A Fast Distributed Suffix Array Generation Algorithm*

João Paulo Kitajima
Bioinformatics Laboratory
Campinas State University (UNICAMP)
13083-970 Campinas, SP, Brazil
jpk@lbi.dcc.unicamp.br

Gonzalo Navarro
Dept. of Computer Science
University of Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract

We present a distributed algorithm for suffix array generation, based on the sequential algorithm of Manber and Myers. The sequential algorithm is $O(n \log n)$ in the worst case and $O(n \log \log n)$ on average, where n is the text size. Using p processors connected through a high bandwidth network, we obtain $O((n/p) \log \log n)$ average time, which is an almost optimal speedup. Unlike previous algorithms, the text is not transmitted through the network and hence the messages exchanged are much smaller. We present some experimental evidence to show that the new algorithm can be faster than the sequential Manber & Myers counterpart.

1. Introduction

To reduce the cost of searching in textual databases, specialized indexing structures are adopted [2, 16]. The most popular of these are the *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.

Suffix arrays [10] or PAT *arrays* [3] 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 [3, 10] or in applications where the concept of word is of no use, such as computational biology [4]. In this model, the entire text is viewed as one very long string. In this string, each position k is associated to a *suffix*, which initiates at position k. 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, in documents, to index only word beginnings). These index pointers are sorted according to a *lexicographical ordering* of their respective suffixes. To find patterns in the text, binary search is performed on the array at $O(\log n)$ cost (where n is the text size).

The simplest approach to build the suffix array sequentially is to perform a traditional sort of the pointers, such as mergesort or quicksort. However, there exist specialized algorithms for sequential construction of suffix arrays, such as the original one of Manber and Myers [10] and, more recently, those of Sadakane [13]. As shown in [13], these specialized algorithms outperform the general purpose sorting techniques.

If the text is very large, we have to resort to secondary memory and the problem is more difficult. The best known sequential procedure for generating large suffix arrays [3] takes time $O(n^2 \log(m)/m)$ where m is the size of the main memory.

We present a new algorithm for distributed parallel generation of large suffix arrays in the context of a high bandwidth network of processors. This is the first distributed algorithm which generalizes a special purpose sequential algorithm (i.e. that of Manber and Myers [10]), instead of the slower general purpose algorithms considered in previous work.

Our parallelism model is that of a parallel machine with distributed memory. Our algorithm sorts in a (distributed) main memory, without secondary storage considerations. Assume that we have a number p of

*This project has been supported in part by CYTED VII.13 AMYRI Project. First author supported by FAPESP fellowship 99/01389-0. Second author partially supported by Fondecyt grant 1-990627.

Algorithm	Average case	Worst case
Msort	$n(\mathbf{I}+\mathbf{C})$	$n^2(\mathbf{I}+\mathbf{C})$
Qsort	$(b\log n)\mathbf{I}+b(\log p)\mathbf{C}$	$n^2(\mathbf{I}+\mathbf{C})$
G-Qsort	$(b\log n)\mathbf{I}+b\mathbf{C}$	$n^2(\mathbf{I}+\mathbf{C})$
MMsort	$(b\log\log n)(\mathbf{I}+\mathbf{C})$	$\log(n)(b\mathbf{I}+n\mathbf{C})$

Table 1. Complexity comparison among our algorithm (MMsort) and previous work.

processors, each one storing b text positions, composing a total distributed text of size $n = pb$. Our final suffix array will be left distributed (so as to reduce query time overhead later, or it can be simply merged otherwise). We assume that the parallelism is coarse-grained, with a few processors, each one with a large main memory. Typical values are p in the tenths or hundreds and b in the millions. We assume at least that $p << \sqrt{n} << b$.

In our communication model, each pair of processors can exchange messages simultaneously without contention, and broadcasting can be done efficiently. This setup corresponds to current fast switching technology with a few large processors [5] such as ATM (Asynchronous Transfer Mode) networks, the IBM SP (based on the High Performance Switch, HPS), or a Myrinet switch cluster.

