1-1-2001

Parallel suffix sorting

Natsuhiko Futamura
Syracuse University, Department of Electrical Engineering and Computer Science, nfutamur@ecs.syr.edu

Srinivas Aluru
Syracuse University

Stefan Kurtz
Universitat Bielefeld, Technische Fakultat, kurtz@techfak.uni-bielefeld.de

Parallel Suffix Sorting

Natsuhiko Futamura*
School of EECS
Syracuse University
Syracuse, NY, USA
nfutamur@ecs.syr.edu

Srinivas Aluru†
Dept. of ECpE
Iowa State University
Ames, IA, USA
aluru@iastate.edu

Stefan Kurtz
Technische Fakultat
Universitat Bielefeld
Bielefeld, Germany
kurtz@techfak.uni-bielefeld.de

Abstract

We present a parallel algorithm for lexicographically sorting the suffixes of a string. Suffix sorting has applications in string processing, data compression and computational biology. The ordered list of suffixes of a string stored in an array is known as Suffix Array, an important data structure in string processing and computational biology. Our focus is on deriving a practical implementation that works well for typical inputs rather than achieving the best possible asymptotic running-time for artificial, worst-case inputs. We experimentally evaluated our algorithm on an IBM SP-2 using genomes of several organisms. Our experiments show that the algorithm delivers good, scalable performance.
Keywords : Burrows-Wheeler transform, computational biology, suffix arrays, suffix trees.

1 Introduction

Let S be a string of length n over an alphabet Σ. Let s_i denote the i^{th} character of S. The i^{th} suffix of S, denoted $Suff_i$, is the substring starting at the i^{th} position of S and ending at the end of S, i.e., $Suff_i = s_i s_{i+1} \ldots s_n$. Suffix sorting is the problem of lexicographically ordering the n suffixes of S. We denote the suffixes by integers using their starting positions in the string. Thus, the output of suffix sorting is a permutation of integers $1, 2, \ldots, n$, listing the suffixes in sorted order using their integer representation.

The sorted list of suffixes of a string reveals the structure of the string and allows the design of efficient algorithms for many string problems. Two fundamental data structures in string processing, *Suffix trees* [4] and *Suffix Arrays* [10], are based on the representation of all suffixes of the underlying string and are related to suffix sorting. These data structures are useful in pattern matching and in many applications in text processing and computational biology [5]. Suffix sorting is also useful in data compression. For example, it is the most time-consuming operation in computing the *Burrows-Wheeler transform* [3].

Any comparison-based sorting algorithm can be used to sort suffixes by using lexicographic string comparison. The resulting algorithm will run in $O(n^2 \log n)$ sequential time because each string comparison takes $O(n)$ time and $O(n \log n)$ comparisons are used. However, much faster methods can be designed because the suffixes are overlapping substrings coming from the same string. In fact, suffix sorting can be done in $O(n)$ time by first constructing a suffix tree of the string [4], followed by a lexicographic traversal of the suffix tree to report the suffixes in sorted order. Even though this algorithm is optimal, it is rarely used in practice due to the large constants in the run-time and space complexities of suffix tree construction.

Suffix array of string S, denoted SA, is an array of size n storing the sorted order of suffixes of S, i.e., $SA[i]$ is j iff $suff_j$ is the i^{th} lexicographically smallest suffix. Suffix arrays are introduced by Manber and Myers [10] as a space-efficient alternative to suffix trees. When augmented with an array storing *longest common prefix* values, suffix arrays can be used in place of suffix trees with modest loss of run-time. It should be clear that the result of suffix sorting is the suffix array and hence the problem of constructing suffix arrays is identical to suffix sorting.

The fastest known sequential algorithms for suffix sorting, without involving the use of suffix trees, run in $O(n \log n)$ time. Manber and Myers presented such an algorithm in the context of constructing suffix arrays [10]. Larsson and Sadakane devised a faster and memory efficient $O(n \log n)$ algorithm by equating suffix sorting to the construction of a ternary sorting tree [8] of $O(\log n)$

*Work performed while conducting research at Iowa State University.

†Research supported by NSF CAREER under CCR-0096288.

worst-case depth. To the best of our knowledge, no parallel algorithms for suffix sorting are reported in the literature.

