Distributed Generation of Suffix Arrays: a Quicksort-Based Approach

João Paulo Kitajima[1][3] Gonzalo Navarro[2][4]
Berthier A. Ribeiro-Neto[1][5] Nivio Ziviani[1][6]

[1] Dept. of Computer Science, Federal University of Minas Gerais, Brazil.
[2] Dept. of Computer Science, University of Chile, Chile.

[3] This author has been partially supported by CNPQ Project 300815/94-8.
[4] This author has been partially supported by Fondef grant 96-1064 (Chile).
[5] This author has been partially supported by CNPQ Project 300188/95-1.
[6] This author has been partially supported by CNPQ Project 520916/94-8 and Project RITOS/CYTED.

Abstract. An algorithm for the distributed computation of suffix arrays for large texts is presented. The parallelism model is that of a set of sequential tasks which execute in parallel and exchange messages between each other. The underlying architecture is that of a high-bandwidth network of processors. In such a network, a remote memory access has a transfer time similar to the transfer time of magnetic disks (with no seek cost) which allows to use the aggregate memory distributed over the various processors as a giant cache for disks. Our algorithm takes advantage of this architectural feature to implement a quicksort-based distributed sorting procedure for building the suffix array. We show that such algorithm has computation complexity given by $O(r \log(n/r) + n/r \ \log r \log n)$ in the worst case and $O(n/r \ \log n)$ on average and communication complexity given by $O(n/r \ \log^2 r)$ in the worst case and $O(n/r \ \log r)$ on average, where n is the text size and r is the number of processors. This is considerably faster than the best known sequential algorithm for building suffix arrays which has time complexity given by $O(n^2/m)$ where m is the size of the main memory. In the worst case this algorithm is the best among the parallel algorithms we are aware of. Furthermore, our algorithm scales up nicer in the worst case than the others.

1 Introduction

We present a new algorithm for distributed parallel generation of large suffix arrays in the context of a high bandwidth network of processors. The motivation is three-fold. First, the high cost of the best known sequential algorithm for suffix array generation leads naturally to the exploration of parallel algorithms for solving the problem. Second, the use of a set of processors (for example, connected by a fast switch like ATM) as a parallel machine is an attractive alternative nowadays [1]. Third, the final index can be left distributed to reduce the query time overhead. The distributed algorithm we propose is based on a parallel quicksort [7, 13]. We show that, among previous work, our algorithm is the fastest and the one that scales best, in the worst case.

The problem of generating suffix arrays is equivalent to sorting a set of unbounded-length and overlapping strings. Because of those unique features and because our parallelism model is not a classical one, the problem cannot be solved directly with a classical parallel sorting algorithm (we review related work in Section 3).

The proposed algorithm is based on the recursive parallel quicksort approach, where a suitable pivot is found for the whole distributed set of suffixes and the partition phase redistributes the pointers of the suffix array so that each processor has only suffixes smaller or larger than the pivot. A generalization of the parallel quicksort was presented in [12], whose central idea is as follows. Consider the global sorted suffix array which results of the sorting task. If we break this array in n/r similarly-sized portions, we can think that each processor holds exactly one such slice at the end. Thus, the idea is to quickly deliver to each processor the index pointers corresponding to its slice. In summary, the generalized parallel quicksort presented in [12] works with r-percentiles obtained in one step, instead of the binary recursive approach based on one pivot used here. It is also worth to mention a previous parallel mergesort based algorithm presented in [9], which is slower than the algorithm presented here on both average and worst cases.

1.1 Suffix Arrays

To reduce the cost of searching in textual databases, specialized indexing structures are adopted. The most popular of these are *inverted lists*. Inverted lists are useful because their search strategy is based on the vocabulary (the set of distinct words in the text) which is usually much smaller than the text and thus, fits in main memory. For each word, the list of all its occurrences (positions) in the text is stored. Those lists are large and take space which is 30% to 100% of the text size.

Suffix arrays [10] or PAT *arrays* [4, 5] are more sophisticated indexing structures with similar space overhead. Their main drawback is their costly construction and maintenance procedures. However, suffix arrays are superior to inverted lists for searching phrases or complex queries such as regular expressions [5, 10]. In this model, the entire text is viewed as one very long string. In this string, each position k is associated to a semi-infinite string or *suffix*, which initiates at position k in the text and extends to the right as far as needed to make it unique. Retrieving the "occurrences" of the user-provided patterns is equivalent to finding the positions of the suffixes that start with the given pattern.

A *suffix array* is a linear structure composed of pointers (here called *index pointers*) to every suffix in the text (since the user normally bases his queries upon words and phrases, it is customary to index only word beginnings). These index pointers are sorted according to a *lexicographical ordering* of their respective suffixes and each index pointer can be viewed simply as the offset (counted from the beginning of the text) of its corresponding suffix in the text.

