Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

# Distributed Parallel Generation of Indices for Very Large Text Databases

João Paulo Kitajima
Maria Dalva Resende
Berthier Ribeiro-Neto
Nivio Ziviani

Relatório Técnico 008/97[1]

April 1997

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

# Distributed Parallel Generation of Indices for Very Large Text Databases

João Paulo Kitajima[†]
Maria Dalva Resende
Berthier Ribeiro-Neto[‡]
Nivio Ziviani[§]

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
Vox +55 31 499 5860/Fax +55 31 499 5858
{kita,dalva,berthier,nivio}@dcc.ufmg.br

**Abstract**

We propose a new algorithm for the parallel generation of suffix arrays for large text databases on high-bandwidth multicomputers. Suffix arrays are structures used in full text indexing which support very powerful query languages. Our algorithm is based on a parallel indirect mergesort (it is not a *simple* mergesort procedure) and is compared with a well known sequential algorithm (which is very efficient running on a single machine). We analyze execution times for both algorithms. Considering parallel systems with low communication latencies, we show that our algorithm is clearly preferable to the sequential one. We present a summary of several experiments performed with the implementation of the parallel algorithm and its sequential counterpart. Although network-bounded, the parallel version is theoretically and experimentally a much better alternative when compared to the sequential version (which is I/O-bounded in disk).

[†]This author has been partially supported by CNPQ Project 300815/94-8.
[‡]This author has been partially supported by CNPQ Project 300188/95-1.
[§]This author has been partially supported by CNPQ Project 520916/94-8 and Project RITOS/CYTED.

# 1 Introduction

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 information retrieval. One such a 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[1]. 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 [3]. 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 large size of a full text collection (consider, for instance, that the texts of all documents in the collection are merged in a single very large text file as done with the TIPSTER collection [4]) and the potentially large size of the string to be searched demand specialized index techniques for efficient retrieval. A fast index for retrieval in full text is the Patricia tree [10]. Indeed, when long search strings are involved, the Patricia tree is the method of choice [6]. PAT arrays [2, 3] or suffix arrays [7] are simpler structures which provide an efficient implementation of Patricia trees while supporting a powerful query language. Another efficient index technique is that based on inverted lists [6]. Although largely employed, queries using inverted lists are less flexible than those using suffix arrays.

The purpose of this paper is to analyze a mergesort-based algorithm for the generation of large suffix arrays in the context of a high-bandwidth network of RISC processors. A network of such processors 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 communication overhead in parallel applications. Second, a high-bandwidth network of processors provides computing power comparable to that of a typical supercomputer but is more cost effective [1]. Nowadays, remote access memory using the network takes approximately the same in time than local disk access. Therefore, we can use the aggregate memory in the high-bandwidth network as a giant cache for disks. More precisely, our idea is to use the aggregate memory in the network as storage for the whole text and its associated suffix array.

In order to validate our parallel algorithm, we compare it with a clever sequential algorithm presented in [3]. This algorithm uses disk extensively and attempts to always access disk sequentially (minimizing the number of seek operations). It has an $O(\frac{T^2}{M})$ complexity, where $T$ is the text size and $M$ is the size of the available primary memory. Our distributed parallel mergesort algorithm has an $O(T)$ time complexity. For practical purposes, we demonstrate that our algorithm provides an improvement in execution time which might go up to one order of magnitude and beyond when commercially available multicomputers are considered. Such a performance

[1]The 2 gigabytes TIPSTER collection [4] includes thousands of documents which are articles extracted from editions of the Wall Street Journal, AP Newswire, abstracts from DOE publications, and US Patents among others. Another example is the World-Wide Web (WWW) with its homepages.

figure is not easily attainable though and requires careful design.

# 2 Basic Definitions

In full text retrieval, the entire text is viewed as one very long string. In this string, each position $k$ is associated to a semi-infinite string which initiates at $k$ and extends to the right as far as needed or to the end of the text. Such semi-infinite strings are called *sistrings* [3]. The user specifies his information need by referring to the sistrings he is interested in. The task of the information system is to search the full text for the occurrences of the user specified sistrings. To perform this search efficiently our information system uses a suffix array (generated for that text) as the indexing structure. Such indices support very complex queries such those based on long sequence of words, some types of boolean queries, regular expression search, longest repetitions and most frequent search [3].

A *suffix array* (or PAT array) is a linear structure composed of pointers to every sistring in the text. These pointers are sorted according to a *lexicographical ordering* of their respective sistrings. Further, each of these pointers can be viewed simply as the offset (counted from the beginning of the text) of the sistring in the text.