Our new algorithm has a computation and communication complexity of $O(b\log\log n)$ on average. Previous distributed algorithms for this problem are generalizations of general purpose sorting algorithms adapted to suffix arrays: mergesort (Msort [8]) and of quicksort (Qsort [7] and G-Qsort [11]) have been used. Their complexities are in Table 1, where $\mathbf{I}$ refers to CPU cost and $\mathbf{C}$ to communication cost (normally the most important). Note that none of the algorithms handle well the worst case, but our new algorithm (MMsort) is more reasonable. For the average case, only G-Qsort compares favorably against our new MMsort.

An interesting feature of the new algorithm with respect to the previous ones is that the text is not transmitted across the network, only requests for data and for updates are sent. Finally, we point out that there is some related work on sorting strings under the PRAM model [1, 6], where there is no difficulty in accessing the data of other processors.

2. The Sequential Algorithm

We describe briefly in this section the sequential algorithm of [10]. The general idea exploits the fact that suffixes are part of other suffixes of the same text. First, the array is bucket-sorted by the first letter of each suffix, in linear time. After this, $\lceil\log_2 n\rceil$ iterations are carried out, numbered $h = 0, 1, ...\lceil\log_2 n\rceil - 1$. At iteration h, the suffix array is already sorted by the first 2^h letters of each suffix, and at the end of the iteration the suffix array is sorted by the first 2^{h+1} first letters. Each such iteration is done in linear time using bucket sort again: the buckets of the h-th iteration are refined to obtain those of the $(h+1)$-th iteration.

The key idea follows. Assume that text positions i and j are currently in the same bucket, i.e. they are equal in their first 2^h characters. In order to sort the text positions i and j by their first 2^{h+1} characters, we just need to know which are the buckets where the positions $i+2^h$ and $j+2^h$ are at this point. Since positions i and j are in the same bucket, then $text[i..i+2^h-1] = text[j..j+2^h-1]$. If we want to sort them by 2^{h+1} characters, we need to compare $text[i..i+2^{h+1}-1]$ against $text[j..j+2^{h+1}-1]$. The outcome of this comparison is exactly that of comparing $text[i+2^h..i+2^{h+1}-1]$ versus $text[j+2^h..j+2^{h+1}-1]$. Since these are two text positions ($i+2^h$ and $j+2^h$) compared by their first 2^h characters, the answer corresponds to the buckets where these two text positions are currently positioned, since all the text positions are currently sorted by 2^h characters.

Figure 1 illustrates this algorithm. We start with the unsorted array. In the first step we bucket sort using the first character. After this, we double the number of characters already sorted until we obtain n buckets. This happens in $\lceil\log_2 n\rceil$ steps but, as in the example, it can happen before.

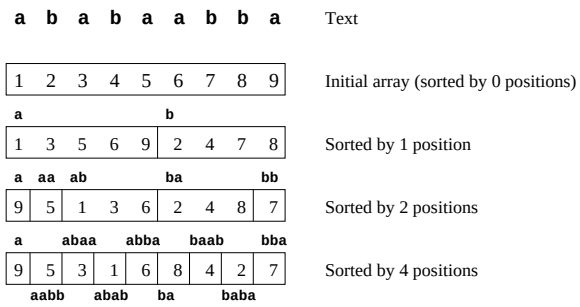

Figure 1. The sorting algorithm, at a high level.

The main problem when converting this simple idea into an efficient algorithm is to devise a way to do all the bucket sorts in linear time avoiding the explosive number of buckets that could be generated[1]. At

[1] A naïve bucket sort on 2^h characters for an alphabet Σ generates $|\Sigma|^{2^h}$ buckets, one per string of length 2^h, and most of them empty. When h approaches $\log_2 n$ the number of buckets

each iteration, we consider the current buckets in order. Think on the first one. The suffixes starting 2^h positions *before* those in the first bucket must become the first suffixes of *their* buckets. This is because all the suffixes of the same bucket are equal in their first 2^h letters, and we are moving to the beginning those suffixes followed by the smallest existing 2^h length strings. Once these are moved to the beginning of their buckets, we continue with the second buckets, and so on. The idea is, then, that instead of sorting positions i and j by searching the buckets of $i+2^h$ and $j+2^h$, we wait until, at some moment, the position $i+2^h$ (or $j+2^h$) is first found in the linear traversal, and then move the position i (or j) to the beginning of the bucket.

