Distributed Generation of Suffix Arrays

Gonzalo Navarro [1] *
João Paulo Kitajima [2] **
Berthier A. Ribeiro-Neto [2] ***
Nivio Ziviani [2] †

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

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 among them. The underlying architecture is that of a high bandwidth network of processors. Our algorithm builds the suffix array by quickly assigning an independent subproblem to each processor and completing the process with a final local sorting. We demonstrate that the algorithm has time complexity of $O(b \log n)$ computation and $O(b)$ communication in the average case, where b corresponds to the local text size on each processor (i.e., text size n divided by r, the number of processors). This is faster than the best known sequential algorithm and improves over previous parallel algorithms to build suffix arrays, both in time complexity and scaling factor.

1 Introduction and Motivation

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 (connected by a fast switch like ATM, for example) 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 generalized quicksort presented in [7, 15]. The algorithm is an alternative to a previous mergesort-based distributed algorithm [10, 16] and to a pure quicksort-based algorithm [18]. We show that the here proposed algorithm is faster and, more important, that it scales up well while the mergesort-based algorithm does not.

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. 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

* This author has been partially supported by Fondecyt grant 1-950622 (Chile).
** 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.

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 [15] presents a quicksort for a hypercube architecture. That algorithm does not take into account the variable size and overlapping in the elements of our problem. Further, the behavior of the communication network in Quinn's work is different (processors are not equidistant one from each other) from the one we adopt here.

1.1 Suffix Arrays

The advent of powerful processors and cheap storage has allowed the consideration of alternative models for information retrieval other than the traditional one of a collection of documents indexed by keywords. 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. The user expresses his information need via words, phrases or patterns to be matched for and the information system retrieves those documents containing the user specified strings. While the cost of searching the full text is usually high, the model is powerful, requires no structure in the text, and is conceptually simple [5].

To reduce the cost of searching a full text, 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 close to the text size.

Suffix arrays [13] or PAT *arrays* [4, 5] are more sophisticated indexing structures which also take space close to the text size. Their main drawback is their costly construction and maintenance procedures (i.e., creating and updating a suffix array). However, suffix arrays are superior to inverted lists for searching phrases or complex queries such as regular expressions [5, 13].

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. Figure 1 illustrates the suffix array for a text example with nine text positions.

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., texts of gigabytes). 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 $O(n^2 \log n \; / m)$ where n is the text size and m is the size of the main memory [5].

Suffix arrays come from the idea of building a digital search tree on all the suffixes of a text. Such

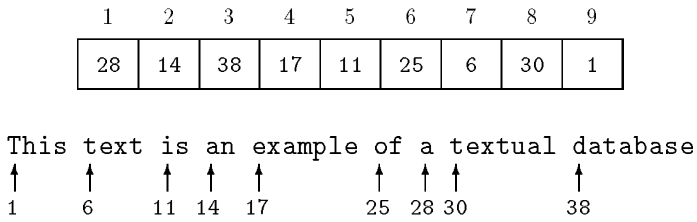

Fig. 1. A suffix array.

search tree allows one to find all the occurrences of a pattern of length m in $O(m)$ time. To reduce the high space requirements, a Patricia tree can be used [14], which compresses unary paths to achieve $O(n)$ storage cost. A Patricia tree built over all suffixes of the text is called a suffix tree [12]. Suffix trees take time $O(n)$ to build [20]. However, this construction is only practical if the tree fits in main memory. Suffix arrays further reduce the space requirements by storing only the leaves of suffix trees. Recently, an intermediate structure between suffix trees and suffix arrays has been proposed [9].

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 at the guaranteed rate without contention. Further, the communication between machines A and B does not interfere with the communication between machines C and D and broadcasting can be done efficiently. Other possible implementations are the IBM SP machine or a Myrinet cluster.

Our idea is to use the aggregate distributed memory of the parallel machine to hold the text. Accessing this aggregate memory requires frequent accesses to remote data (across the network) which take time similar to the time to get data from a local disk at transfer rate [10, 16]. Despite this relatively high remote data access time, use of the distributed aggregate memory to hold the text gives us two critical advantages. First, the aggregate memory allows random access to the data at uniform cost, which we do not have with local disks. Second, we can split our problem in smaller parts and work on them in parallel.

The algorithm we propose is suitable for an environment in which the indexing task is parallelized but the final index is stored at a single processor for sequential query processing. However, the final index may be left distributed along the participant machines.