Recently, Bentley and Sedgewick presented a fast algorithm for sorting arbitrary strings [2], which can also be used to sort suffixes. While specialized algorithms for suffix sorting have a superior asymptotic worst-case performance, Bentley and Sedgewick's algorithm is known to perform well in practice. Larsson and Sadakane [8] have compared several sequential algorithms for suffix sorting. Their results demonstrate that Bentley and Sedgewick's algorithm delivers competitive performance and in fact has the best run-time in about half of the data sets tested.

A related problem is the construction of suffix trees, introduced by Weiner [13]. Space-efficient linear time algorithms for suffix tree construction are given by McCreight and Ukkonen [9, 12]. For a recent survey of suffix tree construction algorithms, see [4]. Parallel construction of suffix trees has been studied on the CREW PRAM model [1, 7]. These algorithms do not, at least directly, translate into efficient implementation on parallel computers.

In this paper, we present a parallel algorithm for sorting the suffixes of a string. We are motivated by the increasing number of applications in computational biology that employ suffix trees and suffix arrays. In one parallel step, our algorithm partitions the suffixes into buckets, which are then assigned to individual processors for local sorting. We have experimentally evaluated our algorithm using the genomes of several organisms, which are large strings over the alphabet $\{A, C, G, T\}$. The run-time of our algorithm scales linearly with the number of processors as our experimental results demonstrate. Our algorithm does no better than the sequential algorithm on a worst-case string, which corresponds to a string consisting of a repeated character. Such worst-case strings are unlikely in practical applications, and our experience suggests that the overhead of an algorithm designed to improve the worst-case performance severely drags down the performance on strings encountered in practical applications. Similar behavior is observed for the problem of sequential construction of suffix trees [4], where an algorithm with $O(n^2)$ worst-case and $O(n \log n)$ expected cased running time is shown to be superior to worst-case linear time algorithms based on experimental evaluation with realistic data sets.

The rest of the paper is organized as follows: In Section 2, we present the parallel primitive operations used in describing our algorithm. In Section 3, we present our parallel suffix sorting algorithm. Experimental evaluation of the algorithm and the corresponding results are presented in Section 4. Section 5 concludes the paper.

2 Preliminaries

In the following, we describe the primitive parallel operations used to describe our algorithm. These operations are well-known and supported by parallel programming standards such as MPI [11]. In what follows, p denotes the number of processors.

All-Reduce: In this operation, each processor has one data item and we are given a binary associative operator that operates on two data items and produces a result of the same type. The objective is to combine all the data using the operator and store the result on all processors. In the vector version of this operation, each processor has an array of data items and the data items in the same index of the arrays are combined.

All-to-All Communication: In this operation, each processor sends a message to every other processor. This is typically used to redistribute data in a parallel computer.

3 Parallel Suffix Sorting

In this section we present our parallel algorithm for suffix sorting, or equivalently, for suffix array construction. Consider a string S of size n. Use an array SA of the same size, distributed across processors such that P_i contains $SA[i\frac{n}{p} + 1 \ldots (i + 1)\frac{n}{p}]$. The array SA is initialized such that $SA[i] = i$, i.e., it represents an unordered list of all suffixes of S.

Let $w \geq \log_{|\Sigma|} p$ denote a *window-size*. The algorithm consists of two phases: a parallel phase followed by a sequential phase. In the parallel phase, the subarray on each processor is divided into $(|\Sigma|)^w$ buckets based on the first w characters of each suffix. This is done as follows: There are $(|\Sigma|)^w$ possible words of length w. Create an array B of size $(|\Sigma|)^w$ on each processor. Let W_i denote the i^{th} lexicographically smallest word of length w. Compute B such that $B[i]$ is the number of suffixes assigned to the processor which start with W_i. To compute this, the processor initializes all entries of B to zero and examines each suffix in turn, incrementing $B[i]$ if the suffix examined starts with W_i. Each suffix can be examined in $O(w)$ time, resulting in $O\left(\frac{n}{p}w\right)$ run-time for constructing B on each processor.