We consider that the user is interested in text words and/or text phrases only such that the suffix array needs to include solely pointers to those sistrings which occur at the beginning of words in the text. Since English words are roughly 6 characters long, the number of such sistrings is about one sixth of common text written (the average word size of the TIPSTER collection varies from 5.7 to 6.5, depending on the document). Let $T$ be the size of the text. Assuming that the pointers in the suffix array are also 6 bytes long (which allow to address sistrings in texts of size up to 64 terabytes), the size of the suffix array is approximately $T$ (i.e., $6 \times 0.1667 \times T$). Thus, the suffix array and the text are roughly the same size.

Searching for a sistring might require a potentially large number of character comparisons because of the sistring semi-infinite property. However, the vast majority of expected search requests do not require comparing very large sistrings. For instance, searching the Oxford English Dictionary for user specified sistrings can be resolved in 97% of the cases with comparisons which involve no more than 48 characters [3].

Considering the definitions above, the generation of suffix arrays consists in sorting an array of text pointers. Although philosophically simple, the fact that the sort is indirect (pointers are lexicographically sorted according to the pointed text sistrings) makes a difference. Every reference to an element of the suffix array aimed at solving a comparison implies in an access to the text. Things become more complex if we consider that text and suffix array occupy too much space to be stored in primary memory.

# 3 Suffix Array Generation

## 3.1 The Sequential Algorithm

The sequential algorithm considers that a single machine is used to build the suffix array. This machine must have enough disk space for storing the whole text and its suffix array. If all the text and its suffix array fit in main memory, the problem becomes one of sorting the suffix array (according to a lexicographical ordering of its corresponding sistrings) in primary memory and can be solved trivially using, for instance, quicksort. For large texts, however, the text must reside on disk and one cannot avoid accessing the secondary memory for retrieving text sistrings. Since accesses to secondary memory are dominated by the seek time, one must minimize the number of seek operations when accessing a large file.

The sequential algorithm is based on the following procedure [6]:

**Algorithm 1** <u>*General procedure for external sorting.*</u>
*(1) break the large file into smaller blocks which fit in main memory;*
*(2) sort the individual blocks in main memory;*
*(3) merge the sorted blocks together until the whole file is sorted.*

The key problem is the implementation of step 3. Let $b_P$ be the portion of the text (composed of one or more blocks of text) corresponding to a partial suffix array $P$. Also, let $p_i$ be the suffix array for the block of text $b_{p_i}$. To merge the suffix arrays $P$ and $p_i$, it is necessary to access the sistrings in $b_P$ and $b_{p_i}$. Since these accesses are non-sequential (they follow the suffix array pointers), the text blocks $b_P$ and $b_{p_i}$ must be in main memory (otherwise, too many seek operations would be required). Unfortunately, as $P$ grows $b_P$ becomes too large to fit in main memory. If we attempt to access the sistrings in $b_P$ directly from disk, the execution time becomes unacceptable due to the large number of seek operations.

To avoid seek operations, the algorithm always accesses the text sequentially (i.e., subsequent disk accesses refer to contiguous disk sectors). This is accomplished by maintaining auxiliary counters which indicate the number of sistrings, pointed by elements of a suffix array, which fall between any two consecutive sistrings pointed by elements of another suffix array [3]. These counters allow merging the two suffix arrays through strictly sequential accesses (with no need to further inspect text sistrings). Most important, these counters can be generated by always accessing text sistrings sequentially. Thus, the whole process can be completed through strictly sequential disk accesses. Using counters forces sequential access to disk, but merging small suffix arrays into large ones makes time complexity square in the text size. More precisely, the time complexity of the algorithm is $O(\frac{T^2}{M})$, where $M$ is the memory size.

## 3.2 The Distributed Parallel Mergesort Algorithm

The mergesort based parallel algorithm we propose works as follows. The text is partitioned in $N_b$ blocks of size $s_{blk}$. Each text block $b_i$ is assigned to a distinct processor such that the corresponding suffix array $p_i$ can be generated locally (using, for instance, quicksort). After the generation of these suffix arrays, the $ith$ machine contains the block of text $b_i$ and its suffix array $p_i$ in its main memory. Our idea is to merge these suffix arrays by moving sistrings solely in the aggregate memory (i.e., without ever storing them on disk). Since each suffix array is already sorted, we adopt the *mergesort* algorithm to perform this distributed merging.