In a distributed environment, the index can be distributed in two different ways. In the first one, each processor builds a local separate index relative to its local text only. The main drawback of this approach is that each query must be broadcast to every processor and the partial results must be later merged. Despite the high parallelism among processors, this strategy reduces concurrency because queries have to be processed sequentially (i.e., one after the other). In the second and more challenging scheme, a global index is computed and then partitioned among the processors, such that each processor holds a lexicographical interval of the index (e.g. a range of words in dictionary order). In this case, a query is normally directed to a few processors. Despite the low parallelism, concurrency is increased at query time and the system throughput (i.e., number of queries processed in a unit of time) tends to improve.

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 is 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 very expensive. To deal with this problem, we use a technique called *pruned suffixes* which works as follows. 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 [5]. 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. In Section 6.2 we find experimentally good values for ℓ.

Before entering into the algorithm itself we put in clear 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 analysis unrealistic and probably useless, 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 The Proposed Algorithm

The central idea of the algorithm is as follows. Consider the global sorted suffix array which results of the sorting task. If we cut this array in b similarly-sized portions (which we call *slices*), 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.

[5] 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.

We recall the definition of a *percentile*. An α-percentile is the value at position αn in the global sorted suffix array. For example, the $(1/r)$-percentile is the element at position b. Our algorithm partitions the data to be worked on by each processor by finding the percentiles $1/r$, $2/r$, ... $(r-1)/r$. An alternative definition for slice is: the portion of the global suffix array between two consecutive (i/r)-percentiles.

The algorithm proceeds in four steps:

Step 1: Every processor builds internally its local suffix array.
Step 2: The processors cooperate to find the r global (i/r)-percentiles. This defines the portion of each slice stored on each processor.
Step 3: The processors engage in a distribution process so that every processor gets the part of its slice stored on any other processor.
Step 4: Every processor completes internally the sorting of its slice.

The analysis is divided in two parts: CPU internal cost for the processors, which is indicated by a factor **I**, and communication cost, which is indicated by a factor **C**. CPU operations occur in parallel while communication operations may occur in parallel between distinct pairs of processors.

3.1 Internal Sorting

For this first step, each processor traverses its local text, finds the index points of interest (e.g., beginning of words), and builds an array with all the positions of those index points. The pointers must be shifted to reflect the offsets in the global text, not the local one. Once this is done, the array must be sorted by the suffix each position points to.

Since the text is local, the cost of this step is $O(b \log b)\mathbf{I}$ in the average and WAT case.

3.2 Finding the Percentiles

Once every processor has sorted its local suffix array, all the processors must collaborate to find the r global percentiles. We first use the median (0.5-percentile) to explain the technique.

It is well known that given two sorted arrays A_1 and A_2 of total size n, the median of $A_1 \cup A_2$ can be obtained in $O(\log n)$. The algorithm proceeds by binary searching on both arrays simultaneously. The search of the median is performed even without knowing the median.

We keep two positions i_1 and i_2, one for each array. The sum of the two positions is always n. If we could find i_1, i_2 such that $i_1 + i_2 = n$ and $A_1[i_1] = A_2[i_2]$, that would be the median, since that value would be in the middle of the sorted union of both arrays.

We first look at the middle of both arrays, i.e., $i_1 = i_2 = n/2$. If $A_1[i_1] < A_2[i_2]$, we conclude that $A_1[i_1] \leq median \leq A_2[i_2]$, and therefore binary search adds $n/4$ to i_1 and subtracts $n/4$ from i_2. The other case is symmetric. In $O(\log n)$ steps the median is found. We are ignoring boundary conditions in this exposition (for instance, it might be that there is no exact median in an array of size $2n$), since their effect in the algorithm is negligible.

Now imagine we have r arrays of size b and want the global median. We begin in the middle of all of them. The $\lfloor r/2 \rfloor$ smaller values must increment their position, while the $\lfloor r/2 \rfloor$ larger must decrement it. At the end of the multiple binary search, the median of all the r final values is the global median.

If we consider that every array is hold by one processor, we obtain that at each step, every processor must broadcast its current value, which costs $O(r)\mathbf{C}$. Since $O(\log b)$ steps are carried out, the cost to extract the global median is $O(r \log b)\mathbf{C}$.