We consider the algorithm in more detail now. We first show the arrays used (all of size n) and then the steps performed.

Pos	: suffix array, sorted by the first 2^h letters
BH	: binary array marking the beginnings of the current buckets
Prm	: reverse Pos permutation being built for the next iteration
$B2H$	: BH mask being built for next iteration
$Count$	: sizes of the new buckets being created

A In the beginning, Pos is bucket-sorted by the first letter of the suffix pointed. This is easily done in linear time. BH is set to 1 at the first cell of each new bucket and to 0 elsewhere.

B Now, the following code is carried out for $h = 0$ to $\lceil \log_2 n \rceil - 1$

1. $B2H$ is initialized to zeros, as well as $Count$ (which needs initialization only where BH is 1).

2. Prm is set to the reverse permutation of Pos, except it points to the beginning of the bucket where the reverse permutation should point. That is, if $Pos[i] = j$, then $Prm[j]$ points to the beginning of the bucket of i. This is easily assigned on a linear pass over Pos, by recalling the position of the last 1 of BH and setting $Prm[Pos[i]]$ to that position (instead of to i).

3. Now, for each bucket of Pos, in order (their limits are signaled in BH), we do:

   (a) For each position i of the current bucket, do

      i. Let $d = Pos[i] - 2^h$ be the text pointer to move

      ii. Look at $e = Prm[d]$ to find its current bucket

      iii. Make $Prm[d]$ point to $e + Count[e]$, which is the next free position in the bucket that starts at e

      iv. Set $B2H$ to 1 in the new $Prm[d]$ position

      v. Increment $Count[e]$

   At this point, the reverse permutation Prm (not Pos) maps the proper suffixes to the beginning of their buckets. Doing this bucket by bucket will correctly complete the iteration. But we need first to leave $B2H$ in 1 only in the first position of each new bucket (currently there is a stream of 1's).

   (b) For each position i of the current bucket, do

      i. Let $d = Pos[i] - 2^h$ be the text pointer to move

      ii. If $B2H[Prm[d]]$ is 1, delete the other 1's to the right (until $B2H$ is 0 or BH is 1). This will only occur for the first element of the new bucket.

4. We have finished this iteration. Just restore a couple of invariants. Set Pos to be the reverse of Prm (which is the reverse permutation of the new iteration), and make $BH[i] = BH[i]$ *or* $B2H[i]$ for all i. If, for all i, $BH[i] = 1$, then the array is already sorted and we can preempt the whole algorithm.

To avoid complicating the description of the above algorithm, we have not considered that, at iteration h, the text positions $n - 2^{h+1} + 1$ to n are not referenced. In fact, since we assume that the text is followed by zeros, these positions must be the first in being moved to the beginning of their buckets, prior to starting the bucket-by-bucket traversal.

Figure 2 illustrates the more detailed algorithm, using the same example of Figure 1. Along the iteration for each h value we build a new sorted array (reflected in Prm, $Count$ and $B2H$). At the end of the iteration we (basically) copy $B2H$ onto BH and the reverse of Prm onto Pos (which is the "real" permutation).

We consider the analysis now. It is clear that each iteration of Step 2 takes $O(n)$ time, since the iterations on each bucket take time proportional to the bucket size, and their sizes add up n. The array is totally sorted after $h = \log_2 n$ iterations because the longest suffix is of length n, and hence sorting all the strings by their first $2^h = n$ letters is enough. The cost is therefore $O(n \log n)$ in the worst case. However, as pointed out in [10], the algorithm is $O(n \log \log n)$ on average. This

becomes $|\Sigma|^n$.

