Massively Parallel Suffix Array Construction

Costas S. Iliopoulos[1,2] and Maureen Korda[3]

[1] King's College London, Dept. of Computer Science, London WC2R 2LS, England, csi@dcs.kcl.ac.uk,
WWW home page: http://www.dcs.kcl.ac.uk/staff/csi
[2] Curtin University of Technology, School of Computing, GPO Box 1987 U, WA, Australia.
[3] King's College London, Dept. of Computer Science, London WC2R 2LS, England, mo@dcs.kcl.ac.uk,
WWW home page: http://www.dcs.kcl.ac.uk/pg/mo

Abstract. This paper considers the construction of the suffix array of a string on the MasPar MP-2 architecture. Suffix arrays are space-efficient variants of the suffix trees, a fundamental dictionary data structure that is the backbone of many string algorithms for pattern matching and textual information retreival. We adapt known PRAM techniques for implementation on the MasPar: bulletin boards, doubling techniques and sorting methods. Performance results are presented.

1 Introduction

Given a text $\mathcal{T}$, the retrieval of statistical information about $\mathcal{T}$ can be accomplished using textual search techniques. One such technique is to firstly compute an index for $\mathcal{T}$ that can subsequently be used to answer a variety of queries. This index must be efficient in terms of construction time, memory use and storage space. In the field of string algorithms, the suffix tree data structure ([16], [11]) is an elegant data structure that provides an index and a statistical resource for a given input text: for a text $\mathcal{T}$ of length n, a suffix tree of $\mathcal{T}$ is a compacted trie of all the (unique) suffixes of $\mathcal{T}$. It is well known that a suffix tree can be constructed in linear sequential time and space. However, the constant hidden by the $O(n)$ space requirement is sufficient to render this data structure impractical in many real applications. Consequently, recent algorithms have been devised that consider the practical implications of suffix trees where space is traded for query time ([3],[2]).

An alternative data structure that also provides an index for a given text is the suffix array [12]. The suffix array of a text $\mathcal{T}$ of length n is an array of size n such that the i-th entry is the i-th smallest suffix according to the lexicographical ordering of strings. The suffix array therefore represents the leaves of the *ordered* suffix tree for the text $\mathcal{T}$. Two additional arrays (or a further $2n - 4$ values) associated with longest common prefix information are required to guide searching in the suffix array. This information represents the inner (branching) nodes in the equivalent suffix tree. Although the suffix array requires the longer time of $O(n \log n)$ to construct, it's redeeming feature is that it requires less than half the space of the equivalent suffix tree. The reduction in storage space implies that querying the suffix array is also very efficient since fewer (external) memory references are required.

In parallel computation, a PRAM algorithm was presented in [1] for the construction of a suffix tree requiring $O(n \log n)$ work and $O(\log n)$ time. This algorithm also requires polynomial space; i.e. $n^{1+\epsilon}$ for $0 < \epsilon < 1$. Drawing from the body of fundamental tools developed for parallel string algorithms, we present a practical PRAM algorithm for suffix array construction. By definition, the construction of the suffix array requires sorting techniques. We use the well known bulletin board technique to implement a deterministic naming scheme that also maintains lexicographical order between substrings. The bulletin board is frequently used in parallel string algorithms and our implementation shows how it can be used in practice. Indeed, for realisitic input sizes, the polynomial space requirement of the bulletin board soon exceeds that provided by main computer memory. In this case, we continue the suffix array construction using a parallel radix sort. Our algorithm resembles that of [12] in that it uses the *doubling technique* to derive logarithmic time bounds. However, we feel that a parallel implementation of the Manber and Myers algorithm is not a practical option for the MasPar architecture due to our use of external memory. In the sequel, the description of our algorithm is associated with its practical adaption for the massively parallel MasPar MP-2 2216.

2 Preliminaries and the MasPar

A *string* x is a finite sequence $x[1..n]$ of characters such that each $x[i]$ is drawn from an alphabet Σ. The length of x is n and is denoted $|x|$. A *substring* of x is a string $x[i..j]$ such that $1 \leq i \leq j \leq n$. Furthermore, we say that the substring $x[i..j]$ *occurs* at position i in x. A *prefix* of x is a substring $x[1..j]$ such that $1 \leq j \leq n$. A *suffix* of x is a substring $x[i..n]$ such that $1 \leq i \leq n$.