This can be reduced to $O\left(\frac{n}{p} + w\right)$ time, thus making it independent of the window-size (assuming $w << \frac{n}{p}$), as follows: Let $f : \Sigma \rightarrow$

$\{0, 1, \ldots, |\Sigma| - 1\}$ be the one-to-one function such that $f(c) = j - 1$ if c is the j^{th} lexicographically smallest character. Using f, each string can be treated as a number represented in base $|\Sigma|$ in the usual manner and converted to its decimal equivalent. In particular, words of size w are represented by integers in the range $0 \ldots (|\Sigma|^w - 1)$. In considering a suffix, first convert the w-long prefix of the suffix to its corresponding number and use this number to index into array B to increment its count. Converting the w-long prefix of the first suffix to a number takes $O(w)$ time. Because the w-long prefixes of two consecutive suffixes differ in a single character, the number for one can be derived from the number for the previous one in constant time. Suppose we just considered $Suff_k$ and the number corresponding to its w-long prefix, $s_k s_{k+1} \ldots s_{k+w}$, is X. Then, the number corresponding to the w-long prefix of $Suff_{k+1}$ is computed as $\left(X - f\left(s_k\right)|\Sigma|^{w-1}\right) \times |\Sigma| + f\left(s_{k+w+1}\right)$. Precomputing and storing $|\Sigma|^{w-1}$, this computation takes only constant time per suffix.

Once array B containing the count of the number of suffixes falling in each bucket is formed, the local subarray of SA can be partitioned into $|B| = |\Sigma|^w$ buckets. This can be done using a temporary array T partitioned according to the bucket sizes and copying each suffix in SA into T according the bucket it belongs to. For convenience, we will continue to use SA to denote the array partitioned into buckets.

The next step is to allocate the buckets to individual processors and sort each bucket locally on the processor it is assigned to. Note that the suffixes belonging to each bucket are themselves partitioned across all the processors; each processor contains as many suffixes of bucket i as given by its local $B[i]$. An *All-Reduce* operation on the B array is used to find the total number of suffixes belonging to each bucket. As a result of this operation, the array B on each processor now contains the total number of suffixes in bucket i. Any load balancing heuristic can be used to examine B and allocate the buckets to processors. The goal is to balance the total number of suffixes allocated to each processor. As B is stored on each processor, no communication is necessary to identify the assignment of buckets to processors.

We use the following simple scheme in our implementation. In an ideal setting, each processor should be assigned exactly $\frac{n}{p}$ suffixes. Imagine the entire array SA partitioned into p parts of length exactly $\frac{n}{p}$. If a partition falls within a bucket, we readjust the partition so that it coincides with the nearest bucket boundary. By computing a prefix sum on array B, the bucket containing each partition can be found and the buckets that should be assigned to each processor can be easily identified.

It now remains to distribute the buckets to their respective processors. The suffixes belonging to a bucket are currently distributed across all processors and they should be sent to the processor to which the bucket is assigned. As each processor known the buckets assigned to all the processors, all buckets can be directed to their respective processors using an *All-to-All Communication*.

At this stage, the array SA is partitioned into buckets and also partitioned across processors such that each bucket lies completely within a processor. The suffixes belonging to each bucket are sorted sequentially within the processor containing the bucket using a standard sequential algorithm. A sequential suffix sorting algorithm cannot be used to sort the suffixes within a bucket. This is because suffix sorting algorithms take advantage of the fact that *all* suffixes from a string need to be sorted. Thus, we must use an algorithm that sorts arbitrary strings of characters. We draw from Bentley and Sedgewick's recent work [2] on fast algorithms for sorting strings.

The running time of the algorithm is clearly affected by the load balancing achieved in allocating the buckets to processors. The *window-size* parameter plays a role in this by determining the number of buckets. For a window-size of w, the number of buckets generated is at most $|\Sigma|^w$. Fewer buckets may be generated if all possible w-long words are not substrings of string S. At least p buckets are needed for a fair allocation of buckets to processors. Thus, w should be at least as large as $\log_{|\Sigma|} p$. A larger value of w will increase the number of buckets and allows better load balancing. It also increases the size of the bucket array B, placing an upper bound of $\log_{|\Sigma|} \frac{n}{p}$ on the maximum allowable window-size. The optimal value for the window size, expected to be function of p, can be found by experimentation.

The algorithm should perform well for random strings. For a random string, the expected length of a prefix sufficient to differentiate between two suffixes is $O(\log n)$ [4]. This also ensures that the sequential algorithm used for sorting each bucket runs fast. The worst-case input for our algorithm is a string consisting of a single repeated character. In this case, only one bucket contains all the suffixes irrespective of the window-size used and the parallel algorithm will degenerate to running the sequential algorithm on the whole input string. Such a worst-case input is clearly artificial and is not representative of strings encountered in practical applications. In the next section, we present experimental evaluation of our algorithm on repre-