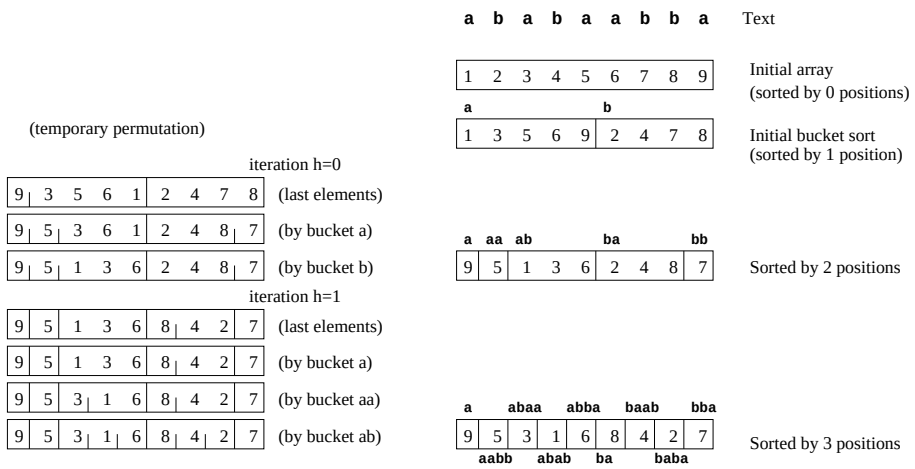

Figure 2. The sorting algorithm, at a detailed level. We have stopped when the array became totally sorted.

is because the average length of the longest repeated substring in the text (or equivalently, the height of the suffix tree) is $O(\log n)$ [14].

3. A Distributed Algorithm

For the distributed version, we assume that we have p processors, and each one stores a contiguous portion of size $b = n/p$ of the text. Each processor will also store a slice of the corresponding Pos, Prm, BH, and $Count$ arrays (and some extra arrays, as seen later).

The most delicate part of the parallelization is that the original algorithm strongly relies on sequential execution. We first explain two heavily used primitives and later depict our distributed algorithm.

Pairwise Exchange At some points, each processor has some data to exchange with each other processor. Assume that the total amount of data each processor has to send and to receive is $O(b)$. Ideally, each processor has on average $O(b/p)$ data to exchange with each other, but the general situation can be unbalanced.

Algorithms to pair all the processors with each other in $O(p)$ turns with synchronization barriers were presented in [15, 11]. If the data is uniformly distributed the average complexity of each turn is $O(b/p)$ [11], and the total average cost for pairwise exchange is $O(b)$. However, in the worst case an unbalanced transfer schedule yields $O(b)$ time per turn, for a total communication complexity of $O(n)$.

Batched List of Requests When a processor needs to perform some actions on the data stored in others, it sends them a request to do so. To avoid the overhead of many small messages, the requests are buffered. An array of p buffers, one per receiving processor, is maintained. The total cost of this primitive is $O(p + r)$, where r is the number of requests to send.

3.1. The Algorithm

We depict now the algorithm using the above explained primitives.

A The first step is to bucket sort the global suffix array. Each processor will end up storing a slice of the alphabet. For instance, the first one could contain all the suffixes starting with 'a', 'b', and some suffixes starting with 'c'. Any choice of which suffixes starting with 'c' are left in the first processor is suitable, as long as all the processors take the same decisions separately and consistently. We perform the following steps.

1. Each processor bucket-sorts its local text in Pos.
2. Each processor broadcasts how many suffixes it has starting with each letter.
3. Each processor, with the information gathered from the others, determines which letters should it administer and which processors have them. All processors use the same algorithm, so each one knows which part of Pos to send to each other.
4. The processes exchange pairwise the corresponding parts of Pos.
5. Each processor initializes its local section of BH.

B This part is the most complex, and hence we use a more powerful formalism. The algorithm to be carried out at each processor is depicted in Figure 3.