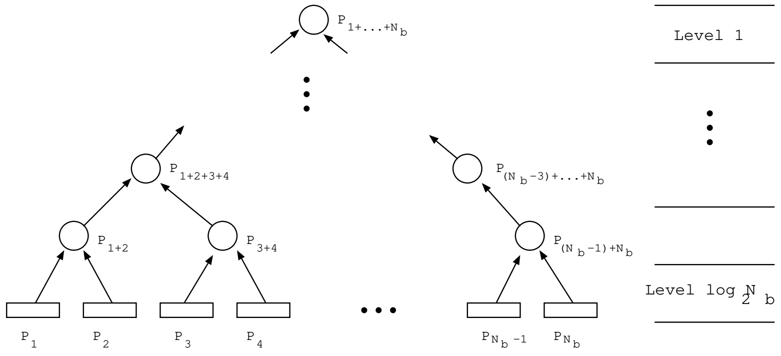


Figure 1: Execution strategy for the distributed parallel mergesort algorithm.

Figure 1 illustrates the merging procedure. At the bottom level (i.e., level $\log_2 N_b$), we group the machines in pairs. The lowest numbered machine in each pair controls the merging operation. The resultant suffix array is stored in the aggregate memory of this pair of machines. The total number of merges at the bottom level is $\frac{N_b}{2}$ and they can all be done in parallel. We assume that the suffix array pointers are words of 48 bits which already include absolute text offsets such that they do not need to be adjusted when they are moved from one machine to another. The algorithm for merging a pair of neighbor suffix arrays $p_{2t-1}$ and $p_{2t}$ is as follows.

**Algorithm 2** *Merging of the suffix arrays $p_{2t-1}$ and $p_{2t}$ by machine $2t-1$*

$i = 2t - 1$; $j = 2t$; $s = 0$; $sj = 0$;
*for* ($si = 0$; $si \leq n_i$; $si + +$) {
*while (sistring(*$p_j[sj]$*) <= sistring(*$p_i[si]$*))* $p_{i+j}[s + +]$ = $p_j[sj + +]$;
$p_{i+j}[s + +]$ = $p_i[si]$;
};

We assume the presence of proper sentinels at the end of the suffix arrays $p_i$ and $p_j$ which make the test for array limits unnecessary. Further, the code above does not make it explicit that the resultant suffix array $p_{i+j}$ is stored in the aggregate memory formed by the $ith$ and $jth$ machines. At the immediately above level (i.e., level $(\log N_b) - 1$), we group two pairs of machines into quadruples. At the next above level, eight machines are grouped together, and so on. At the root node, the final merge is done and the suffix array $p_{1+2+\ldots+N_b}$ generated.

## 3.3 Parallel Execution Time

The execution time of the parallel algorithm is composed of the following two times: (a) time $t_{suffix}$ to compute the suffix arrays $p_1, \ldots, p_{N_b}$ in parallel and (b) time $t_{merge}$ to merge the suffix arrays in the aggregate memory. The first of these times is:

$$t_{suffix} = t_{dsk} + 2 * t_{mem} + t_{qck}$$

where $t_{dsk}$ is the time to access one text block from disk, $t_{mem}$ is the time to access one text block from primary memory and $t_{qck}$ is the time to perform quicksort on main memory. The time $t_{merge}$ to merge suffix arrays depends heavily on the number of message packets exchanged which means that care must be exercised.

We first observe that as the computation moves up in the execution tree (Figure 1), a given suffix array pointer and its respective sistring might end up residing in the memories of two distinct machines (because pointers are moved in the aggregate memory but sistrings are not).

We divide the available memory in 4 portions and reserve one of them for text. Thus, the number $N_b$ of text blocks is given by $\frac{4T}{M}$, where $M$ is the memory size. The 4 portions of memory are used to hold: (a) a block-size portion of the resultant suffix array which is currently been computed, (b) the suffix array which is participating in the current merging operation, (c) a block of text $b_i$, and (d) its respective suffix array $p_i$. The suffix array $p_i$ is kept in memory throughout the computation to allow accessing the sistrings in $b_i$ in lexicographical order. Remote suffix array pointers and sistrings are now retrieved according to the following protocol (considering in this analysis that pointers are 6 bytes long):

(1) send out a packet requesting $\frac{s_{pkt}}{6}$ adjacent pointers
(2) receive a packet with the $\frac{s_{pkt}}{6}$ pointers requested
(3) for each of the received pointers do:
(3a) send out a packet (to the proper machine) requesting $\frac{s_{pkt}}{6}$ sistrings of size 6 sorted lexicographically
(3b) receive a packet with the requested sistrings

A minimum extra buffer space (of size $s_{pkt} * N_b$) is required to hold the newly arrived data (i.e., a buffer of size $s_{pkt}$ bytes is reserved locally for each remote machine in the network). As a result of the above protocol, the next $\frac{s_{pkt}}{6}$ remote pointers (and their respective sistrings) are promptly available for the merging operation. Furthermore, since the sistrings in each machine buffer are sorted lexicographically, they can be processed sequentially as the merging operation moves on.