To find the user patterns, binary search is performed on the array at $O(\log n)$ cost (where n is the text size). The construction of a suffix array is simply an *indirect sort* of the index pointers. The difficult part is to do this sorting efficiently when large texts are involved (i.e., gigabytes of text). Large texts do not fit in main memory and an external sort procedure has to be used. The best known sequential procedure for generating large suffix arrays takes time $O(n^2/m \ \log m)$ where n is the text size and m is the size of the main memory [5].

1.2 Distributed Parallel Computers

Parallel machines with distributed memory (multicomputers or message passing parallel computers) are a good cost-performance tradeoff. The emergent fast switching technology has allowed the dissemination of high-speed networks of processors at relatively low cost. The underlying high-speed network could be, for instance, an ATM network running at a guaranteed rate of hundreds of megabits per second. In an ATM network, all processors are connected to a central ATM switch which runs internally at a rate much higher than the external rate. Any pair of processing nodes can communicate simultaneously at the guaranteed rate without contention and broadcasting can be done efficiently. Other possible implementations are the IBM SP based on the High Performance Switch (HPS), or a Myrinet switch cluster. Our idea is to use the aggregate distributed memory of the parallel machine to hold the text. Accessing remote memories takes time similar to that of transferring data from a local disk, although with no seek costs [9].

2 Preliminaries

Our parallelism model is that of a parallel machine with distributed memory. Assume that we have a number r of processors, each one storing b text positions, composing a total distributed text of size $n = rb$. Our final suffix array will also be distributed, and a query solved with only $O(\log n)$ remote accesses. We assume that the parallelism is coarse-grained, with a few processors, each one with a large main memory. Typical values are r in the tenths or hundreds and b in the millions.

The fact that sorting is indirect poses the following problem when working with distributed memory. A processor which receives a suffix array cell (sent by another processor) is not able to directly compare this cell because it has no local access to the suffix pointed to by the cell (such suffix is stored in the original processor). Performing a communication to get (part of) this suffix from the original processor each time a comparison is to be done is expensive. To deal with this problem we use a technique called *pruned suffixes*. Each time a suffix array cell is sent to a processor, the first ℓ characters of the corresponding suffix (which we call a *pruned suffix*) are also sent together. This allows the remote processor to perform comparisons locally if they can be decided looking at the first ℓ characters only. Otherwise, the remote processor requests more characters to the processor owning the text suffix cell[7]. We try to select ℓ large enough to ensure that most comparisons can be decided without extra communication and small enough to avoid very expensive exchanges and high memory requirements.

We define now what we understand by a "worst-on-average-text" (WAT) case analysis. If we consider a pathological text such as `"a a a a a a ..."`, the classical suffix array building algorithm will not be able to handle it well. This is because each comparison among two positions in the text will need to reach the end of the text to be decided, thus costing $O(n)$. Since we find such worst-case texts unrealistic, our analysis deal with *average* random or natural language text. In such text the comparisons among random positions take $O(1)$ time (because the probability of having to look at more than i characters is $1/\sigma^i$ for some $\sigma > 1$). Also, the number of index points (e.g., words) at each processor (and hence the size of its suffix array) is roughly the same. A WAT-case analysis is therefore a worst-case analysis on *average* text. We perform WAT-case and average-case analysis.

3 Related Work

For the PRAM model, there are several studies on parallel sorting. For instance, Jájá et al. [8] describe two optimal-work parallel algorithms for sorting a list of strings over an arbitrary alphabet. Apostolico et al. [2] build the suffix tree of a text of n characters using n processors in $O(\log n)$ time, in the CRCW PRAM model. Retrieval of strings in both cases is performed directly. In a suffix array, strings are pointed to and the pointers are the ones which are sorted. If a distributed memory is used, such indirection makes the sorting problem more complex and requires a more careful algorithm design.

The parallelism model we adopt is that of parallel machines with distributed memory. In such context, different approaches for sorting can be employed. For instance, Quinn [13] presents a quicksort for a hypercube architecture. That algorithm does not take into account the variable size and overlapping in the elements to be sorted, as in our problem. Furthermore, the behavior of the communication network in Quinn's work is different (processors are not equidistant) from the one we adopt here.

4 The Quicksort-Based Distributed Algorithm

Our algorithm also utilizes the aggregate memory as a giant cache for disks. Unlike mergesort, the hardest work occurs at the point of higher parallelism. It also improves over the generalized quicksort, because the partitioning is binary and therefore bad biased cases are handled better.

Our algorithm starts by determining the beginning of each suffix in the text (i.e., the beginning of each word) and by generating the corresponding index

[7] As we will see, in some cases this is not necessary and one might assume that the suffixes are equal if the comparison cannot be locally decided.

pointers. Once this is done, the pointers are sorted lexicographically by the suffixes they point to (i.e. the local suffix arrays are built). This task is done in parallel for each of the r blocks of text. Since computation of the whole suffix array requires moving index pointers among processors without losing sight of the suffixes they point to, index pointers are computed relative to the whole text.

The processors then engage in a recursive process which has three parts: (1) find a suitable pivot for the whole distributed set of suffixes; (2) partition the array: redistribute the pointers so that each processor has only suffixes smaller or larger than the pivot (keep local arrays sorted), and (3) continue the process separately inside each group of processors.