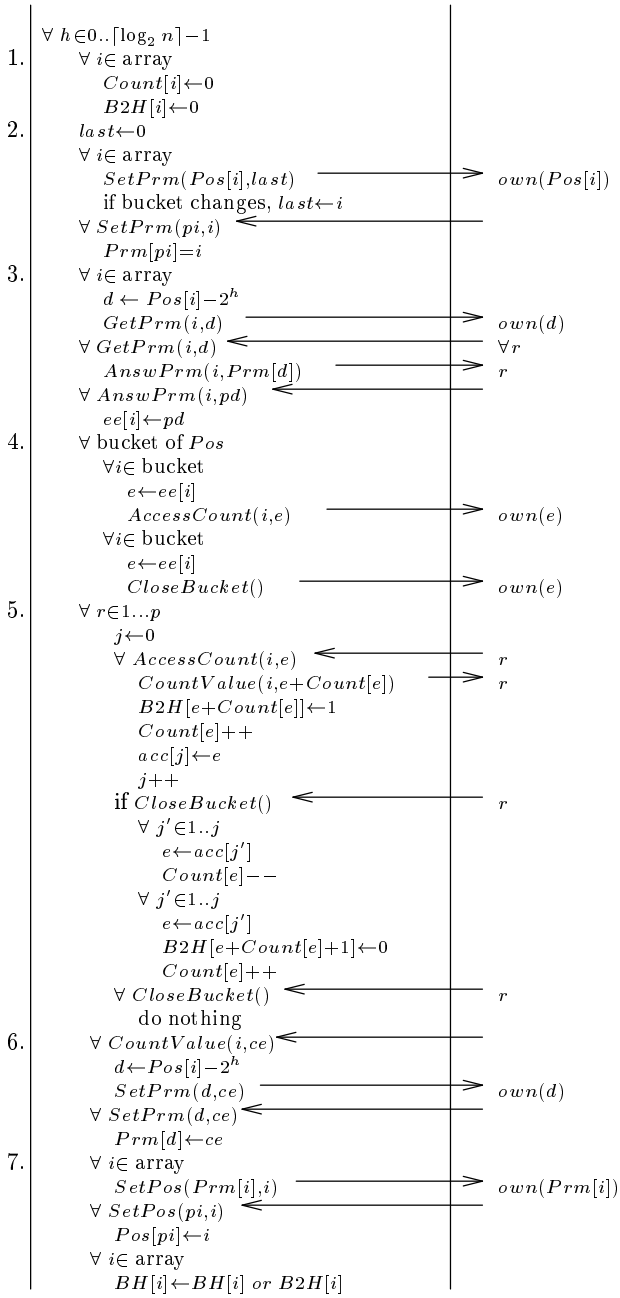

Figure 3. Part B of the distributed algorithm.

Left-to-right arrows represent messages emitted. The receiving processor is given at the end of the arrow, where $own(x)$ means the processor owning the array position x. Right-to-left arrows represent incoming messages, which are generally used to execute a loop body for each message arrived. Sometimes we indicate the originating processor, in order to put a condition on it, or just in order to use the processor number (we use $\forall r$ to indicate that there is no condition on the sending processor r). The rest is a quite traditional pseudocode.

It is important to note that all the messages are exchanged using the technique of pairwise exchange explained and sent in batch. Only after a loop is completed we send all the outgoing messages generated during the loop.

We explain now the rationale of the different parts, using the numbers on the left of the code. The following actions are carried out at each processor, for $h = 0$ to $\lceil \log_2 n \rceil - 1$

1. The local portions of $Count$ and $B2H$ are initialized to zero.

2. Prm is set to point to the beginning of the buckets where the reverse permutation should point. This corresponds to perform a linear pass over Pos, setting $Prm[Pos[i]] = last$, where $last$ is the position of the beginning of the last bucket before position i. However, the Prm positions to update may reside in other processors. Note that there is no sequentiality constraints in the order of honoring the $SetPrm()$ requests, because each position of Prm is updated only once.

3. We need to read Prm positions from other processors. This step is equivalent to B.3.a.i and B.3.a.ii of the sequential algorithm, i.e. to computing $d = Pos[i] - 2^h$ and then $e = Prm[d]$ for all the positions of the local Pos array. $ee[]$ is an extra array used to store the e values that the sequential algorithm computes on the fly.

Steps 4 to 6 correspond to B.3.a.iii to B.3.a.v and B.3.b of the sequential algorithm. In order to parallelize it we make a separate pass over the buckets joining all these steps.