Given an input text $\mathcal{T} = x[1..n]$ and an ordered alphabet Σ, we define the set of suffixes of $\mathcal{T}$ to be $\{s_1, s_2, \ldots s_n\}$ such that $s_i = x[i..n]$. The *suffix array* of $\mathcal{T}$ is a dictionary or indexing data structure consisting of the following components:

(i) An array SA of size n of integers in the range $1..n$ representing the lexicographically sorted suffixes of $\mathcal{T}$.
(ii) Two further arrays, LEFT-LCP and RIGHT-LCP, each of size $n - 2$ and containing integers in the range $0..n - 1$. The arrays LEFT-LCP and RIGHT-LCP are used to guide the search during the query procedure.

Each integer value stored in the arrays in (ii) above is the length of the longest common prefix between a pre-determined pair of suffixes of $\mathcal{T}$. These suffix pairs are exactly those equal to all the intervals that arise during a binary search in SA: i.e., let $[j, k], 1 \leq j < k \leq n$ be any interval that arises during a binary search in SA. Furthermore let $i = \lfloor (j + k)/2 \rfloor$ be the midpoint of this interval. Then the value stored at LEFT-LCP$[i]$ is the longest common prefix between the suffixes associated with SA[i] and SA[j]. Also RIGHT-LCP[i] is the longest common prefix between the suffixes associated with SA[i] and SA[k].

A simple way to determine the supplementary information needed in (ii) above is to firstly compute an array LCP of length $n-1$ that contains the length of the longest common prefix between the suffixes represented by SA[i] and SA[$i+1$] for all $1 \leq i < n$. Our implementation uses this approach. Consider the string $\mathcal{T} = abbaabaaababbb$: then the array SA for T

$$\text{SA}[14] = [7, 4, 8, 5, 9, 1, 11, 14, 6, 3, 10, 13, 2, 12]$$

In this example s_7 is the lexicographically smallest suffix and s_{12} is the largest. The accompanying array LCP for SA is:

$$\text{LCP}[13] = [2, 4, 1, 3, 2, 3, 0, 1, 3, 2, 1, 2, 2]$$

An optimal algorithm for computing the minimum value in a given interval can be used to compute LEFT-LCP and RIGHT-LCP from the array LCP.

2.1 The MasPar

The massively parallel idiom is one which gains performance through replication by linking thousands of simple processing elements (PE's) via a suitable interconnection network in a specific topology. In the MasPar MP-2, 16,384 PE's are connected in a 128×128 two-dimensional mesh known as the PE Array. Each PE has 64K of local memory giving an aggregate 1 Gigabyte of distributed memory. In addition, PE's can access a shared memory of size 512K. (See [4] for more details) Communications between PE's are executed in one of two ways. The XNet interconnect furnishes fast local communications such that each PE can communicate with one of its 8 nearest neighbours (i.e., the adjacent horizontal, vertical and diagonal neighbours) in a *register to register* fashion. For more arbitrary communications there is a Global Router that is implemented using a *multi-stage interconnection network*. There is one originating router port and one target port per cluster of 16 PE's. Router communication is constant or independent of the position of the communicating PE's. We implement our algorithms using the MasPar Parallel Application Language, abbreviated to MPL, which allows the programmer specific control over data distribution and interprocessor communications, see [4] for details.

3 Data Structures and Techniques

3.1 Substring Naming

When gathering string statistics for a string x the technique known as *substring naming* is commonly used to group all equal substrings of x together and to associate a unique integer with each group. Substring naming together with recursive doubling (see [7]) are techniques that are widely used in parallel dictionary computations (see [6, 13, 10].) In [1] it was used in the parallel construction of suffix trees.

