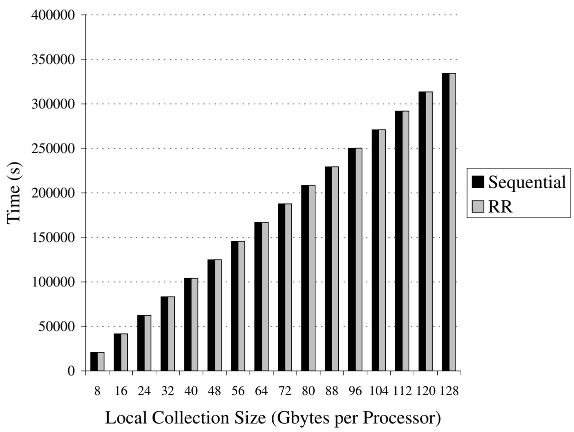

Figure 7: Execution times for the sequential and the RR algorithms considering various collection sizes.

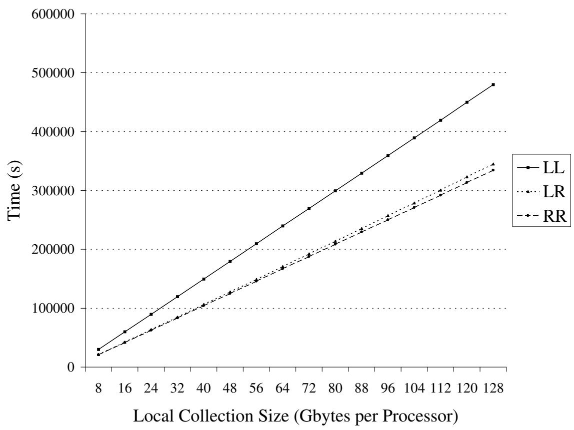

Figure 8: Execution times of the LL, LR, and RR algorithms for various sizes of the collection in each processor.

We observe that at this network speed, the cost of each of our algorithms remains basically constant as the number of processors increase. This means that larger and larger collections can be inverted at almost no additional cost in time if (a) a larger number of processors is available and (b) the processors can communicate in parallel without contention.

In Figure 10 we compare the execution times of our distributed algorithms for various sizes b of the local memory buffer at each processor. The number of processors is 8 and the local collection size is 8 Gbytes. We first observe that a relatively small memory buffer (with regard to the size of the local collection) might be disastrous for the LR algorithm. The reason is that it executes a pR-way merge which becomes costly if the number R of runs increases too much. This effect is not present in the LL algorithm because it executes an R-way merge followed by a p-way merge (the RR algorithm executes only an R-way merge). Fortunately, this effect disapears with a relatively modest increase in the memory buffer size. For instance, for $b = 16$ Mbytes the effect is no longer noticeable. Thus, the LR algorithm might be the preferable choice as long as the ratio between the memory buffer b and the local collection c

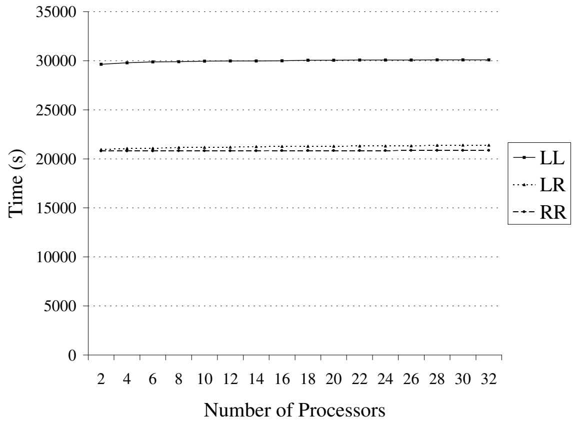

Figure 9: Execution times of our distributed algorithms for various numbers of processors for a collection $c = 8$ Gbytes in each processor.

is kept large enough (in the case of Figure 10, larger than 16Mbytes/8Gbytes).