The algorithm to find general α-percentiles is conceptually the same. As before, all arrays start in their middle positions. However, instead of selecting the median of the r values, we select the α-percentile. Therefore, $\lfloor \alpha r \rfloor$ processors increment their position and $\lfloor (1-\alpha) r \rfloor$ decrement it. The rest proceeds the same as before. A binary search is performed at each array, and the percentile sought drives the number of processors increasing or decreasing their position. Note that in this case it is not true that at any moment the sum of all the positions equals αn. However it is easy to show that with this strategy that sum converges to the correct value after $\log_2(n(1-2\alpha))$ steps for $\alpha \leq 1/2$ (the case $\alpha > 1/2$ is symmetric). Therefore, the algorithm converges to the correct sum before the end of the binary search.

Since each percentile must be found separately, the total cost of this algorithm is $O(r^2 \log b)\mathbf{C}$ in the average and WAT case (it is true that a percentile found can reduce the search area for the others, but the gain is marginal).

Observe that the processors cannot send the complete text suffixes when they broadcast their values, but only pruned suffixes. The first ℓ characters are compared and equality is assumed if the comparison cannot be decided. Therefore, additional characters are never requested. This involves some details to deal with. First, care must be exercised to ensure that, at each step, exactly $\lfloor \alpha r \rfloor$ processors move their position forward and $\lfloor (1-\alpha) r \rfloor$ move backward, even in the case of repeated values. Second, when the value of the final (pruned) percentile is known, the processors must agree on a complete (not pruned) percentile to perform all internal partitions consistently. For example, they can put to the left the suffixes that, once pruned, are smaller or equal to the pruned percentile.

Therefore, the obtained slices can be slightly different in size because of the possible lack of precision when comparing pruned suffixes. These errors are negligible on normal text and the affected processor can easily absorb the few extra items (cf. Section 6.1). Since the number of percentiles broadcast along this process is small, a large ℓ value can be used to ensure a good partition.

3.3 Redistributing the Slices

Once every processor knows the r uniform percentiles, it knows the local slice in its suffix array that must be sent to every other processor. At this point they engage in a redistribution process to send to each other processor the corresponding local slice. This process must ensure that every pair of processors gets a chance to exchange their slices.

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

In general, the exchange mechanism works as follows. At stage i, we ensure that every processor p is paired with processors $p+i$ and $p-i$ (for simplicity, we speak *modulo* r in this passage). This can always be accomplished with three rounds of pairing. To show this, we distinguish groupings of pairs of processors that can communicate all in parallel (grayed in the Figure). In the first round of stage i, the

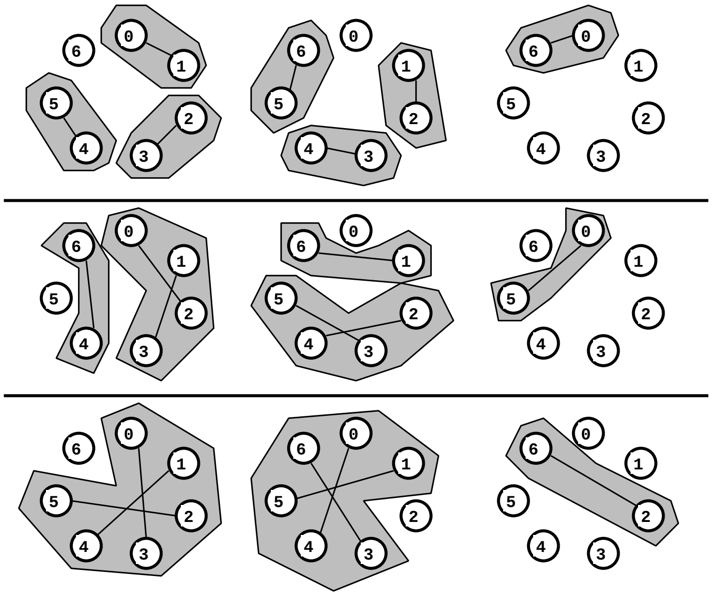

Fig. 2. A redistribution process with 7 processors.

groupings of pairs are $[(0, i), (1, i+1), ..., (i-1, 2i-1)]$, $[(2i, 3i), (2i+1, 3i+1), ..., (3i-1, 4i-1)]$, and so on. In the second round of stage i, the groupings of pairs are $[(i, 2i), (i+1, 2i+1), ..., (2i-1, 3i-1)]$, $[(3i, 4i), (3i+1, 4i+1), ..., (4i-1, 5i-1)]$, and so on. Notice that, in general, the number of processors in a grouping is $2i$. Whenever r is a multiple of $2i$, all pairings in stage i are accomplished with only two rounds. However, if r is not a multiple of $2i$, processors might be left unpaired in the first and second rounds. In this case, a third round is required, which pairs exactly the couples left out in the other two rounds.