This recursion ends when a partition is completely inside a local processor. Since all the time the suffixes at each processor are sorted up to pruning, the process is completed with (4): a final sorting of equal pruned suffixes inside each processor.

We now describe the algorithm more in detail. Let $E(i)$ be the set of index pointers stored in the processor i. Further, let p be a reference to an index pointer and let $S(p)$ be the pruned suffix pointed to by p.

4.1 Finding a Pivot

The goal of this stage is to find a suffix which is reasonably close to the median of the whole set, at a low cost. To achieve this, all processors

(a) take the middle element $m(i)$ of their local suffix array;
(b) broadcast that (pruned) median $m(i)$;
(c) knowing all the other medians, do $m = median\{m(1), ..., m(r)\}$;
(d) binary search the median of medians m in their suffix array, therefore partitioning their index pointers in two sets $L(i)$ and $R(i)$:

$$L(i) = \{p \in E(i) \mid S(p) \leq m\}; \quad R(i) = \{p \in E(i) \mid S(p) > m\} \quad (1)$$

(e) broadcast the sizes $|L(i)|$ and $|R(i)|$ of the computed partitions.

Observe that in part (e) a pruned suffix which is found to be equal to the (pruned) pivot m is put at the left partition. This works well and avoids at all requesting full suffixes to other processors. However, as the algorithm progresses, this pivoting process can worsen the randomness of the partition. Such effect tends to get worse at the final stages of the sorting process.

We proved in [12] that this median of medians is very close to the exact median, and we show in Section 6 that this is the case in practice, even using pruned suffixes. Notice that it is possible to find the *exact* pruned median by using the $O(r \log b)$ process described in [12]. However this would add a complication to the algorithm and does not change the complexities, as we see later.

4.2 Redistributing Pointers

The processors engage in a *redistribution* process in which they exchange index pointers until each processor contains all of its index pointers in either L or R, where

$$L = \bigcup L(i); \quad R = \bigcup R(i) \tag{2}$$

We say that the processor becomes *homogeneous* when this happens. There can be left at most one processor whose index pointers lie in both L and R (we explain later how this is accomplished). This processor is called *non-homogeneous*.

The process of redistributing index pointers is carried out in a number of steps which are completely planned inside each processor (simulating completion times for exchanges) and later followed independently. To accomplish such effect, the processors are paired in a fixed fashion (for instance, pair the processor $(2i)$ with the processor $(2i + 1)$ for all i). Each pair manages to exchange a minimum number of index pointers such that one of them is left homogeneous. The homogeneous processor in each pair is left outside of the redistribution process. The remaining half processors engage in a new redistribution process in which the processor $(4i)$ or $(4i + 1)$ is paired with the processor $(4i + 2)$ or $(4i + 3)$ (depending on which one is still non-homogeneous). Notice that, since all processors have the information needed to predict the redistribution process, they know which processor to pair with at each iteration, and no synchronization messages have to be exchanged. This ends when there is only one non-homogeneous processor.

Let us focus in the task of making one of the processors in a pair homogeneous. Consider the pair composed of processors P_a and P_b. By comparing its suffixes with the computed median m, the processor P_a separates its index pointers according to the internal partition (L_a, R_a). Analogously, the processor P_b separates its index pointers according to the internal partition (L_b, R_b). Let $|L_a|$, $|R_a|$, $|L_b|$, and $|R_b|$ be the number of index pointers in each of these partitions. Without loss of generality, let $\min(|L_a|, |R_a|, |L_b|, |R_b|) = |L_a|$. Then, processor P_a can make himself homogeneous by sending all the index pointers in its partition L_a to processor P_b while retrieving (from processor P_b) $|L_a|$ index pointers of partition R_b. After this exchange, processor P_a is left with all its index pointers belonging to R (and thus, homogeneous) while processor P_b is left with a partition $(L_b \bigcup L_a, R'_b)$, where $R'_b \subset R_b$ and $|R'_b| = |R_b| - |L_a|$. The other cases are analogous. See Figure 1.

Notice that instead of pairing the processors in an arbitrary fashion, we should try to pair processors P_a and P_b such that $|L_a|$ is as close as possible to $|R_b|$, therefore minimizing the amount to transfer and the number of redistribution steps on average (since it is more probable that both processors are left homogeneous or close to). An easy way to do this is to sort the processors by their $|L_a|$ value and then pair the first and last processors, the second and the next-to-last, and so on. This needs not exchange of synchronization messages, because all processors have the necessary information to plan the same exchange sequence.

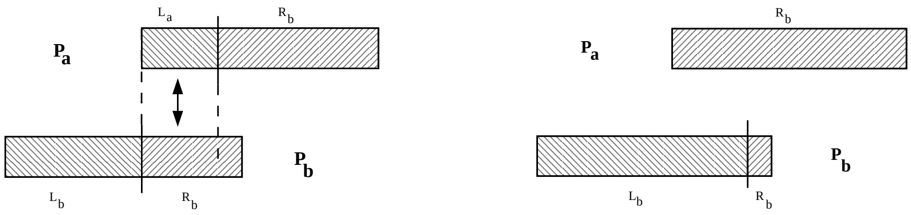

Fig. 1. Illustration of the exchange process. Processor P_a is made homogeneous since it owns the smaller partition. This partition is exchanged for a similarly sized portion of P_b.