More formally, let $x[i..j]$ be a substring of an input text $\mathcal{T}$ of length $\ell = j-i+1$ for some $1 \leq i < j \leq n$. A deterministic substring naming function is a function that takes a substring such as $x[i..j]$ as an argument and returns an integer value *name*. We use ℓ*-name* to be the name given to a substring of $\mathcal{T}$ of length ℓ and ℓ*-name*(i) to denote the name of the substring of length ℓ that starts at position i in $\mathcal{T}$. The name for a string of length n can be computed in $\log n$ applications of the following naming function: $f($*name1*, *name2*$)$ = *newname*, where (*name1*, *name2*) is a tuple of integers, each of which is the name of a substring of length ℓ. Furthermore, *newname* is the 2ℓ*-name* of the substring of length 2ℓ that is formed by concatenating the substrings associated with *name1* and *name2*, i.e., the substrings ℓ*-name*(i) and ℓ*-name*$(i + 2^\ell), 1 \leq i \leq n$.

3.2 Naming Using a Bulletin Board

A *bulletin board* BB$[1..n][1..n]$, is a two-dimensional array data structure that is frequently used in CRCW PRAM string algorithms. It enables processors to update the value of a specific parallel variable in constant time as follows: all processors that wish to write to a location BB$[r, c]$ attempt to do so. Depending on which CRCW model is used, a write conflict resolution mechanism is used to determine a winning processor from amongst them that will succeed in writing. For our algorithm, the naming function is computed by associating the rows (r) and columns (c) of BB with the *name1* and *name2* values respectively. We then use BB to convert two values into one as required by the naming function as follows: for all processors associated with positions in $\mathcal{T}$ such that *name1* $=$ r and *name2* $= c$, the winning processor writes a 1 at location BB$[r, c]$. This location is now said to be *active*. A unique value is then associated with each active location by computing the prefix sums of all active locations. This is the value of the variable *newname*. All processors that attempted to write to BB$[r, c]$ can now read the same value of *newname* from this location. For full details of the above implementation see [8].

The bulletin board is implemented as a set of two dimensional arrays distributed across the memory of the PE array. For a bulletin board of size $n \times n$, and given a total of p processing elements, a simple distribution allocates processor $p_i, i = 0..p-1$ to the bulletin board location $[i \text{ div } n+1, i \bmod n+1]$. When $n^2 > p$ this data structure is *virtualised*. Restricting our array bounds to powers of 2 we found that with 1 Gigabyte of PE memory we can implement a bulletin board of size 4096×4096, together with the associated longest common prefix data structures. Therefore each PE is associated with a sub-block consisting of $32 \times 32 = 1024$ BB locations.

Initially, the input alphabet Σ is distributed to all PE's and the size of the first bulletin board required is $|\Sigma|+1$. We now remove all characters from Σ that are not contained in the input text $\mathcal{T}$ and assign *name*s to those that remain, since they represent all substrings of length 1 in the text. The input $\mathcal{T}$ is read in consecutive blocks consisting of p contiguous text characters from the external memory assigning one text position to each PE. Initially, it is possible to *stack* the text characters onto the PE's in layers since the bulletin board is very small. All bulletin board updates for each stage of sorting therefore require $O(n/p)$ external memory reads (and writes).

3.3 Longest Common Prefix Computation

Let $\mathrm{lcp}[i,j]_k$ denote the length of the longest common prefix between the substrings of the input text $\mathcal{T}$ starting at positions i and j of length k. For $1 \leq h \leq \log|\mathcal{T}|$ we have

$$\mathrm{lcp}[i,j]_h = \begin{cases} 2^{h-1} + \mathrm{lcp}[i+2^{h-1}, j+2^{h-1}]_{h-1}, & \text{if } \mathrm{lcp}[i,j]_{h-1} = 2^{h-1} \\ \mathrm{lcp}[i,j]_{h-1}, & \text{otherwise} \end{cases} \tag{1}$$

The value of the longest common prefix that exists between each suffix and its right neighbour (i.e., the suffix that is currently the next largest one) is updated at each stage of naming using the following arrays:

1. a $q \times q$ array NEW_LCP is used to store the values of the longest common prefix which exist between every pair of *newnames* computed in the *current* stage. (q is the largest name assigned in the current stage.)
2. an $w \times w$ array OLD_LCP is used to store the values of the longest common prefix which exist between every pair of *newnames* computed in the *previous* stage. (w is the largest name assigned in the previous stage.)