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

Hence, we need a total of $3\lfloor r/2 \rfloor = O(r)$ exchange rounds. Since each round takes time proportional to the largest exchange in the round, we have a WAT case of $O(b)\mathbf{C}$ cost per round, for a total WAT case of $O(n)\mathbf{C}$. However, we prove in Appendix A that the cost of each round is on average $O(b/r)$, even taking into account that we wait for the slower exchange in the round. Therefore, the average cost of this step is $O(b)\mathbf{C}$ (we verify this fact experimentally in Section 6.1). Recall that the processors need to exchange not only the elements of the suffix array, but also the first ℓ characters of each suffix pointed to by each element. This allows the target processor to complete the sorting without asking the suffix to the owner of the text in most cases.

3.4 Final Sorting

Once each processor obtained all the elements of its slice, the process is completed by an internal sorting. Observe, however, that the situation is not the same as in the initial sorting, because the elements point to remote text, and therefore reliance on the pruned suffixes transmitted together with the pointers is necessary. Another difference is that the elements are arranged in r sorted sequences (i.e., the slice sent by each processor).

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")`. In Section 6.2 we show that gains near 50% can be expected.

Since we have r sorted sequences, we use a heap to merge them at $O(b \log r)\mathbf{I}$ cost in the average and WAT case. This has an additional advantage: the r local slices received are accessed sequentially and therefore can be stored on disk with little penalty. This is important because the set of all suffix array cells plus their pruned suffixes may not fit in main memory.

Additional communication may be necessary to break ties between equal pruned suffixes. More text may be retrieved from the processors owning the texts. However, as explained, we use long enough pruned suffixes to guarantee that this will occur so infrequently in practice that its effect can be neglected (cf. Section 6.2).

4 Analysis

We compute the global cost of the algorithm. We perform the analysis in terms of r and b, as well as a simplification that is valid whenever $r = o(\sqrt{b/\log b})$, which holds in practice.

Summing up the costs of the algorithm shows an average case of

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

while in the WAT case we have

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

The CPU time improves over the sequential algorithm [5], which is $O(n^2 \log n/b)$ time (assuming $m = b$), by a factor of $\Theta(r^2)$ (this is because the sequential algorithm is not optimal but tries to minimize seek time). The improvement over an optimal sequential algorithm of cost $O(n \log n)$ is $\Theta(r)$, which is optimal.

To analyze the scalability of the algorithm, we consider how the cost is increased if we double the text size and the number of processors, i.e.,

$$\frac{C(2n, 2r)}{C(n, r)} = \frac{b \log(2n)}{b \log n}\,\mathbf{I} + \frac{4r^2 \log b + b}{r^2 \log b + b}\,\mathbf{C} = 1 + O\left(\frac{1}{\log n}\right)\mathbf{I} + O\left(\frac{r^2 \log b}{b}\right)\mathbf{C} = 1 + o(1)$$

which is very good for the practical values involved, though not for very large r.

We compare now the complexity against other parallel algorithms. In [10] a mergesort-based parallel algorithm is proposed, which is $O(b \log n)\mathbf{I} + O(n)\mathbf{C}$ in the average and WAT case. The WAT case is similar to ours, but our average case is much better. In [18], a recursive quicksort-based parallel algorithm is presented, which is $O(b \log n)\mathbf{I} + O(b \log r)\mathbf{C}$ on average and $O(b \log n)\mathbf{I} + O(b \log^2 r)\mathbf{C}$ in the WAT case. Although our average case is better, their WAT case is better than ours. This is because they use a process of pivoting and partitioning by half which allows bad partitions (i.e., one taking more processors than the other). The partition continues until each processor contains a slice. Our present algorithm can be seen as a version of the above procedure in which partitions are built in just one step, losing however the flexibility to handle bad partitions efficiently.

5 A Simpler Algorithm

We show experimentally in Section 6.1 that it is not necessary in practice to compute the exact percentiles. A quick approximation works equally well. This allows to devise a simpler algorithm with the same average case, although the WAT case is worse.