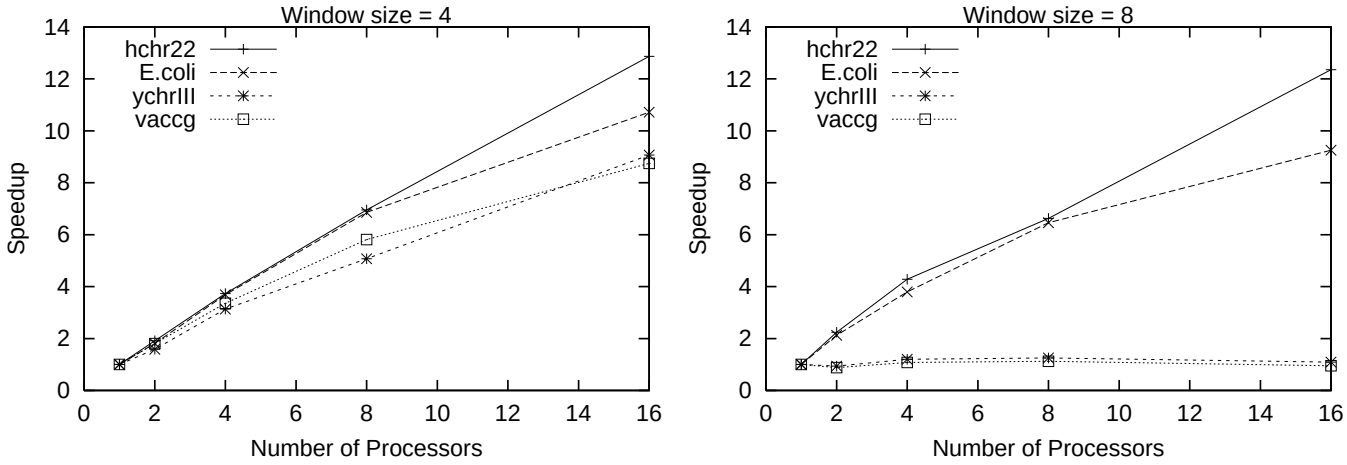

Figure 1: Speedup as a function of the number of processors for fixed window size.

sentative data sets drawn from real applications.

4 Experimental Results

We implemented our parallel suffix sorting algorithm using C and MPI. In the parallel phase of the program, the array holding the suffixes is partitioned into buckets such that the buckets are sorted with respect to each other but not within themselves. The buckets are allocated to processors for further sorting within the buckets. In the sequential phase of the program, processors sort the buckets assigned to them using Bentley and Sedgewick's algorithm [2]. We have used their code, made available in the public domain, for this purpose.

Because computational biology is an important application area for suffix sorting (or equivalently, suffix array construction), we experimentally evaluated the program using data sets drawn from genomes of several organisms. A genome consists of one or more DNA sequences, which are strings over the alphabet $\{A, C, G, T\}$. In higher organisms, the genome consists of pairs of chromosomes, each of which is a DNA sequence. We tested the program on the following data: complete genome of the vaccinia virus (*vaccg*, length $191,737$ bases), chromosome III of the yeast genome (*ychrIII*, length $315,339$ bases), complete genome of *E. Coli* (length $4,638,690$ bases), and the first 10 million bases (characters) of human chromosome 22 (*hchr22*).

The program is run on an IBM SP-2 using each data set and varying the number of processors and the window size w. We considered three important features of the program for experimental evaluation: 1) the scaling of performance with respect to the number of processors, 2) the distribution of the bucket sizes as a function of the window-size, and 3) the effect of window-size on the run-time.

The scaling of the program for various data sets using window sizes of 4 and 8 is shown in Figure 1. In drawing the graphs, the running time of the sequential program is taken to be the runtime of Bentley and Sedgewick's sequential code, not the parallel program running on one processor. If the window size is appropriately chosen, the program exhibits good scaling with number of processors. Notice that superlinear speedup is observed in some cases. This is because Bentley and Sedgewick's algorithm has a more than linear time complexity ($n\times$ number of characters required to differentiate between input strings), making it advantageous to run the algorithm on several small problems instead of a single large problem of the same total size. In some cases, the benefit obtained by this more than compensates for the time spent in parallel phase, resulting in superlinear speedup. The size of the bucket array B grows exponentially with window size ($|B| = |\Sigma|^w$). The poor performance of the program on *vaccg* and *ychrIII* when $w = 8$ is because w is too large when compared to the data size. In fact, for this choice of parameters, the bucket size on a processor exceeds the size of the suffix array allocated to it, violating the upper bound requirement on the size of w $\left(\log_{|\Sigma|} p\right)$.