Once a processor receives a portion of another suffix array, it merges the new portion with the one it already had. This ensures that the suffixes are lexicographically sorted inside each processor all the time. This is of course true only up to pruning, since equal pruned suffixes are stored in any order. However, those pruned suffixes coming from the same processor are known to be originally in the correct order, and therefore this merging process does not modify the ordering between equal suffixes of the same processor.

4.3 Recursive Step

This redistribution of index pointers splits the processors in two groups: those whose index pointers belong to L and those whose index pointers belong to R. The two groups of processors proceed independently and apply the algorithm recursively.

The non-homogeneous processor could potentially slow down the process, since it has to act in two (parallel) groups. Although it does not affect the total complexity (since a processor belongs at most to two groups), it can affect the constants. To alleviate the problem, we can mark it so that in the next redistribution process it is made homogeneous in the first exchange iteration. It may take longer, but the processor is free for the rest of the iterations.

The recursion ends whenever an L or R set of index pointers lies entirely in the local array of a processor. In this case, all that remains to be done is to sort L or R locally.

4.4 Final Local Sorting

Throughout the process, the suffixes at each processor are sorted up to pruning. Moreover, we guarantee that equal pruned suffixes coming from the same processor are correctly sorted already. We must, therefore, correctly sort all equal pruned suffixes coming from different processors. To decide those comparisons, more characters of the suffixes must be requested to the remote processors owning the suffixes. The number of such remote accesses depends on the text and on the size of the pruned suffixes. Refer to Section 6 for further details.

Therefore, this step proceeds as follows, for each processor: the suffix array is sequentially traversed. Each time a sequence of equal pruned suffixes is found, they are put in r queues, one per originating processor. Inside each queue, the original order of the suffixes is respected. Then, the first heads of all queues are collected and arranged into a heap data structure (each comparison involves requesting remotely more suffix characters). Once the head of the heap is removed, it is replaced by the next element of the appropriate queue, until we sort all elements. With this ad-hoc heapsort we make only the necessary comparisons.

5 Analysis

5.1 WAT Case

We consider the cost $T(r)$ of our distributed algorithm described in Section 4. Since the size of the problem is reduced at each recursion step, the number of processors in the newly generated L and R groups decreases. We consider the cost of a recursive step with r processors initially. The final cost of the recursion is that of solving the subproblems it generates. Note also that there is an initial part outside the recursion, namely the initial local sorting.

The initial cost of sorting locally the suffix arrays is $O(b \log b)$ **I**, since it is done in parallel at each processor.

Apart from this, the cost $T(r)$ of our algorithm for r processors is as follows:

1. Costs for finding the pivot (costs are parallel for all processors i):
   (a) selecting the middle element $m(i)$ is $O(1)$ **I**;
   (b) broadcasting the median $m(i)$ is $O(r)$ **C**;
   (c) computation of the median m is $O(r)$ **I**;
   (d) searching m in the local suffix to determine $L(i)$ and $R(i)$ is $O(\log b)$ **I**;
   (e) broadcasting the sizes $|L(i)|$ and $|R(i)|$ is $O(r)$ **C**.

2. Cost of redistributing index pointers in subproblems L and R is as follows. There are at most $\log r$ steps because at least half of the processors is made homogeneous at each redistribution step. Since at most b index pointers are exchanged in each step (because $\min(|L_a|, |R_a|, |L_b|, |R_b|) \leq b/2$), the total cost is $O(b \log r)(\ \mathbf{I}\ +\ \mathbf{C}\ )$ (we also count the factor **I** because of the merging between the old and new pointers).

3. Cost for the recursive calls (processing of groups L and R) depends on the worst-case partition. Let r_L be the number of processors in group L and r_R be the number of processors in R.

   We show that $r/4 \leq r_L \leq 3r/4$ in the worst case: observe that the estimated median m is larger than $r/2$ local medians, each one in turn larger than $b/2$ elements of the corresponding processor. Hence, m is larger than $n/4$ elements which implies that r_L is larger than $r/4$. The proof for the upper bound is analogous.

Hence, there are at most $\log_{4/3} r$ levels in the recursion in the worst case. The number of processors in the larger partition is at most $3/4r$ (the smaller partition works in parallel and does not affect completion times). Therefore, $T(3/4r)$ must be added to $T(r)$.

4. Cost of sorting the index pointers locally. In the worst case the suffixes are all equal and the same number originated at each processor. In this case the heapsort is $O(b \log r)(\ \mathbf{I}\ +\ \mathbf{C}\ )$. Note that this r is the original one, independent of the recursion (we call it r_0). Notice also that this worst case analysis does not improve significantly if instead of a long run of equal pruned suffixes there are many short runs (except when the runs are so short that $\log r$ becomes very pessimistic).