This algorithm replaces Step 2 of the previous one. Instead of engaging in a process of computing the global percentiles, each processor broadcasts its local r uniform percentiles (this costs $O(r^2)\mathbf{C}$). Every processor receives all the percentiles and *estimates* the global percentiles by taking the median of the samples. The rest proceeds in the same way.

We prove in Appendix B that the deviation from the actual values is extremely small on average (i.e., $O(1/\sqrt{n})$), and our experiments in Section 6.1 confirm these assertions. Therefore, the average case cost of this algorithm is

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

To analyze the WAT case, we find the maximum size of an approximated slice. Suppose that we compute an α-percentile. Since every processor broadcasts a value which is larger than αb local items, the median of the values is guaranteed to be larger than $\alpha/2\ n$ elements. With the same argument, it is guaranteed to be smaller than $(1-\alpha)/2\ n$ values. If we take the smallest possible value on an estimated percentile and the largest value in the next percentile, the slice in the middle can be up to $n/2 + b = O(n)$. These $O(n)$ pointers are to be sent to a single processor, which will need $O(n)\mathbf{C}$ time to receive the elements and $O(n \log n)\mathbf{I}$ time to sort them. This is the WAT case of this algorithm: worse than sequential sorting.

6 Simulation Results

The implementation of the proposed algorithm is not concluded yet. However, we performed experiments to validate the most contrived assumptions used in our work.

The first experiment shows that, for a typical text file[6], the distribution of words inside each processor approximately follows that of the whole text, and therefore Steps 2-3 will work well on average text, as well as the simpler algorithm.

The second experiment is related to Step 4. The goal is to find a suitable pruned suffix size ℓ, so that the processors are able to sort locally without normally asking more characters of remote suffixes.

[6] In our experiments, the 262,755,189 bytes Wall Street Journal file from TIPSTER/TREC collection [6].

6.1 Word Distribution

In this simulation, the Wall Street Journal (WSJ) file is broken into $r = 16$ blocks of (almost) identical sizes b. For each block, we computed $r-1$ local percentiles. Next, these percentiles are made available to every simulated processor which computes $r-1$ medians, each one corresponding to a percentile. Table 1 presents the average and standard deviation of the slice sizes exchanged between any pair of processors. Suffixes were pruned at 48 characters (recall that a large ℓ can be used for Step 2).

p	sent	received	p	sent	received
	β ± stdev	β ± stdev		β ± stdev	β ± stdev
1	0.94 ± 2.32%	1.01 ± 7.68%	9	0.98 ± 2.40%	1.01 ± 3.25%
2	0.96 ± 2.39%	1.00 ± 2.78%	10	1.01 ± 1.95%	1.00 ± 3.01%
3	0.97 ± 1.46%	1.00 ± 3.55%	11	1.00 ± 1.95%	0.99 ± 2.99%
4	0.99 ± 1.12%	1.00 ± 2.89%	12	1.03 ± 1.73%	1.01 ± 3.03%
5	0.97 ± 2.01%	1.01 ± 3.04%	13	1.00 ± 1.17%	1.00 ± 2.98%
6	1.00 ± 1.02%	0.99 ± 3.24%	14	1.03 ± 2.14%	1.00 ± 2.98%
7	1.00 ± 1.64%	1.01 ± 3.21%	15	1.03 ± 1.67%	1.00 ± 3.46%
8	1.03 ± 1.27%	0.99 ± 3.10%	16	1.07 ± 1.34%	1.00 ± 3.83%

Table 1. Amount of exchanged data in bytes for $r = 16$ (β is the ratio between the average and the expected b/r). Standard deviation is presented as a percentage of the average.

We remark that the amount of messages sent and received for each pair is approximately the same in most cases. This shows that the partition in words is quite even among processors, and that each processor ends up with a slice of size almost b to perform Step 4.

With regard to the number of bytes transferred during redistribution we observe that the variation among the slices sent by a given processor is rather low ($< 2.5\%$). On the other hand, the variation of the number of bytes received by a given processor from each other processor is higher ($< 8\%$: the higher variation is 7.68% followed by a 3.83%). The largest slice transferred in the whole process is 11% over the expected b/r. This shows that the redistribution of slices is $O(b)\mathbf{C}$ in practice, even with pruned suffixes (of length 48 in this case).

Finally, since we are using estimated percentiles, this shows that the simpler algorithm of Section 5 performs well on natural language texts.