A problem arises when the first 6 characters of a sistring are not enough to decide a comparison. In this case, an extra request message is dispatched to the proper machine requesting the whole sistring (which fits in a single packet). This event happens with a probability $q$. In Section 4, we show that $q$ is dependent on the input text.

In [11], it is shown that the time $t_{parallel}$ to conclude the whole suffix array generation task can be computed as

$$t_{parallel} = 6 * N_b * \frac{s_{blk}}{6} * \frac{6 * t_{pkt}}{s_{pkt}} + O(\log N_b)$$

where $s_{pkt}$ is the size of message packet and $t_{pkt}$ is the user-to-user transfer time of one such packet. Since $T = N_b * s_{blk}$, the complexity of the algorithm is $O(T)$.

A preliminary comparison of this parallel algorithm with the sequential one is as follows. Consider first the situation in which the number of machines needed for computing the suffix array for the whole text is small ($\sim$ 10). In this trivial case (in which the size of the available memory is of the same order of magnitude of the text size), both the sequential and parallel algorithms present execution times which are comparable and the sequential algorithm is preferable because its implementation is simpler.

When the text size far exceeds the size of the memory available (the most common situation for large texts), the factor $T$ becomes dominant and the parallel version becomes the algorithm of choice. We also point out that the parallel algorithm does not have the disk space requirements of the sequential algorithm (as a matter of fact, disks are not used in the parallel version).

### 3.4 Scalability Issues

We remark that the sequential algorithm is an $O(\frac{T^2}{M})$ algorithm and the parallel version has an $O(T)$ complexity, where $T$ is the text size and $M$ is the memory available per machine. Three scenarios are of interest. First, consider doubling the number of processors and keeping $T$ and $M$ fixed. The execution time is not changed for the parallel version. The extra processors and respective memories do not help in improving the performance of the algorithm. For the sequential algorithm, nothing changes too (text and memory sizes are not changed). Second, consider doubling the number of processors and doubling $T$, but keeping $M$ fixed. This implies that the execution time doubles. For the sequential algorithm, on the other hand, the execution time becomes 4 times larger. Third, consider doubling the text size $T$ and memory size $M$. This implies that the execution time for both algorithms doubles.

Despite the fact that the parallel algorithm scales poorly in the number of processors and in the size of the problem, to each instance of the sequential algorithm, there is a corresponding parallel instance which executes faster. The only scenario favoring the sequential algorithm is when just $M$ doubles. The sequential algorithm executes twice faster, but the execution time of the parallel version does not change. So, the parallel algorithm is more suitable when there are a lot of processors with small memory size (which is the typical situation). The larger the memory size (with respect to the text size), the more suitable the sequential algorithm becomes.

## 4 Experimental Analysis

Theory and practice should complement each other. We know that although an analytical model is useful for system insight and understanding of the main tradeoffs, there are several implementation details which are not taken into account in the modeling phase. We implemented the parallel algorithm in order to understand the different parameters involved. We analyzed its behavior considering the input (i.e., characteristics of the documents databases) and the underlying parallel system. A comparison with the sequential program is also presented.

## 4.1 The Sequential and Parallel Programs

The sequential program is written in C++ and is based on the algorithm proposed in [3]. The parallel program is also in ANSI C. For the experiments, we used pointers and integers of 4 bytes. The employed parallel programming paradigm is that based on message exchanging. The PVM (Parallel Virtual Machine) library [12] is used. As discussed in Section 3, the parallel algorithm is that based on a master-slave strategy. The master process is responsible for reading the text from disk, dividing it in blocks and sending those to the slave processes. Between each two consecutive levels of the mergesort algorithm (there are $\log_2 p$ levels, where $p$ is the number of processors - see Figure 1), a synchronization with the master process is performed. At the end of the program execution, the suffix array is either left distributed through the network or coalesced and stored on disk by the master process. The program is rather flexible, there are several possible optimization presented below in this section. To measure execution and communication overhead times, both sequential and parallel programs use a very precise `getrealtime` function (microsecond precision).