The complexity of the total execution time is given by the recurrence

$$\begin{aligned}
&T(1) = O(b \log b)\,\mathbf{I} + O(b \log r_0)(\ \mathbf{I} + \mathbf{C}\ ) = O(b \log n)\,\mathbf{I} + O(b \log r_0)\ \mathbf{C} \\
&T(r) = O(r + b \log r)\ \mathbf{I} + O(r + b \log r)\ \mathbf{C} + T(3/4\ r)
\end{aligned}$$

which gives

$$T(r) = O(r + b \log r \log n)\ \mathbf{I} + O(r + b \log^2 r)\ \mathbf{C}$$

where we can assume $r < b$ to obtain $T(r) = O(b \log r \log n)\ \mathbf{I} + O(b \log^2 r)\ \mathbf{C}$. The communication complexity is better than all previous work.

This part of the complexity is the most important in practice (as the remote accesses cost much more than CPU operations). Hence, we concentrate in communication costs. The exact constants for the main part of the cost are given by $b \log_2 r \log_{4/3} r$.

If we replace the estimated median algorithm by the one given in [12] that obtains the exact median, we have a cost of $O(r \log b)$ instead of $O(r + \log b)$ in Step (1). As a compensation, the partition is exact and therefore there are exactly $r/2$ processors on each side. Redoing the analysis for this case we get

$$T(r) = O(r \log b + b \log r \log n)\ \mathbf{I} + O(r \log b + b \log^2 r)\ \mathbf{C}$$

which is the same as before when we consider $r < b$. However, the constants of the main part of the cost improve, namely the communication cost becomes $b \log_2^2 r$.

We consider scalability now. If we double n and r, the new cost $T(2n, 2r)$ becomes

$$T(2n, 2r) = T(n, r) \times$$

$$\left(1 + \frac{r + b(1 + \log_{4/3} r + \log_2 n)}{r + b \log_{4/3} r \log_2 n}\ \mathbf{I} + \frac{r/\ln(2) + b(2 \log_2 r + 1)}{r + b \log_2 r \log_{4/3} r}\ \mathbf{C}\right) = 1 + o(1)$$

which as long as $r < b$ is

$$T(2n, 2r) = T(n, r)\left(1 + O\left(\frac{1}{\log r}\right)\ (\ \mathbf{I}\ +\ \mathbf{C}\ )\right)$$

(the ideal scalability condition is $T(2n, 2r) = T(n, r)$). While our algorithm does not scale ideally, it does scale much better than previous algorithms (whose scaling factor is 2 in the WAT case). Further, as the number of processors increase, the additional computational time (given by the fraction $1/\log r$) drops considerably. For instance, if the number of processors doubles from 256 to 512 the execution time goes up by a factor of 25% (instead of also doubling).

5.2 Average Case

We show in this section that the algorithm works almost optimally in the average case. The most involved part of the proof is to show that, for large n, the estimated median is almost the exact median. We have proved it in [12] (i.e. the local median is off the global median by a factor of $O(n^{-1/2})$). The proof is obtained by considering only one processor, as a process where the global median is estimated by sampling b elements out of n. When the median of r so computed medians is used, the estimation is even better. Therefore, the distance between the real median and the middle of the local array is $O(\sqrt{b/r})$.

Once we prove that, we have that each redistribution session exchanges almost all the data in a single step (since $|L_a| \approx |L_b| \approx |R_a| \approx |R_b|$), being the remaining steps so small (in terms of communication amounts) that can be ignored. The first iteration exchanges $O(b)$ elements, and the rest exchange portions of the array of size $O(\sqrt{b/r})$. Therefore, the cost of the $O(\log r)$ exchanges is $O(b + \sqrt{b/r} \log_2 r) = O(b)$. It is also possible to perform the merges between old and new arrays at the same cost.

Moreover, since the partition is almost perfect, $|L| \approx |R|$, and the next subproblems are almost half the size of the original one, the logarithm previously in base $4/3$ is now base 2 and the network is used all the time. To see this, observe that instead of adding $T(3/4\ r)$ to $T(r)$, we add $T((b/2 + \sqrt{b/r})\ r/b) = T(r/2 + \sqrt{r/b}) = T(r/2 + o(1))$, which makes the final cost $b \log_{2/(1+o(1))} r = b \log_2 r\ (1 + o(1))$. Therefore, the average time cost of our algorithm is

$$T(r) = O(r + b \log n)\ \mathbf{I}\ +\ O(r + b \log r)\ \mathbf{C} = O(b \log n)\ \mathbf{I}\ +\ O(b \log r)\ \mathbf{C}$$

(the simplification being valid for $r < b$). The scalability factor for communication becomes $1 + (r + b)/(r + b \log_2 r)$, i.e. of the same order but about a half of that of the WAT case, while for CPU costs it is $1 + O(1/\log n)$. Despite this improvement, the algorithm [12] has better average complexity.