4. We traverse Pos linearly, bucket by bucket. For each bucket we send all the $AccessCount()$ requests to update $Count$ and $B2H$, and to retrieve the resulting $Count$ values. After each bucket we need to signal the bucket termination to all the processes we have sent some message for this bucket. We could send only one $CloseBucket()$ message to each such processor, but this cannot be efficiently done when there are much less than p elements in the bucket (which is a normal case). So we do it as in the sequential algorithm (B.3.b): if a processor received x $AccessCount()$ messages, it will receive x $CloseBucket()$ messages.

5. We now serve the requests sent in the previous step. It is crucial to do it ordered by the sending processor (the requests sent by each processor are internally ordered). The list of requests received from each processor has blocks of $AccessCount()$ requests followed by blocks (of the same length) of $CloseBucket()$ requests. The $AccessCount()$ requests are used to send back to the processor the current $e + Count[e]$ value, and then to put 1 in $B2H$ at this position[2] (step B.3.a.iv of sequential algorithm) and increment $Count[e]$ (step B.3.a.v of sequential algorithm). We also store in an extra array $acc[]$ the e values received[3], for purposes that are made clear shortly.

   After the block of $AccessCount()$ is served we have the block of $CloseBucket()$. Only the first such message of each block is considered: we review (using $acc[]$) the set of e values in the last $AccessCount()$ block and use this to transform the stream of 1's of $B2H$ so as to leave only the first of these 1's. This is done in a way somewhat different from the sequential algorithm (step B.3.b), because we have not yet updated Prm.

6. We use the information received in $CountValue()$ about the values of $e + Count[e]$, in order to set $Prm[d] = e + Count[e]$ (step B.3.a.iii of the sequential algorithm). Since the d position is in general in another processor we need to exchange requests.

7. Set Pos to be the reverse of Prm, using a process similar to that used to set Prm in terms of Pos. Later, make $BH = BH$ *or* $B2H$. This corresponds to step B.4 of the sequential algorithm.

Again, we have not complicated the description with the initial pass that, at each iteration, must be made to update the final positions of the text.

4. Analysis

We analyze now the computation and communication complexity of our algorithm. We present the complexities using factors **I** (for internal computation time) and **C** (for communication time). To simplify some measures, we make the assumption $p \leq b$.

We divide the analysis in the same parts A and B of the description of the algorithm.

[2]Note that the BH position to access could surpass the scope of the current processor, and a request to the neighboring one could be necessary for this matter.

[3]Note that $ee[]$ and $acc[]$ can share their memory.

A The global bucket sort by the first letters has three steps. Step 1 has $O(b)$ computation complexity and no communication. Step 2 has $O(|\Sigma|p)$ communication complexity, where Σ is the alphabet of the text and is considered of constant size. Step 3 involves only computation and can be easily done in $O(|\Sigma|+p)$ time. Step 5 is clearly $O(b)$ computation time. The most interesting step is the 4th, a pairwise exchange. As explained in Section 3, this costs $O(b)$ communication on average, but the worst case is $O(n)$. Hence, the total complexity of Part A is $O(b+p)\mathbf{I} + O(b+p)\mathbf{C} = O(b)(\mathbf{I}+\mathbf{C})$ on average, and $O(b)\mathbf{I}+O(n)\mathbf{C}$ in the worst case.

B The second part is repeated $\log_2 n$ times at most, but only $O(\log\log n)$ times on average (see the end of Section 2). The step-by-step analysis is simple, once we note that most of them consist in pairwise exchanges (see Section 3) which in all cases are uniformly distributed. Step 1 is $O(b)\mathbf{I}$. Steps 2 to 7 are all $O(b)\mathbf{I} + O(b)\mathbf{C}$ on average and $O(b)\mathbf{I} + O(n)\mathbf{C}$ in the worst case. The most interesting step is perhaps the 5th, because of the multiple traversals on the same requests, but these impose in any case a constant factor overhead that does not change the complexity. Hence, Part B is $O(b\log\log n)(\mathbf{I} + \mathbf{C})$ on average and $O(b\log n)\mathbf{I} + O(n\log n)\mathbf{C}$ at worst.