6.2 Suffixes Comparison

In this experiment, we generated sequentially the suffix array for a 100 megabytes subset of the WSJ. We computed for each suffix the number L of identical characters when compared with the previous suffix (given by 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
```

The purpose of this experiment is to find an ℓ which will work reasonably well even in the final moments of the sorting process, when the algorithm compares suffixes that are almost neighbors in the final suffix array. If each word in the text is considered an index point, we find that the average L is 15.04 with a standard deviation of 10.19. If we consider instead that suffixes do not start with *stop words* (e.g., `"a"`, `"the"`, etc), the average L is 15.45 with a standard deviation of 10.62. The distribution of L is given in the Figure 3.

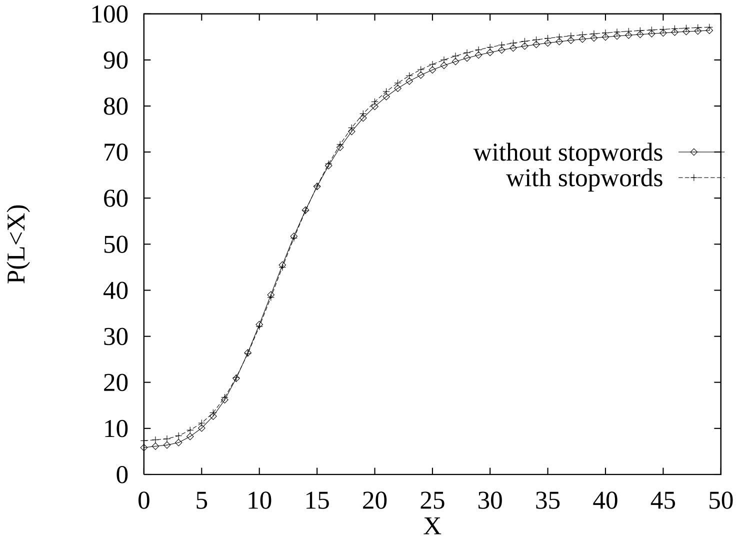

Fig. 3. Distribution of L (probability of a given $L < X$) for 100 megabytes of the WSJ file.

For both cases (suffixes starting and not starting with stop words), the distributions are similar. With $\ell = 30$, 90% of the comparisons are locally solved (i.e., the pruned suffixes differ). In our algorithm, the graph presented in Figure 3 can be considered an upper bound, as explained. Since $L \approx 15$ on average, by using $\ell = 30$ we can save 50% of communication and storage costs by compressing the slices to redistribute (cf. Section 3.4) [7]. For larger texts, the value of ℓ will grow if the same probability of remote access is to be maintained. However, the growing rate is known to be very low (i.e., $O(\log n)$, the average height of a leaf in the suffix trie [19]). The average L will grow at a similar rate, what allows to keep the same compression ratio.

7 Conclusions and Future Work

We have discussed a distributed algorithm for the generation of suffix arrays for large texts. The algorithm is executed on a parallel computer composed of processors connected through a high bandwidth network. The aggregate memory of the various processors is used as a giant cache for disks. In such

[7] Recent and more realistic simulations of the sorting process show that this upper bound is pessimistic. With $\ell = 20$ we have 90% of the comparisons decided locally (and therefore compression is 75% effective), and for $\ell = 30$ the probability of a successful local comparison is 95%.

aggregate memory, remote accesses are as time consuming as sequential accesses to a local disk. The algorithm quickly splits the problem in one independent subproblem per processor and the rest proceeds locally. The improvement in performance comes from parallelism and from the fact that remote memory can be accessed randomly at uniform cost.

We analyzed the average and worst (on average text) case complexity of our algorithm considering a text of size n and the presence of r processors storing b index points each. Such analysis points out many important advantages over previous work. First, our proposed algorithm has average running time complexity $O(b \log n)$ for computation and $O(b)$ for communication on average, which has optimal speedup over sequential algorithms. Second, it is faster than the previous parallel algorithms that solve this problem. Third, it scales up much nicer than other previous algorithms (e.g., one based on mergesort).

We are currently working on the implementation of the above parallel algorithms. The mergesort implementation is concluded [11] and we compared its performance with that of a local implementation of the sequential algorithm. Besides such implementation efforts, we are investigating the application of our ideas to the generation of the more popular inverted lists [17].

Acknowledgments

We thank the anonymous referees for their useful comments to improve this work.

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. 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.