The non-homogeneous processor does not add too much to the cost, since it has $\approx b/2$ elements in each partition, and hence exchanges $\approx b/4$ on each group. This takes the same as exchanging $b/2$, which is the normal case in the other processors.

The final sorting can be cheaper on average than $O(b \log r)$. However, this analysis is much more difficult and highly dependent on the nature of the text and the length of the pruned suffixes. We can make it cheaper by using longer pruned suffixes (and pay more communication cost) or vice versa. Moreover, the big-O analysis does not change because there are already other $O(b \log r)$ costs involved. We leave this point for the experiments that follow.

6 Experimental Results

Although we have not implemented yet the parallel algorithm here presented, we performed a preliminary analysis of its behavior taking into account some real texts (Wall Street Journal extracts from the TIPSTER collection [6]). We study the critical aspects of the behavior of the algorithm.

6.1 Pruned Suffixes

One phenomenon of interest is the effect of pruned suffixes in the algorithm. Suffixes are pruned at ℓ characters in order to reduce interprocessor communication of processes asking remote suffixes for local comparison. Pruning influences the whole algorithm because, after the first step of recursion, pointers may point to remote suffixes. We evaluate here the implications of pruning on interprocess communication.

We begin by considering the last local sorting. We implemented an external merge of r queues using a heap. We obtain the fraction of comparisons that generated remote accesses for more characters of the suffixes (these correspond indeed to a tie between pruned suffixes). In Table 1 we present average and standard deviation (stdev) for different block sizes and ℓ values, considering an input file of 10 megabytes.

In turn, each tie implies two to four remote accesses (depending on just one or both are remote pruned suffixes). This is because there is a request and an answer for each suffix retrieved. However, suffixes already brought from a remote processor can be buffered locally in order to solve eventual posterior ties, with no need to ask them again remotely.

We also counted the number of messages really exchanged among processors, if this local buffering is used. Let $ties$ be the total number of ties occurring on a given processor. In the same table we present, in the sixth column, the fraction of the messages exchanged when compared with the worst case (that is, $4 * ties$). This gives a measure of the effectiveness of the local buffering scheme.

We present also the maximum number of messages sent in each case (i.e., the number of messages of the most communicant processor) normalized in percentage to the number of total suffixes on the corresponding processor. Since all processors work in parallel, this is related to the global completion time for Step 4 of the algorithm.

Finally, we estimate the time in seconds to transfer this maximum number using the model of [11] for smaller messages (see Section 6.3: $\alpha = 47$ and $\tau = 0.0254$) and considering a message of 8 bytes to request (suffix pointer plus processor address) and 54 to answer (processor address plus 50 bytes of the suffix).

The results show that a pruned suffix of 30 characters is already a good trade-off ($< 5\%$ remote requests in almost all cases). We observe that the variation between the percentage of ties among processors is rather high. As a matter of fact, the larger the pruned suffix, the larger the variation of the percentage of

T	P	ℓ	% ties	stdev	messages / $4 * ties$	stdev	max mess. / # suffix	estimated time (s)
10 Mb	8	10	27.84%	9.25%	20.53%	5.31%	2.50%	8.28
10 Mb	8	20	7.51%	29.20%	28.00%	15.93%	0.88%	2.90
10 Mb	8	30	4.10%	35.23%	26.50%	15.89%	0.44%	1.46
10 Mb	8	40	2.54%	37.14%	28.61%	15.10%	0.29%	0.94
10 Mb	16	10	24.74%	19.78%	13.13%	16.30%	2.97%	4.91
10 Mb	16	20	6.55%	51.49%	20.76%	46.92%	1.29%	2.12
10 Mb	16	30	3.67%	62.17%	20.23%	51.16%	0.66%	1.09
10 Mb	16	40	2.22%	76.54%	22.47%	43.79%	0.49%	0.80

Table 1. Amount of exchanged messages due to pruning (stdev is a percentage over the average). T is the text size and P the number of processors.

ties. For example, for ℓ equal to 10 and 8 processors, we obtained a standard deviation of 9.25% over the average. For ℓ equal to 40, this percentage increases to 37.14%. This means that larger pruned suffixes imply few ties (and few remote accesses), but more text is stocked locally and text characteristics (distribution of words and phrase composition) start to influence the occurrence of identical suffixes. For example, "Wall Street Journal" (19 characters) occur frequently in the text database we use. The processor containing suffixes starting with `"W"` may ask more remote suffixes than other processors.

Another interesting point is compression. To reduce communication overhead when exchanging suffix arrays, we use a compression scheme based on similarity of pruned suffixes. Since the processor that sends a slice will send all the pruned suffixes in ascending order, most suffixes will share a common prefix with their neighbors. This can be used to reduce the amount of communication. This technique has been previously applied to compress suffix array indices [3], and works as follows: the first pruned suffix is sent complete. The next ones are coded in two parts: the length of the prefix shared with the previous pruned suffix; and the remaining characters. For example, to send `"core"`, `"court"` and `"custom"`, we sent `"core"`, `(2,"urt")` and `(1,"ustom")`.