While the choice of w dictates the exact number of buckets, the number of suffixes falling into respective buckets is completely dependent upon the input data. The average bucket size is given by $\frac{n}{|\Sigma|^w}$, the ratio of the number of suffixes divided by the number of buckets. From the viewpoint of load distribution, the size of the largest bucket is of importance. The average bucket size, the largest bucket size, and the ratio of the largest

w	*hchr22*			*E.coli*			*ychrIII*			*vaccg*		
	avg. size	max size	ratio	avg. size	max size	ratio	avg. size	max size	ratio	avg. size	max size	ratio
2	625000	786955	1.25	289918	383734	1.32	19709	34494	1.75	11984	23112	1.93
3	156250	275407	1.76	72480	115631	1.59	4927	12680	2.57	2996	7784	2.60
4	39063	112627	2.88	18120	37472	2.07	1232	4903	3.98	749	2839	3.796
5	9766	55377	5.67	4530	13377	2.95	308	1985	6.45	187	1061	5.66
6	2441	32089	13.14	1133	5392	4.76	77	901	11.70	47	400	8.55
7	610	22804	37.36	283	2139	7.56	19	480	24.94	12	205	17.52
8	153	17895	117.28	71	770	10.88	5	297	61.72	2.9	117	39.99

Table 1: The average size of a bucket, maximum size of a bucket and the ratio of the maximum size to the average size, depicted as a function of w. The average size, given by $\frac{n}{|\Sigma|}$, is rounded to the nearest integer. The maximum size is measured by experiment.

Window size = 2

p	Total time	Parallel Phase	Serial Prhase
1	96.338731		-
2	88.366306	14.792675	73.573631
4	33.709612	7.976901	25.732711
8	20.373651	4.617254	15.756397
16	11.949479	3.181977	8.767502

Window size = 8

p	Total time	Parallel Phase	Serial Prhase
1	96.338731		-
2	42.929573	22.347625	20.581948
4	22.506574	12.262658	10.243916
8	14.542419	9.240219	5.3022
16	7.797947	5.15593	2.642017

Table 2: Decomposition of the total run-time into time spent in parallel and serial phases for *hchr22*. The run-time of the parallel phase increases with w and the run-time of the serial phase decreases with w.

bucket size to the average are shown in Table 1 for varying w. As expected, increasing the number of buckets increases the ratio of the largest to the average bucket. However, the size of the largest bucket decreases without exception. This should be so, because an increase in window size splits each existing bucket into several buckets. Therefore, the largest bucket size must necessarily be a non-increasing function of w. Due to the increasing ratio, the largest bucket size does not decrease proportionately with the increase in number of buckets, limiting the advantage derived from increasing the buckets to less than the maximum possible.

A decomposition of the total running time into parallel and sequential phases for various number of processors is shown in Table 2. As w increases, the run-time of the parallel phase increases and the run-time of the serial phase decreases. This is because we generate the bucket array of size $|\Sigma|^w$ and perform an *All-Reduce* operation on it in the parallel phase. The serial phase benefits from a larger w due to fragmentation of the number of suffixes into a larger number of smaller subproblems. Also, a very small value of w does not create enough buckets to effect sufficient load balancing for the serial phase. Analytically, we can place lower and upper bounds on w as $\log_{|\Sigma|} p \leq w \leq \log_{|\Sigma|} \frac{n}{p}$. This places a restriction on the number of processors that can be used for a fixed data size – $n \geq p^2$ to allow w to satisfy the lower and upper bounds. The program will run correctly even otherwise, but the performance will be affected. For a fixed number of processors, as w ranges from its lower bound to upper bound, the total run-time should form a U-shaped curve, the bottom of the curve yielding optimal value of w. Near the lower bound value of w, the run-time decreases with increase in w. Near the upper bound, the run-time sharply increases with increase in w (because the communication cost increases exponentially with w). Towards the bottom, the curve need not be sharp and result in a unique minimum because it depends on the input data. The variation in run-time as a function of w for *vaccg* and *hchr22* is shown in Figure 2. For *vaccg*, the range of permissible values of w is small because of the small data size, and the entire range is shown in the figure. Our experiments indicate that a value of w that is slightly smaller than its upper bound ($\log_{|\Sigma|} \frac{n}{p}$) results in the best running time. As w is increased from its lower bound, the sharpest decreases in the run-time occur near the lower bound, with decreasing incremental advan-