These arrays are distributed across PE memory and we exploit the architecture's ability to efficiently broadcast values within the rows and columns of the PE array.

4 Bulletin Board and LCP Table: Performance Results

In Algorithm *LCP* below we describe how the LCP values are maintained. The algorithm takes as input the array OLD_LCP and it outputs the array NEW_LCP. An "active" processor is one which is associated with a bulletin board location that has been marked 1 (and was therefore written to). Consider the computation to determine the longest common prefix that exists between two ℓ-strings at i and j in $\mathcal{T}$. Let α_1, β_1 denote the values of *name1* and *name2* associated with the substring at position i and α_2, β_2 denote the values of *name1* and *name2* associated with the substring at position j. From (1) we can immediately derive the following lemma that is used to guide the longest common prefix computation:

Lemma 1. Using the α, β representation of substrings, the common prefix between two substrings such that $\alpha_1 \neq \alpha_2$ does not change from the previous iteration of naming.

Algorithm LCP

STEP 1: The first row and first column, r_0 and c_0, of NEW_LCP are initialised as follows: each active processor writes its *name1* value to both α_1 of processor p_δ in row r_0 and to α_2 of processor p_δ in column c_0. The same processor then writes its *name2* value to both β_1 of processor p_δ in row r_0 and to β_2 of processor p_δ in column c_0.
STEP 2: All locations in NEW_LCP are now initialised as follows: processor p_δ of row r_0 broadcasts α_1 and β_1 to all processors in the same column, c_δ. Similarly, each processor p_δ of the column c_0 broadcasts α_2 and β_2 to all the processors in the same row, r_δ.
STEP 3: Each location in NEW_LCP has now received 4 values which are grouped into the two pairs: $\gamma = (\alpha_1, \alpha_2)$ and $\zeta = (\beta_1, \beta_2)$.
STEP 4: All locations in the table NEW_LCP now consider their γ and ζ pairs. If $\alpha_1 \neq \alpha_2$, then by fact (1) and Lemma 1 the longest common prefix value does not change, else the longest common prefix value is equal to $|\alpha_1|+$ OLD_LCP$[\beta_1, \beta_2]$.

Table 1 shows how rapidly the size of the bulletin board grows for increasing lengths of English text and DNA. The results show that the bulletin board size for DNA and English text inputs grows very rapidly and that only 3 and 2 iterations respectively of naming are possible for input sizes larger than 16384: suffixes are sorted according to a uniform prefix of length 8. Indeed, the largest bulletin boards that are made use of are of size 260^2 for DNA and 539^2 for English text.

Table 1. *Iterations* = number of iterations of naming, *Last BB* = largest BB realised, *Next BB* = next BB size required, *Secs* = time in seconds

DNA				
n	*Iterations*	*Last BB*	*Next BB*	*Secs*
8k	11	260	6818	88.07
16k	3	260	12002	3.41
32k	3	260	21946	4.62
64k	3	260	34767	7.83
128k	3	260	47664	14.59

English Text				
n	*Iterations*	*Last BB*	*Next BB*	*Secs*
8k	4	4036	4138	31.52
16k	2	450	7222	77.34
32k	2	487	12389	3.09
64k	2	507	15150	4.99
128k	2	539	22611	9.09

5 Radix Sort via Merge-Sort

In Section 4 we showed that the bulletin board cannot be used to construct SA for large input sizes due to memory restrictions. When our memory limit is reached say at iteration j of renaming, the input text $\mathcal{T}$ can be represented by an unordered array of integers in the range $1..q$, where q is the largest *newname* computed in the last iteration of the naming stage. By sorting these integers, (each tagged with the index of the suffix it represents), the suffix indices can now be rearranged into their current lexicographic order. In our implementation we do this using parallel merge-sort. In this way, all suffixes that share a common prefix of length at least ℓ are placed into a *partially* completed suffix array in contiguous locations, PART_SA. Furthermore, each suffix can now be encoded by a new, shorter string as follows: let s_i be a suffix of $\mathcal{T}$. We define the q-string of s_i to be $(k_1, k_2, \dots k_t)$ such that

(i) k_j for some $1 \leq j \leq t$ is the name of the substring of length ℓ starting at position $i + \ell(j-1)$ in $\mathcal{T}$.
(ii) $0 \leq k_j \leq q$, for $1 \leq j \leq t$.