Compression rates (i.e. compressed size divided by uncompressed size) averages and standard deviation are presented in Table 2. With an ℓ of 30 characters, a 25% of reduction is achieved. As expected for lower ℓ, compression may reduce the size of the pruned suffixes to almost the half of the size. However, as presented in Table 1, a small ℓ implies more communication during the local sort of pointers. We also verify that compression rates are also sensible to the text size. The larger this size, the better the compression, due to the higher degree of similarity between contiguous suffixes in the sorted array. Note that we measure compression rates in the first exchange. This should improve in further steps of the recursion, since the suffixes become more and more sorted and therefore longer prefixes are shared among contiguous suffixes.

text size	# proc.	ℓ	mean compression rate	stdev compression rate
10 Mb	8	10	56.06%	4.26%
10 Mb	8	20	65.99%	4.41%
10 Mb	8	30	73.46%	3.93%
10 Mb	8	40	78.19%	3.40%
10 Mb	16	10	59.17%	6.90%
10 Mb	16	20	69.05%	6.54%
10 Mb	16	30	75.90%	5.85%
10 Mb	16	40	80.23%	5.11%

Table 2. Percentage of compression (average and percentage of stdev over the average).

6.2 Estimated Medians

We have generated the suffix arrays and the corresponding file of sorted suffixes for two extracts of the Wall Street Journal [6]. These two extracts have 32 and 100 megabytes. We partitioned these files in 8 and 16 blocks, and used $\ell = 30$. Then, we obtained the medians of the blocks ($m(i)$) and computed m, the median of the medians. Next we compared:

- m with the real median of the whole extract (called D_1);
- m with each local $m(i)$ (called $D_2(i)$).

We present the distance (in percentage) from the real median. If it is the exact median, the percentage is 0%. If it corresponds to the first or last element of the local suffix array, the deviation is of 100%. The results for the maximum deviations are presented in Table 3.

	100Mb-8P	100Mb-16P	32Mb-4P	32Mb-8P	32Mb-16P
D_1	0.42%	0.35%	0.11%	0.16%	0.06%
max ($D_2(i)$)	1.49%	0.98%	0.29%	0.75%	1.51%

Table 3. Deviation among real and estimated medians. Text sizes of 32 and 100 megabytes and number of processors of 4, 8, and 16.

According to the numbers presented in Table 3, the text presents a characteristic of auto-similarity, that is, the text blocks on each processor have similar medians, which are in turn similar to the exact global median. Approximate medians (those considering the median of medians, that is, m) are taken on pruned suffixes. Therefore, even using pruned suffixes with a reasonable ℓ, we obtain good approximations ($< 2\%$).

We did not go on with the partitioning process, but we also estimated what would happen in the last steps of the recursion process. In these last levels, the compared suffixes are much more similar and the median approximation is based on few samples (but on a smaller text space). For this approximation, we took the global sorted suffix file called GS. We divided GS in 8 and 16 blocks (GS_i, where $1 < i < 8$ or $1 < i < 16$) and took the two first blocks GS_1 and GS_2 (for example, comprising suffixes starting with `"A"` until `"C"`). Next we took each suffix of these two initial blocks and chose randomly a processor to hold it (keeping the lexicographical order - since GS has sorted suffixes). Finally, we took the median on each processor and compared with the real median (the last suffix of GS_1 or the first of GS_2). Results are presented in the Table 4 for a 100 megabytes file.

file size	100 Mb	100 Mb
block size fraction	1/8	1/16
distance block 1	0.06%	0.10%
distance block 2	0.07%	0.10%

Table 4. Deviation among real and estimated medians in part of the last step of the recursion. Simulation is used.

The estimated medians on pruned suffixes are very close to the real median. This shows that the estimation is even better in the last steps of the recursion, even considering the effects of pruning. It is important to remark that in both cases (Tables 3 and 4) the approximations are very good for $\ell = 30$ and different number of processors. This is expected considering that the number of samples is proportionally the same when compared to the size of the text being sampled: e.g., for 16 processors, we sample 16 medians for the whole text. With 2 processors in the last step, we sample 2 medians, but from a text 8 times smaller.

6.3 Partition Exchange

We know that if the partitions are exact (i.e., m is always identical to the real median of the (sub)set), the partition (L or R) exchanges are performed in one step and without interference among pairs (using a no contention switch). In general, communication in parallel machines can be modeled by a linear equation [11]:

$$t_{com} = \alpha + \tau s_p$$

where t_{com} is the total communication time, α is the time spent to startup the line and other eventual overheads (e.g., packing), τ is the time to send a byte through the communication link, and s_p is the partition size in bytes. If the partitions are exactly equal in size, all the partition exchanges are done in one turn. For the examples of Section 6.2 (texts of 32 and 100 megabytes; 4, 8, and

16 processors; and $\ell = 30$), we present in Table 5 the estimated time of the first partition exchange of the recursion. We use the parameters of a typical IBM SP parallel machine: $\alpha = 390\mu s$, $\tau = 0.115\mu s/byte$ [11][8]. It is important to remark that one *partition* is composed of integers and the pruned suffixes. For Table 5, we estimated the communication time without using a compression scheme.