The result is that our algorithm has an average complexity of $O(b\log\log n)$ both in computation and communication, while the worst case is not better than that of the sequential algorithm. Notice that, if $\sqrt{n} \leq b \leq n$, then $\log\log n = \Theta(\log\log b)$. We consider scalability now. If we double n and p, the new cost $T(2n, 2p)$ (both in **I** and **C** complexities) becomes $b\log\log(2n) = T(n,p) \times (1 + O(1/(\log n \log\log n)))$. The ideal scalability condition is $T(2n, 2p) = T(n, p)$.

Table 1 presents a comparison between our complexity (MMsort) and that of previous algorithms (Msort [8] based on mergesort and Qsort [7] and G-Qsort [11] based on quicksort), when all the suffixes are indexed.

As explained in Section 1, previous algorithms did not handle properly the worst case. Our algorithm handles this case much better. On average, we have the best computation complexity (inheriting from the sequential algorithm), but our communication complexity is $O(b\log\log n)$ instead of $O(b)$ of G-Qsort.

5. Experimental Analysis

5.1. Sequential Generation

A suffix array is a vector of addresses, ordered by the value of the addressed entity (and not by the address itself). Therefore, a naïve implementation of a suffix array builder may use a standard sort algorithm, implemented, for example, as a built-in library sort routine like C `qsort`. However, these classical algorithms do not assume any content interdependence among the elements to be sorted. This drawback has already been proven experimentally relevant, comparing suffix arrays generators implemented using standard sort procedures against programs using smart techniques like those based on Manber & Myers [10] and Sadakane [13] algorithms.

We compared once more the execution times of two sequential and in-core suffix array generators, one based on an efficient version of the QuickSort algorithm (`qsort`) [9] and one based on the Manber & Myers strategy. Both programs were executed on an Intel Linux workstation with half gigabyte of main memory and a 400 MHz processor. The input text is a random-generated genome. We measured total elapsed time on a quiet machine, starting with 1 megabyte of input text. With a 30 megabyte genome, the Manber & Myers program generated a suffix array in 85% of the elapsed time of the standard `qsort` based implementation. Our measures also showed the $O(n \log \log n)$ behavior of the Manber & Myers algorithm and the $O(n \log n)$ complexity for `qsort`. For increasing input text sizes, the gap between execution times of both programs will keep improving. A collateral conclusion was derived from this first comparative study: execution times may be important even for suffix arrays being generated completely in gigabytes primary memory (we ran a suffix array generator based on quicksort on a genome with 25 megabases and it took almost 2 hours).

5.2. Parallel Generation

We implemented a very simplified prototype of our parallel algorithm in order to perform some comparative measures. We used an interpreted programming language, Perl, running on a 2 processor shared memory workstation. Communication is performed through message passing using sockets. Although much less efficient than compiled programs, Perl scripts are very suitable for string processing and for the development of Web active pages. Like Java, another interpreted language, Perl also has been used currently for the development of parallel applications. The goal of the shared memory here is to simulate a very fast network.

The first phase of the parallel algorithm is compared against the first phase of the sequential algorithm (Table 2).

input size	// comp. time	// bucket	// comm. time	seq. phase 1
8 kb	0.15	0.08	0.12	0.13
16 kb	0.29	0.14	0.19	0.33
32 kb	0.62	0.33	0.31	0.64
64 kb	1.21	0.66	0.62	1.28
128 kb	2.35	1.32	1.39	2.57
256 kb	4.86	2.62	2.78	5.24
512 kb	9.70	5.26	5.67	10.34
1024 kb	19.33	10.45	11.77	20.67

Table 2. Execution and communication times of Phase 1. Time unit is second.

The measures presented in Table 2 show that the prototype program should improve its efficiency in two directions: (1) reduce the additional computation time, not related to the bucketsort and (2) reduce the communication time.