6. D. Harman. Overview of the third text retrieval conference. In *Proceedings of the Third Text Retrieval Conference - TREC-3*, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology. NIST Special Publication 500-225.
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. Karkkainen. Suffix cactus: A cross between suffix tree and suffix array. In *Proc. CPM'95*, pages 191–204. Springer-Verlag, 1995. LNCS 937.
10. J. P. Kitajima, B. Ribeiro, and N. Ziviani. Network and memory analysis in distributed parallel generation of PAT arrays. In *Fourteenth Brazilian Symposium on Computer Architecture*, pages 192–202, Recife, August 1996.
11. J.P. Kitajima, M.D. Resende, B. Ribeiro, and N. Ziviani. Distributed parallel generation of indices for very large text databases. Technical Report 008/97, Universidade Federal de Minas Gerais - Departamento de

Ciência da Computação, Belo Horizonte, Brazil, April 1997. ftp://ftp.dcc.ufmg.br/pub/research/-nivio/papers/.
12. Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching.* Addison Wesley, 1973.
13. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22, 1993.
14. D.R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric. *JACM*, 15(4):514–534, October 1968.
15. M. J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, second edition, 1994.
16. 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. ftp://ftp.dcc.ufmg.br/pub/research/nivio/papers/.
17. B. Ribeiro, J.P. Kitajima, G. Navarro, and N. Ziviani. Parallel generation of inverted lists on a network of workstations. Technical Report 009/97, Universidade Federal de Minas Gerais - Departamento de Ciência da Computação, Belo Horizonte, Brazil, April 1997. ftp://ftp.dcc.ufmg.br/pub/research/-nivio/papers/.
18. B. Ribeiro, G. Navarro, J. P. Kitajima, and N. Ziviani. Recursive parallel generation of suffix arrays. Technical Report 010/97, Universidade Federal de Minas Gerais - Departamento de Ciência da Computação, Belo Horizonte, Brazil, April 1997. ftp://ftp.dcc.ufmg.br/pub/research/nivio/papers/.
19. W. Szpankowski. Probabilistic analysis of generalized suffix trees. In *Proc. CPM'92*, pages 1–14. Springer-Verlag, April 1992. LNCS 644.
20. E. Ukkonen. Constructing suffix trees on-line in linear time. *Algorithmica*, 14(3):249–260, Sep 1995.

Appendix A: Analysis of Pairwise Exchange

We show that the maximum amount of data exchanged by a pair of processors in the Step 3 of our algorithm is $O(b/r)$ on average.

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

The amount of pointers exchanged between $\lfloor r/2 \rfloor$ pairs of processors can be seen as r independent random variables with the same Binomial distribution (since the pairs exchange data in both ways). The maximum amount of data exchanged in a stage corresponds therefore to the maximum of r independent random variables with distribution $B(b, 1/r)$. Let $X_1, ..., X_r$ be those random variables.

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

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

and we observe that

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

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

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

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

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

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

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

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

(an $O(\log r)$ error term is discarded assuming $r < b$).
We substitute now $k = \alpha b/r$, for constant α, in the above equation and simplify, to obtain

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

which by expanding logarithms yields

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

which is clearly achieved by some constant α.

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

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

what completes the proof.

Appendix B: Average Median by Sampling b Out of n