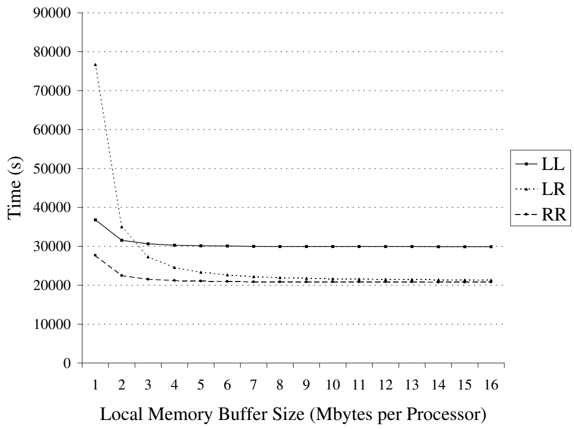

Figure 10: Total execution times for various sizes of the local memory buffer ($p = 8$, $c = 8$ Gbytes).

In Figure 11 we evaluate the impact of varying the available network bandwidth. We notice that a network bandwidth much smaller than the disk transfer rate leads to poor performance by the LL and LR algorithms. However, this effect disapears quickly as the network bandwidth increases. For instance, for a ratio t_n/t_r close to 1 (equivalent to a network bandwidth of 80Mbps in Figure 11), the effect is no longer noticeable.

¿From our analysis, we conclude that the LR algorithm is currently the preferrable choice because it is simpler to implement and presents performance close to that of the RR algorithm in most practical situations.

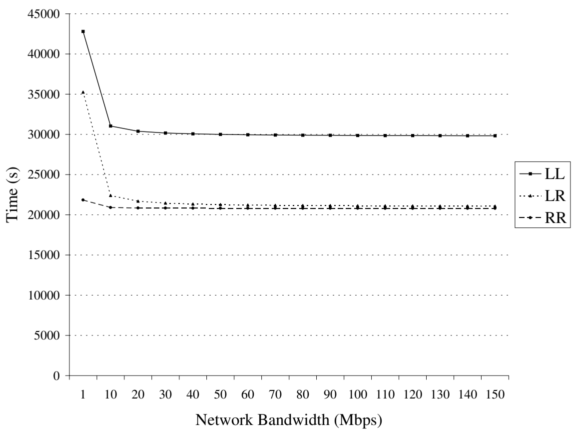

Figure 11: Execution times of our distributed algorithms for various values of network bandwidth.

7 Conclusions

In this paper we investigated three new and scalable algorithms for the distributed disk-based generation of very large inverted files in the context of a high bandwidth network of workstations. We focused our study on workstations whose main memory is considerable smaller than the inverted file to be generated. Further, we considered that the inverted files to generate are compressed. This allows fast index generation with low space consumption, which is important in a distributed environment where data has to be moved across the network.

We modified a well known sequential disk-based algorithm for generating compressed inverted files described in [13] to generate frequency-sorted inverted lists because these allow faster ranking [8, 9]. We then discussed how to modify this sequential algorithm to transform it into a distributed one. We showed that three distinct variations are possible which we call LL, LR, and RR. Through experimentation and analysis, we discussed the performance of these three algorithms and the tradeoffs among them. We were able to show that the LR and RR algorithms are superior algorithms which present an efficiency close to 1, whenever the network bandwidth is comparable to the disk bandwidth. Further, we argued that the LR algorithm is preferrable in most practical situations because, in those situations, it presents performance comparable to that of the RR algorithm but is simpler to implement.

Our experiments confirm that our algorithms are efficient to invert very large text collections and that, for practical purposes, their costs vary linearly with the size of the local text in each workstation. For instance, with 8 processors and 16 Mbytes of memory available in each processor (in a network at 80 Mbps), the LR and RR algorithms are able to invert a 100 gigabytes collection (the size of the very large TREC-7 collection) in roughly 12 hours.

In the near future we intend to experiment with our algorithms using collections whose sizes go up to 0.5 terabytes.

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. Barbosa. Desempenho de consultas em bibliotecas digitais distribuidas. Master's thesis, Computer Science Department, Federal University of Minas Gerais, Brazil, 1998. Master thesis. In Portuguese.

[4] 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.

[5] Z.J. Czech, G. Havas, and B.S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing and Letters*, 43:257–264, 1992.

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

[7] 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.

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

[9] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

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

[11] B. Ribeiro-Neto, J.P. Kitajima, G. Navarro, C. Santana, and N. Ziviani. Parallel generation of inverted files for distributed text collections. In *Proceedings of the XVIII International Conference of the Chilean Society of Computer Science (SCCC'98)*, pages 149–157, Antofagasta, Chile, 1998.

[12] 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.

[13] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Van Nostrand Reinhold, New York, 1994.