## 4.2 The Parallel System

Although PVM is a portable library implemented on a large number of systems, we have used an enhanced version of PVM adapted to the IBM SP (Scalable POWER) parallel system [5]. The SP can be used in different ways: as a parallel machine, as a computing server, or as a distributed mainframe. Compared to typical vector supercomputers or massively parallel processors, those machines provide more flexibility and a better price-performance ratio. Each processor is a RISC IBM RS/6000 and the communication network is a proprietary omega multistage switch (the HPS - High-Performance Switch) with a unidirectional bandwidth of 40 megabytes per second and latencies around 40 microseconds. The HPS can be used by PVM using two protocols: (1) the IP protocol and (2) a proprietary protocol, part of the User Space Communication Subsystem (US CSS). With the IP protocol, standard PVM is used. However, to obtain the maximum performance of the communication network, the second protocol must be used. For this protocol, an enhanced version of PVM is available (PVMe). The library interface is very similar than that of the standard PVM. However, execution is restricted to one PVMe process per processing node, no matter how many users are connected to that processing node. Each node runs AIX (IBM Unix).

In particular, for our experiments, we have used the IBM SP of the Inria-IMAG APACHE Project at Grenoble, France. This system is composed of 32 processing nodes at 66 MHz and 64 megabytes of primary memory. Each processor has 3 gigabytes of local disk, being very suitable for our applications which work with large databases. Besides primary memory, each processor has a 64 kilobytes data cache and a 8 kilobytes instruction cache. The employed version of the AIX is 3.2.1. All the experiments were performed considering no extra load from other users. The only additional load was that incurred by the operating system.

## 4.3 The TIPSTER Collection

For the experiments done in this work, we used documents extracted from the 2 gigabytes TIPSTER collection. In particular, we worked with the following files: (1) AP: AP Newswire (1989), (2) FR: Federal Register (1989), (3) WSJ: Wall Street Journal (1987, 1988, 1989), and (4) ZIFF: articles from the *Computer Selected* magazine (Ziff-Davis Publishing). We consider a word as a sequence of characters in the sets A..Z, a..z, and 0..9. Capital and small letters are considered equivalent.

## 4.4 Influence of Filtering and of Stop Words

The execution time of a text indexing program is considerably affected by operations of text pre-processing, such as filtering and elimination of stop words. In the immediately following, we evaluate the impact of these two operations on the execution time of our parallel indexing program.

**Filtering**. When reading a document for indexing, text filtering is usually performed in order, for example, to ease the search of keys with more than one word. For example, when filtering, the sistrings:

```
Parallel Processing
Parallel                 Processing
```

are identical. If the user searches for `Parallel Processing` (with one blank space between `Parallel` and `Processing`), the two above sistrings (inside their respective contexts) will return. However, (1) the cost for such filtering is expensive and (2) it is done whenever a sistring is manipulated: filtered text should always be mapped onto the non-filtered text. Thanks to the Unix `gprof` utility, estimates on filtering could be done. `gprof` generates a profile of procedure calls and execution times. With the above utility, we noticed that a function in the parallel program called `_ReadFilteredString` spends 15.8% of the total execution time. Another problem with filtering is the reduction of the alphabet. Remembering Subsection 3.3, when one processor asks a remote sistring for comparison when sorting, only some characters of this remote sistring are used in the comparison. If these characters are not enough, more characters of the sistring is demanded to the remote processor owner of the sistring. When filtering, the alphabet used for sistrings is reduced (special characters are not considered). With a smaller alphabet, the probability of mismatches when using just the initial characters of a sistring is lower (that is, a lot of identical pieces of sistrings will occur). This implies asking more characters of sistrings to solve the comparison. Figure 3 presents the execution times when filtering and no filtering are performed.

Unfiltered text yields better execution times, but we should not forget that query processing will be more complex when indices based on unfiltered texts are used. This is because the query system should itself perform the filtering.

**Stop words**. Another possible optimization is to avoid the indexing of what are called *stop words*. These words are in general pronouns, adverbs, and prepositions which function as elements of style or syntax, but without important meaning. We can cite, for instance, *the*, *that*, and

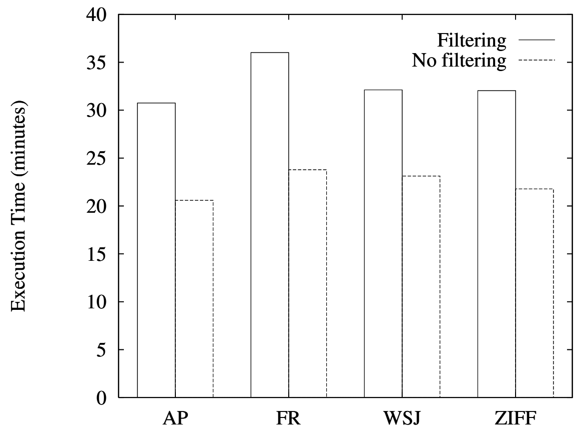


Figure 2: Influence of text filtering on program performance.