The additional computation time not related to the bucketsort is the computation time related to housekeeping of arrays and counters, much of which could be simplified. When we compare the sequential phase 1 with only the bucketsort of the parallel version, we verify an ideal speedup of 2, showing that the parallelism is being exploited.

The reduction of the communication time is related to the use of parallel communication and an efficient communication network. Even using shared memory, as in the case of our prototype, the use of TCP/IP sockets generates a high overhead. In our prototype, communication is not parallel and the measures presented in Table 2 can be reduced by 2.

The second phase of our algorithm is much more complex and the corresponding prototype program presented very low efficiency due mainly to the use of sockets. Table 3 presents the obtained measures.

We tried to understand in more details this low efficiency. We measured only phases B1, B2, and B3 total elapsed and communication times for different input sizes and verified a communication overhead superior to 50% of the total execution time. For steps B4 to B7, communication corresponded to around 40% of the total execution times of these steps. This communication overhead was also detected in phase I and even when communicating through a shared memory, the use of a protocol based on sockets is not the best choice (be-

input size	// total time	// comm. time	seq. phase 2
8 kb	1.15	0.70	0.08
16 kb	3.05	1.66	0.16
32 kb	8.12	3.67	0.34
64 kb	23.49	8.75	0.67
128 kb	75.36	22.65	1.37

Table 3. Execution and communication times of first iteration of Phase 2. Time unit is second.

sides the need of a switch to exploit communication parallelism). Also, like in phase I, this prototype does not use threads: there is no parallelism between send and receive (communication times should be reduced by a factor of 2).

6. Conclusions

We have presented and analyzed a new distributed suffix array generation algorithm. It is based on the Manber & Myers sequential algorithm [10], unlike previous work [8, 7, 11] which are based on general purpose sequential sorting algorithms. Some unique features of the new algorithm are its low CPU cost and the fact that the text is never transmitted through the network (and hence the messages are much shorter in practice).

The presented algorithm can be adapted to index words on natural language texts. The only change is the Step A of the algorithm, which must be replaced by the computation of the whole vocabulary and its frequencies. Such a distributed algorithm has already been presented in [12], and its cost is close to $O(b)\mathbf{I} + O(\sqrt{n})\mathbf{C}$, which is negligible compared to that of Part B. We have omitted it here for lack of space.

References

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

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

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

[4] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computaional Biology*. Cambridge University Press, 1997.

[5] J. Hennessy and D. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[6] J. Jájá, K. W. Ryu, and U. Vishkin. Sorting strings and constructing digital search trees in parallel. *Theor. Comp. Sci.*, 154(2):225–245, 1996.

[7] J. Kitajima, G. Navarro, B. Ribeiro, and N. Ziviani. Distributed generation of suffix arrays: a quicksort-based approach. In *Proc. WSP'97*, pages 53–69. Carleton University Press, 1997.

[8] J. Kitajima, B. Ribeiro, and N. Ziviani. Network and memory analysis in distributed parallel generation of PAT arrays. In *Proc. 8th Brazilian Symp. on Comp. Arch. - High-Performance Processing*, pages 193–202. Brazilian Comp. Soc., 1996.

[9] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[10] U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, Oct. 1993.

[11] G. Navarro, J. Kitajima, B. Ribeiro, and N. Ziviani. Distributed generation of suffix arrays. In *Proc. CPM'97*, LNCS 1264, pages 102–115, 1997.

[12] B. Ribeiro, J. Kitajima, G. Navarro, C. Sant'Ana, and N. Ziviani. Parallel generation of inverted lists for distributed text collections. In *Proc. SCCC'98*, pages 149–157. IEEE CS Press, 1998.

[13] K. Sadakane. A fast algorithm for making suffix arrays and for burrows-wheeler transformation. In *Proc. DCC'98*, pages 129–138, 1998.

[14] W. Szpankowski. Probabilistic analysis of generalized suffix trees. In *Proc. CPM'92*, pages 1–14, 1992. LNCS 644.

[15] T. Tabe, J. Hardwick, and Q. Stout. Statistical analysis of communication time on the IBM SP2. *Comp. Sci. and Statistics*, 27:347–351, 1995.

[16] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York, 1994.