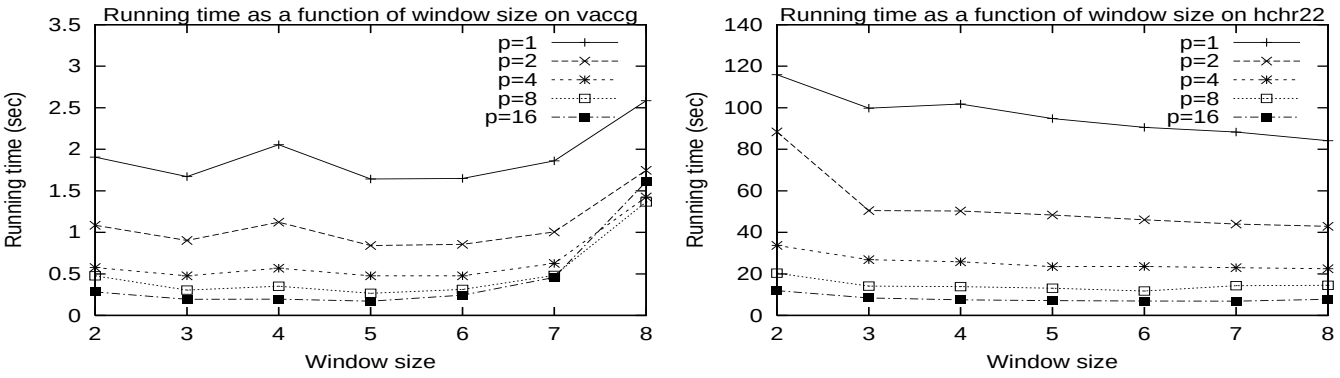

Figure 2: Total run-time as a function of w for *vaccg* and *hchr22*.

tage as w increases. Thus, a value of w that is at a conservative distance from the upper bound appears to be the best choice.

5 Conclusions

We presented a simple parallel algorithm for suffix sorting, or equivalently, the construction of suffix arrays. We have created a fast implementation of this algorithm using Bentley and Sedgewick's code for the sequential part. A thorough experimental evaluation of our program using data sets from computational biology produced satisfactory results. A similar approach can be used to derive a fast parallel algorithm for suffix tree construction. Design of worst-case optimal parallel algorithms for suffix sorting and a comparison of their implementation with ours remains an open problem.

Acknowledgements

We gratefully acknowledge the Maui High-Performance Computing Center for access to IBM SP-2.

References

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

[2] J. Bentley and R. Sedgewick, "Fast Algorithms for Sorting and Searching Strings," *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 360-369, 1997.

[3] M. Burrows and D.J. Wheeler, *A block-sorting lossless data compression algorithm*, Technical Report, SRC (digital, Palo Alto), no. 124, 1994.

[4] R. Giegerich, S. Kurtz and J. Stoye, "Efficient implementation of lazy suffix trees," *Proc. 3rd Workshop on Algorithm Engineering, Lecture Notes in Computer Science, vol. 1668*, pp. 30-42, Springer-Verlag, Berlin, 1999.

[5] D. Gusfield, *Algorithms on strings, trees and sequences*, Cambridge University Press, New York, NY, 1997.

[6] R. Giegerich and S. Kurtz, "From Ukkonen to Mcreight and Weiner: A unifying view of linear-time suffix tree construction," *Algorithmica, vol. 19* pp. 331-353, 1997.

[7] R. Hariharan, "Optimal parallel suffix tree construction," *Journal of Computer and System Sciences, vol. 55, no. 44*, pp. 44-69, 1997.

[8] N.J. Larsson and K. Sadakane, "Faster suffix sorting," *Technical Report in Department of Computer Science, Lund University, Sweden, LU-CS-TR:99-214*, 1999.

[9] E.M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM, vol. 23, no. 2*, pp. 262-272, 1976.

[10] U. Manber and G. Myeres, "Suffix arrays: A new method for on-line string searches," *SIAM Journal of Computing, vol. 22, no. 5*, pp. 935-948, 1993.

[11] P.S. Pacheco, *Parallel programming with MPI*, Morgan Kaufmann Publishers, Los Altos, CA, 1997.

[12] E. Ukkonen, "On-line construction of suffix-trees," *Algorithmica, vol. 14, no. 3*, pp. 249-260, 1995.

[13] P. Weiner, "Linear pattern matching algorithms," *Proc. IEEE Symposium on Switching and Automata Theory*, pp. 1-11, 1973.