*of*. For example, in AP, FR, WSJ and ZIFF files (considering pieces of 16 megabytes for each file) from the TIPSTER collection, stop words correspond to 34 to 41% of the total number of words (using the list of stop words provided by [9]). When stop words are not indexed, the index itself is smaller and consequently less messages are exchanged between pairs of processors. Figure 3 presents the execution times when indexing and not indexing sistrings starting with stop words, considering pieces of 16 megabytes for each file of the collection. In the following experiments, filter and elimination of stop words are performed.

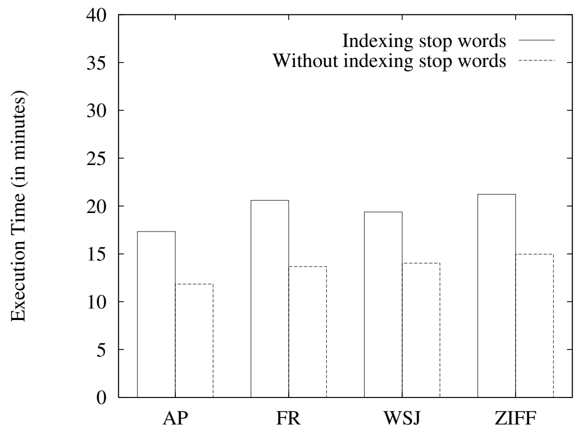


Figure 3: Influence of stop words indexing on program performance.

## 4.5 Varying Sistring and Message Sizes

In parallel programming, communication between tasks should be minimized. Indexing is an I/O-bound application which becomes network-bound when done in parallel. Therefore, in order to minimize communication overhead, there are two parameters we can modify: (1) the sistring size and (2) the message size.

**Sistring size**. The sistring size tradeoff is: the larger the remote sistring demanded, the less is the probability of asking more characters to solve a comparison. There is another tradeoff: the smaller the sistring, more sistrings can be received inside a message. For example, a message with 48 bytes holds 8 sistrings of 6 characters. 6 characters can not be enough to solve a comparison: more characters have to be brought from the remote processor. In other words, the larger the sistring, we risk to bring useless characters, because comparison is solved with just a part of the sistring.

To better understand the sistring comparison behavior, we sorted all sistrings of different files from TIPSTER (all of them with 16 megabytes). In this experiment, we generated sequentially the suffix array for each file. We computed for each suffix the number $L$ of identical characters shared with the immediately previous suffix in the sorted suffix array. For example:

```
suffix x  : "A document is a piece of paper..."
suffix x+1: "A document preparation system..."   L=11
suffix x+2: "A dollar in my pocket..."           L=4
```

We then built the probability distribution function for $L$ (Figure 4). We immediately notice

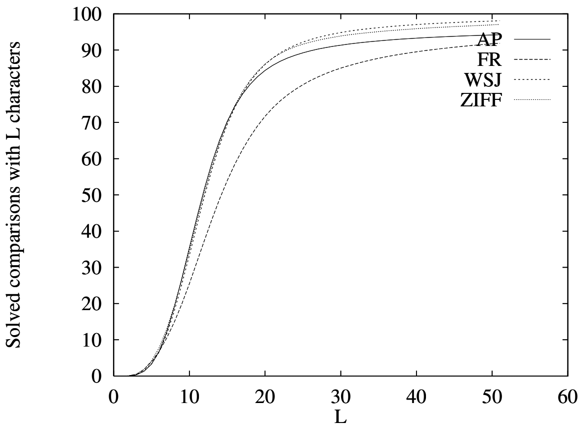


Figure 4: Probability distribution for $L$.

that the assumption in Section 3 that 6 characters is enough to decide a comparison between two arbitrary sistrings does not hold. The analytical model presented in Section 3 is too optimist, even considering that Figure 4 is too pessimist (the numbers were obtained comparing adjacent *sorted* sistrings). Figure 5 presents an example of execution times of the parallel algorithm considering different sizes of sistrings for 4 megabytes of the WSJ file. Beyond 24 bytes, increasing the sistring size is not interesting because more messages will be necessary to carry sistrings (more network traffic).