	100Mb 8P	100Mb 16P	32Mb 4P	32Mb 8P	32Mb 16P
estimated time	5.06	2.54	3.68	1.85	0.93

Table 5. Estimated communication time in seconds of partitions for the first iteration of the recursion. Non-homogeneous processors are not considered.

Using the measures of Section 6.2, we can estimate the communication time loss due to unequal partitions. However, due to the regularity of the partitions, the remaining bytes to make processors homogeneous are not representative. For example, we take the 32 megabytes text and make a simulation of partition exchange, considering different sizes of partitions (we consider, for each, communication turn, the largest message exchanged and 8 processors):

1. for the above case, the first communication is of $469,011 * 34$ bytes. This will consume around 1.83 seconds (1,834.22 milliseconds). Half of the processors are made homogeneous;

2. next, we check the number of bytes to be exchanged to make more one fourth processors homogeneous. We have then to communicate $112 * 34$ bytes, corresponding to 0.83 milliseconds, i.e., 0.05% of the previous time;

3. finally, $12 * 34$ bytes are sent to make one processor homogeneous and other non-homogeneous with $100*34$ more bytes than the other processors. This last exchange consumes 0.44 milliseconds, i.e, 0.02% of the partition exchange original time.

7 Conclusions and Future Work

We have discussed a quicksort-based distributed algorithm for the generation of suffix arrays for large texts. The algorithm is executed on processors connected through a high-bandwidth network. We have shown how to deal with the particular aspects of suffix arrays, namely the unbounded size and overlapping of the elements. We analyzed the average and worst case complexity of our algorithm considering a text of size n and the presence of r processors. Such analysis

[8] The IBM SP has different linear communication models for different sizes of messages. The α and τ of Section 6.1 correspond to small messages (< 4 kilobytes). The parameters used here correspond to larger messages (> 32 kilobytes).

proves that the algorithm has the best communication complexity and scaling factor in the WAT case. A comparative table follows.

Algorithm	Complexity		Scaling Factor $(1 + ...)$	
	WAT	Average	WAT	Average
Mergesort [9]	$n\ (\ \mathbf{I}\ +\ \mathbf{C}\ )$	$n\ (\ \mathbf{I}\ +\ \mathbf{C}\ )$	$\mathbf{I}\ +\ \mathbf{C}$	$\mathbf{I}\ +\ \mathbf{C}$
Generalized Quicksort [12]	$b \log n\ \mathbf{I}$ $+\ n\ \mathbf{C}$	$b \log n\ \mathbf{I}$ $+\ b\ \mathbf{C}$	$1/\log n\ \mathbf{I}$ $+\ \mathbf{C}$	$1/\log n\ \mathbf{I}$
Quicksort (present work)	$b \log r \log n\ \mathbf{I}$ $+\ b \log^2 r\ \mathbf{C}$	$b \log n\ \ \mathbf{I}$ $+\ b \log r\ \mathbf{C}$	$1/\log r\ (\ \mathbf{I}\ +\ \mathbf{C}\ )$	$1/\log n\ \mathbf{I}+$ $1/\log r\ \mathbf{C}$

We are currently working on the implementation of the quicksort based algorithm in order to have real experimental times instead of simulations. We also plan to repeat the experiments with larger texts for the final version.

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. A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.

3. E. Barbosa and N. Ziviani. From partial to full inverted lists for text searching. In R. Baeza-Yates and U. Manber, editors, *Proc. of the Second South American Workshop on String Processing (WSP'95)*, pages 1–10, April 1995.

4. 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.

5. G. 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.

6. D. Harman. Overview of the third text retrieval conference. In *Proceedings of the Third Text Retrieval Conference - TREC-3*, pages 1-19. National Institute of Standards and Technology. NIST Special Publication 500-225, Gaithersburg, Maryland, 1995.

7. J. Jájá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

8. J. Jájá, K. W. Ryu, and U. Vishkin. Sorting strings and constructing digital search trees in parallel. *Theoretical Computer Science*, 154(2):225–245, 1996.

9. J. P. Kitajima, B. Ribeiro, and N. Ziviani. Network and memory analysis in distributed parallel generation of PAT arrays. In *14th Brazilian Symposium on Computer Architecture*, pages 192–202, Recife, August 1996.

10. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22, 1993.

11. 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.

12. G. Navarro, J. P. Kitajima, B. Ribeiro, and N. Ziviani. Distributed generation of suffix arrays. In A. Apostolico and J. Hein, editors, *Proc. of the Eighth Symposium on Combinatorial Pattern Matching (CPM97)*, Springer-Verlag Lecture Notes in Computer Science v. 1264, pages 102-115, Århus, Denmark, June 1997.

13. M. J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, second edition, 1994.

This article was processed using the LaTeX 2$_\varepsilon$ macro package with CUP_CS class