We call each k_j a k-component and use the value 0 to denote a k-component that occurs beyond position n in the input. At this stage the construction of the array SA is completed using a *radix* sort, to base $q+1$ and each round of this radix sort is a parallel merge-sort.

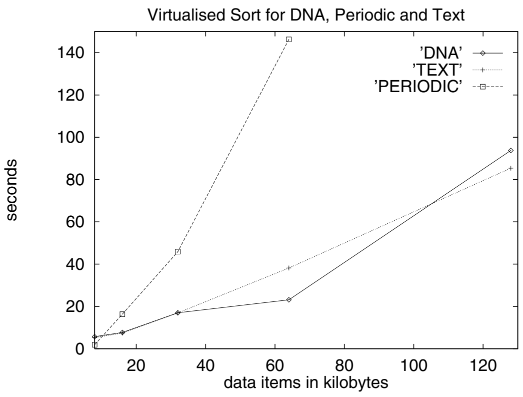

Fig. 1. Merge-Sort applied to DNA, Highly periodic and Text strings

5.1 Parallel Merge-Sort

The underlying computational structure of the traditional PRAM algorithm for merge-sort is a complete binary tree. Our merge procedure is an adaption of the $O(\log \log n)$ merging of [9] and [15]. To avoid the communication overheads that are incurred by their $O(1)$ parallel ranking procedure, we implement ranking using a simple binary search. This increases the (theoretical) time complexity for the merging to $O(\log n \log \log n)$. We implement the binary tree computation using a $2 \times n$ array SORT. The merge-sort procedure requires $O(\log^2 n \log \log n)$ time, using n processors. For $p < n$ processors this data structure is virtualised using a cut-and-stack data mapping as in [5]: no two contiguous input elements reside on the same processor. Figure 1 shows the time taken to perform this merge-sort for three different input types.

From Figure 1 we see a sudden jump in the running time for the DNA input of size $32k$. This particular sample contains 29 occurrences of the substring $TTTTTTTT$, the other samples of DNA contain no more than 6 such substrings. In comparison, the input samples of text contain no such repetitive substrings of this length. Substring statistics in relation to suffix trees have been studied extensively by Szpankowski in [14].

5.2 Segmented Merge-Sort

Scan primitives can be applied to vectors which are divided into contiguous blocks called *segments*. The segments are defined using *flags* which mark the *end-of-segment* boundaries. If there are m such flags in a given input vector $\mathcal{V}$, the primitive will be executed $m+1$ times in parallel, once for each segment in $\mathcal{V}$. In our implementation we consider each segment as a bucket, the contents of which are the indices of suffixes of $\mathcal{T}$ that share a common prefix. The buckets are refined using segmented merge-sort until each contains only one suffix index. The segment boundaries are defined using the following rule: any virtual processor that is associated with a different *ℓ-name* to its right neighbour signifies the end of a segment. We achieve the effect of applying merge-sort to all segments in parallel by renumbering each (virtual) processor so that the first processor of each segment has number 0. For each virtual processor, its new processor number *new_iproc* depends on the position at which its segment starts and on its original (virtual) processor number. Algorithm **RADIX** below takes as input the following items: (i) an unordered array of names in the range $1..q$ where q was the largest *newname* assigned in the last iteration, say iteration j, of renaming (each name represents the prefix of length 2^j of each suffix of $\mathcal{T}$); (ii) the associated LCP values; (iii) the table NEW_LCP, also computed in iteration j of renaming. The output is the completely sorted suffix array, SA.