**Message size**. In the experiments above, the message size (which contains either pointers to sistrings or sistrings) was fixed in 48 bytes. However this parameter can also be changed with

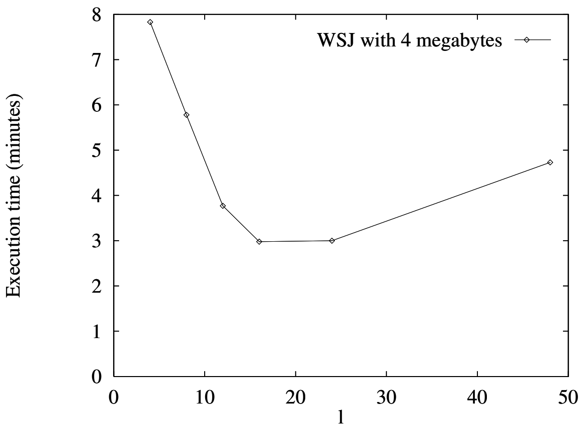


Figure 5: Execution time of the parallel algorithm for different values of $l$ (the number of sistring characters transmitted at first).

a tradeoff similar to that for the sistring size (the larger the message, the more information is brought, the more primary memory space is used). However, in this case, there is also a system motivation. In the IBM SP architecture, it is more efficient to send few larger messages than several smaller ones [8]. Although smaller messages present better latency, due to an eager communication protocol, message start-up is considered heavy. On the other hand, larger messages suffer from worst latency (mainly in loaded networks) due to a synchronous protocol - sender only sends message when receiver is able to receive. However, given that the path between sender and receiver is open, the message flows without waiting delays.

Larger messages imply in lower execution times until virtual memory begins to be used. Figure 6 shows execution time in function of message size for different WSJ text sizes on 16 processors. Figure 7 presents the same information but with larger WSJ pieces (64 megabytes on 4 processors and 260 megabytes on 16 processors).

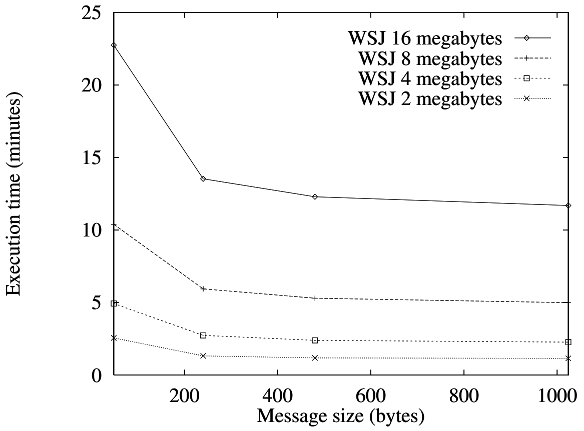


Figure 6: Execution time in function of message size for different text sizes on 16 processors.

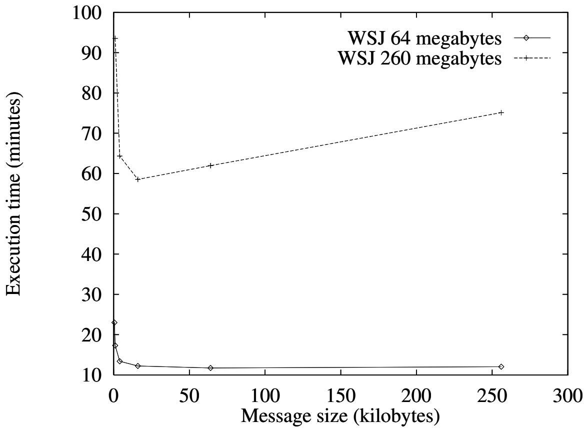


Figure 7: Execution time for larger texts.

### 4.6 Parallel versus Sequential: Experimental Results

In this section, we compare the parallel and sequential algorithms for building suffix arrays. First, we executed the parallel program with different sizes of the WSJ file and different number of processors. The linear dependence of the execution time on the text size, predicted in the analytical model, is verified. We used small texts to guarantee the use of the whole primary memory (for low number of processors) and to avoid the use of virtual memory. The measured execution times are presented in Figure 8. We also verify poor scalability in the sense that the more processors we use, the slower the algorithm executes for a given text size. Parallel indexing is extremely sensible to I/O and to memory sizes. We can trade disk access by network access (as we have done), but we can not (still) trade primary memory access with network access. If few processors have more primary memory than a cluster composed by tenths of processors (with low memory size), it will be better to execute on the smaller configuration.

Figure 9 presents the execution times for WSJ with different text sizes . Parallel execution time is compared with the sequential program execution time. For large text files, the quadratic cost of the sequential algorithm makes our parallel algorithm clearly the one of choice. Considering a larger IBM SP system, our algorithm should be able to invert a text of 3 gigabytes under 10 hours.