We prove that by sampling b elements out of n we arrive at the correct median with a relative error of $O(n^{-1/2})$, provided $b > \sqrt{n}$. Since the median is the highest variance percentile, the proof is automatically valid for any percentile. This is stronger than the result we need, since we show that even the median estimation at a single processor is good enough, and therefore doing the same at r processors and combining the results (as done in the paper) is better.

For simplicity, we assume that $b = 2m+1$. The probability $s(j)$ of our estimated median being the position j in the sorted array is that of, in our sampling, selecting m elements in the range $[1..j-1]$, m elements in the range $[j+1..n]$, and of course selecting j. This is

$$s(j) = \frac{\binom{j-1}{m}\binom{n-j}{m}}{\binom{n}{2m+1}}$$

We are interested in the expected proportional size of the larger partition. This is

$$P \;=\; \frac{1}{n}\left(\sum_{j=1}^{n/2}(n-j+1)s(j) \;+\; \sum_{j=n/2+1}^{n} js(j)\right) \;=\; 2\sum_{j=n/2+1}^{n}(j/n)s(j)$$

We call $t(j) = js(j)$ and $f(x) = t(nx)$ the continuous version of $t(j)$ over the interval $[1/2 \,..\, 1]$. Hence

$$P \;=\; \frac{2}{n}\sum_{j=n/2+1}^{n} t(j) \;=\; \frac{2}{n}\sum_{j=n/2+1}^{n} f(j/n) \;\le\; 2\int_{1/2}^{1-m/n} f(x)dx$$

(since $t(j)$ is descending for large n). Since $f(x) = t(nx)$, it follows that

$$\frac{f(x+1/n)}{f(x)} = \frac{t(nx+1)}{t(nx)} = v(nx)$$

where we have just defined

$$v(j) = \frac{t(j+1)}{t(j)} = \frac{(j+1)(n-m-j)}{(j-m)(n-j)}$$

Taking logarithms and multiplying by n, we have

$$\frac{\ln f(x+1/n) - \ln f(x)}{1/n} = n \ln v(nx)$$

This last equation defines $(\ln f)'$, hence

$$f(x) = K\ e^{n\int_{1/2}^{x} \ln v(ny)dy}$$

the constant K coming from the integration. We obtain it observing that $f(1/2) = K = t(n/2)$, from where

$$f(x) = \sqrt{\frac{m}{\pi}}\ e^{n\int_{1/2}^{x} \ln v(ny)dy}(1 + O(m/n) + O(1/m))$$

We now solve the integral of $\ln v(ny)$. We have

$$n\int_{1/2}^{x} \ln v(ny)dy \;=\; \int_{n/2}^{nx} \ln v(z)dz$$

$$\le\; m(2\ln 2 + \ln x + \ln(1-x)) + \ln 2 + \ln x + O(1/n)$$

We then rewrite the equation for $f(x)$ as follows

$$f(x) = \sqrt{\frac{m}{\pi}}\, 2^{2m+1} x^{m+1} (1-x)^m (1 + O(m/n) + O(1/m))$$

and return to our wanted result on P

$$P \leq 2 \int_{1/2}^{1-m/n} f(x)dx \quad \leq \quad \frac{4^{m+1}\sqrt{m}}{\sqrt{\pi}} \int_{1/2}^{1} x^{m+1}(1-x)^m dx\ (1 + O(m/n) + O(1/m))$$

This last integral is not trivial. We solve it by induction. Let

$$h(d) = \int_{1/2}^{1} x^{m+1+d}(1-x)^{m-d} dx$$

then

$$h(m) = \frac{1 - \frac{1}{4^{m+1}}}{2m+2}$$

and our desired result is $h(0)$. Using $fg = \int f'g \ + \ \int fg'$, we have

$$h(d) = \int_{1/2}^{1} x^{m+1+d}(1-x)^{m-d} dx = \frac{1}{(m-d+1)4^{m+1}} + \frac{m+d+1}{m-d+1} \int_{1/2}^{1} x^{m+d}(1-x)^{m-d+1} dx$$

where the last integral is $h(d-1)$, hence the recurrence. By using $g(i) = h(m-i)$ we have the more conventional one

$$g(0) = \frac{1 - \frac{1}{4^{m+1}}}{2m+2} \quad , \quad g(i+1) = \frac{(i+1)g(i) - \frac{1}{4^{m+1}}}{2m-i+1}$$

which yields

$$g(m) = \frac{1}{8}\frac{\sqrt{\pi/m}}{4^m}\ (1 + O(1/m))$$

and we have the final result

$$P \leq \frac{1}{2}\ (1 + O(m/n) + O(1/m)) = \frac{1}{2}\ (1 + O(b/n) + O(1/b))$$

what shows that the estimated median is very close to the real one for moderately large b.

A question that naturally arises is why the result seems to be worse as b grows (i.e., the $O(b/n)$ error term). This is because we used upper bounds in some parts, hiding factors depending on m that made the error smaller. Since it is clear that, as b grows, the estimation gets better, and that we can assume $b > r$ (i.e., $b > \sqrt{n}$), we have an estimation error independent of b

$$P \leq \frac{1}{2}\ \left(1 + O\left(1/\sqrt{n}\right)\right)$$

This proves that the largest piece of the partition is $O(1/\sqrt{n})$ in excess over the average, for the median and for every percentile, even if only one processor samples the data.

This article was processed using the LaTeX macro package with LLNCS style