Algorithm RADIX

STEP 1. Each suffix s_i is represented by its q-string $q(i)$. Compute the partial suffix array PART_SA as follows: apply the merge-sort procedure to the first k-component k_1, of each q-string. Each q-string is relocated to its newly sorted position together with the index of the suffix which it represents and the associated LCP value. For $j = 1..\lfloor n/2^\ell \rfloor$ do each of the following substeps:
STEP 2a. Using component k_j as a key, mark all locations t in PART_SA such that the k-component for PART_SA$[t]$ is not equal to the k-component for PART_SA$[t+1]$. This partitions the array PART_SA into segments such that all locations with the same k-component belong to the same segment or *equivalence class*.
STEP 2b. Compute the *size* of each segment. If all segments are of size 1, then compute the LCP values for the entire input and terminate the computation. Else, for all locations contained in segments of size 1, store the value of k_j in K, $j-1$ in J and mark the location as "non-active".

STEP 2c. For all "active" segments of size greater than 1, apply the merge-sort procedure using the component k_{j+1} as the sort key. Remove k_j from each $q(i)$ and send the remaining q-string to its new sorted location.

6 Conclusion

We give an algorithm for the parallel construction of the suffix array data structure. Our empirical results show that the PRAM bulletin board technique for naming and sorting substrings requires more main memory than is available on a MasPar MP-2 for practical input sizes. This is partly due to the fact that we also maintain tables for computing longest common prefix information. The theoretical time of $O(\log^2 n)$ "reduces" to $O(kn/p \ \log n)$ where $k < \log n$ is the number of BB iterations that main memory can accomodate, and p is the number of PE's. However, the bulletin board can be used to reduce the input size and to create a more succinct representation of each suffix, the q-string. Each character in a q-string represents a substring of length 2^ℓ for some $0 \leq \ell < \log n$. For DNA and English text the length of q-strings is in the region of $n/4$. A radix sort is then used to complete the suffix array construction. Assuming that the main memory of the PE array holds n integer values, one round of the radix sorting is implemented using a segmented merge-sort and requires $O(n/p \ \log^2 m \log\log m)$ time, where $m < n$ is the size of the largest segment or bucket to be sorted in the current round. The number of rounds of radix sort required depends on the length of the longest repeated substring in the input: for DNA and English text we found that at most 3 rounds were required to eliminate all but a small fraction of q-strings from the sort.

References

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

[2] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. In *Proceedings of Combinatorial Pattern Matching*, pages 102–115, 1996.

[3] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software - Practice and Experience*, 25, 2:129–141, 1995.

[4] Digital Equipment Corporation, Maynard, Massachusetts. *DECmpp Programming Language Reference Manual*, 1992.

[5] C. Farrell and D. Kieronska. Implementation of automatic virtualisation into parallel compilers for massively parallel simd architectures. In *SS'93 Conf on High Performance Computing*, pages 55–61, Calgary, Canada, 1993.

[6] Z. Kedem, G. M. Landau, and K. Palem. Optimal parallel suffix-prefix matching algorithm and applications. In *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 388–398, 1989.

[7] R. Karp, R. Miller, and A. Rosenberg. Rapid identification of repeated pattrns in strings, trees and arrays. In *Proc. 4th Annual ACM Symposium on Theory of Computing*, pages 125–136, 1972.

[8] M. Korda. *Two dimensional string covering and parallel suffix array construction.* PhD thesis, King's College, University of London, Strand, London, 1998.

[9] C. Kruskal. Searching, merging and sorting in parallel computation. *IEEE Transaction on Computers*, C-32;10:942–946, 1983.

[10] G. M. Landau and U. Vishkin. Pattern matching in a digitized image. *Algorithmica*, 12(4/5):375–408, 1994.

[11] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[12] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[13] S. Muthukrishnan and K. Palem. Highly efficient parallel dictionary matching. In *Proc. 5th ACM Symposium on Parallel Algorithms and Architechtures*, 1993.

[14] W. Szpankowski. Expected behaviour of typical suffix trees. In *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 422–431, 1992.

[15] L. G. Valiant. Parallelism in comparison models. *SIAM Journal on Computing*, 4:348–355, 1975.

[16] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.