## 5 Conclusions

We proposed a new algorithm for the parallel generation of large suffix arrays in high-bandwidth multicomputers. Further, we also discussed a well known sequential algorithm which is very efficient running on a single machine. We carefully analyzed the complexity of the execution time for both algorithms. Considering a commercially available parallel system, we showed empirically that our algorithm is clearly preferable to the sequential one and that the improvements

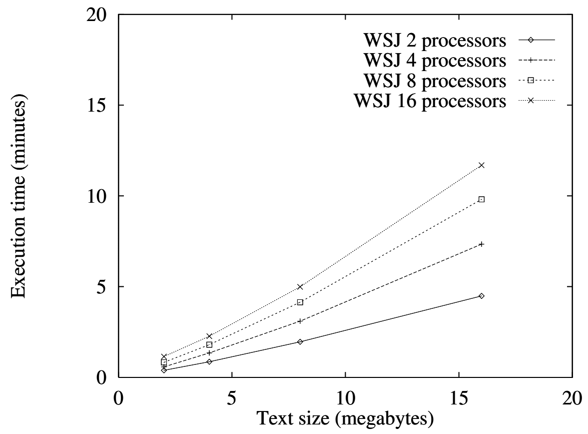


Figure 8: Evidence of linearity in execution time (message size of 1 kilobyte and sistring size of 16 bytes).

in execution time it provides are expected to go up to one order of magnitude and beyond. Such results confirm our analytical predictions.

We also discussed several experiments which evaluated the impact of varying distinct parameters of the problem. These experiments led us to the following conclusions. Executions are faster when text is not filtered and when stop words are not indexed. Moving larger sistrings is also better than moving small sistrings because many comparisons can often not be decided in this last case. Larger messages are also preferred due to the interconnection network latency. Finally, although very network-bounded, the parallel version is theoretically and experimentally a better alternative when compared to the sequential version (I/O-bounded in disk).

Work is currently being done using three basic techniques to enrich our basic parallel algorithm: (1) compression, which should minimize memory and network requirements, (2) use of local disk, which should alleviate memory usage, and (3) overlap of computation, communication, and disk access.

# Acknowledgments

We would like to thank the Laboratoire de Modélisation et Calcul (LMC), a research unit of the Institut d'Informatique et de Mathématiques Appliquées de Grenoble (IMAG), and the *APACHE* Project, both in France, for making available their IBM SP. We also wish to acknowledge Kemio de Oliveira Couto who helped particularly with the implementation of the sequential algorithm.

# References

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

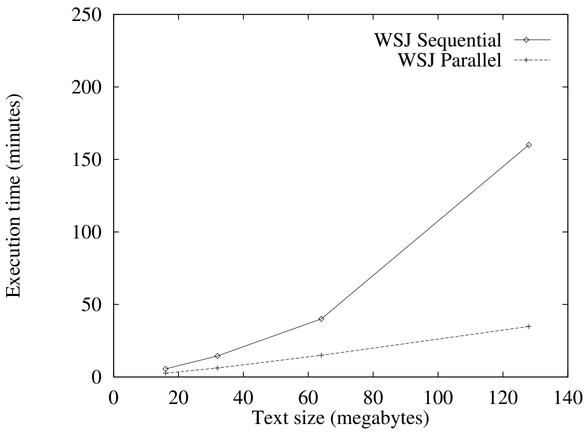


Figure 9: Parallel versus sequential execution times (message size of 4 kilobytes and sistring size of 48 bytes).


[2] G. Gonnet. *PAT 3.1: An Efficient Text Searching System – User's Manual*. Centre of the New Oxford English Dictionary, University of Waterloo, Canada, 1987.

[3] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval – Data Structures & Algorithms*, pages 66–82. Prentice-Hall, 1992.

[4] Donna 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.

[5] IBM. IBM SP WWW home page, 1997. http://www.rs6000.ibm.com/hardware/largescale /index.html.

[6] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison Wesley, 1973.

[7] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *First ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, San Francisco, 1990.

[8] J. Miguel, A. Arruabarrena, R. Beivide, and J. A. Gregorio. Assessing the performance of the new IBM SP2 communication subsystem. *IEEE Parallel & Distributed Technology*, 4(4):12–22, Winter 1996.

[9] G. A. Miller, E. B. Newman, and E. A. Friedman. Length-frequency statistics for written English. *Information and Control*, 1:370–380, 1958.

[10] D.R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric. *JACM*, 15(4):514–534, October 1968.

[11] 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.

[12] 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.