APPENDICE A

ON AN IMPROVED PARALLEL CONSTRUCTION OF SUFFIX ARRAYS FOR LOW BANDWIDTH PC-CLUSTER

Pusat Pengajian Sains Komputer[1]
Pusat Pengajian Sains Matematik[2]
Kok Jun Lee[1]
Nur'Aini Abdul Rashid[1]
Rosni Abdullah[1]
Norhashidah Md. Ali[2]

Universiti Sains Malaysia
11800 Minden,
Pulau Pinang,
MALAYSIA
Tel.: +(604) 65377888 **Fax.:** +(604) 6573335
Main author email: kokjl@hotmail.com

Presented at
International Symposium on Information Technology 2003

Crown Princess Hotel
Kuala Lumpur Malaysia.
30th Sept – 2nd October

ON AN IMPROVED PARALLEL CONSTRUCTION OF SUFFIX ARRAYS FOR LOW BANDWIDTH PC-CLUSTER

Universiti Sains Malaysia
kokjl@hotmail.com, (nuraini, rosni, shidah)@cs.usm.my

Abstract: An algorithm for the parallel construction of suffix arrays generation for any texts with larger alphabet size on distributed memory architecture is presented. Our main goal is to achieve constant communication complexity for any string with large alphabet size while the number of processors involved in the computation is increased, and to derive a practical implementation that works well for typical input string. Our experiments using several genomes and texts data set on low cost network PC-cluster shows that our algorithm works well and scalable. We present some experimental evidence to show that our parallel algorithm works well even in low bandwidth network architecture.

Keywords: Affordable Computing, Knowledge Representatives, Parallel Processing, Distributed Computing

1. INTRODUCTION

Suffix array is the ordered indexes of suffixes in lexicographical for a given string. Let $T[1, n]$ denote a string over the alphabet Σ. The suffix array for T is an integer array $SA[1,n]$ such that $\forall i \in \{1, 2, \ldots, n\}. T[SA[i], n] < T[SA[i+1], n]$, where "<" denote the lexicographical order. It is a conceptually simple [17] and compact [17] data structure which is important in string processing, computational biology [3], and data compression [11].

The suffix array is the most compact and simple among suffix automaton and suffix tree. The searching time is competitive with the suffix tree in practice [17], but the suffix arrays do require three to ten times longer to be built. Suffix array construction is simple as it can be constructed even with any string sorting algorithm. However, general purpose string sorting algorithm fails to take advantage of the fact that we are sorting a collection of related suffixes. Much faster sequential methods ([5], [7], [11], [14], [17]) have been designed for sorting the suffixes of a string. Another alternative to overcome the computational bottleneck in the construction of suffix arrays is the utilization of parallel algorithms.

The use of a set of connected processors, as parallel machine is an attractive alternative nowadays. This type of distributed memory parallel machine is a good cost-performance tradeoff. One may employ fast switching technology to allow the dissemination of high-speed networks of processors at relatively low cost compared to shared memory super computer.

A number of parallel and distributed suffix array construction algorithms have been designed for message passing environment previously ([6], [9], [10], [13]). Performance of the algorithm in such platform is mainly affected by its total communication overhead. The communication complexity usually depends on the number of processors involved, the number and length of the messages to be sent and received. One may reduce the communication overhead by designing an algorithm with less communications as possible [13] or by choosing efficient communication network machine [9] such as **IBM SP** based on the High Performance Switch (HPS) or Myrinet switch cluster.

All previous works ([6], [9], [10], [13]) were experimented and analyzed on efficient communication network machine. They obtained good and scalable performance on such platform. Unfortunately, it is not clear whether ([6], [9], [10], [13]) will perform well if they run their works on a low cost network PC-cluster with low bandwidth which is affordable by most people today. The main issue is that their algorithms involve communication overhead. This issue leads us to design a parallel algorithm without any communication. A little constant computational overhead is incurred, but it is worth since our consideration is to reduce very expensive communication cost. We intend to experiment our work on low cost network PC-cluster with low bandwidth to see if the same performance can be achieved. The parallelism model we adopt here is that of low cost network PC-cluster with low bandwidth where the communication cost is very expensive.

2. PREVIOUS AND RELATED WORK

It is necessary to distribute sorting workload in order to sort all suffixes of a string concurrently. Several parallel algorithms are currently available to sort suffixes in distributed memory architecture

A generalization of the parallel quick sort is presented in [6]. This algorithm considers the global sorted suffix arrays which results in the sorting task to be broken into n/p similarly-sized portions and distributed among p processors. Each processor holds exactly one such slice at the end. A key idea of this algorithm is to quickly deliver to each processor the suffixes index corresponding to its slice. This algorithm works with p-percentiles obtained in one step and it has total time complexity of $O(n/p)$ communication in the average case.

Another parallel quick sort-based algorithm is presented in [10]. This algorithm is based on the recursive parallel quick sort approach, where a suitable pivot is found for the whole distributed set of suffixes and the partition phase redistributes the indexes of the suffix array so that each processor has only suffixes smaller or larger than the pivot. Instead of finding p-percentiles [6], this algorithm used binary recursive approach based on one pivot. This algorithm has a total time complexity of $O((n \log p) / p)$ communication in the average case.

Previous distributed algorithms are generalizations of general purpose sorting algorithm. However, much faster special purpose algorithm for suffix sorting can be adapted, because the suffixes are overlapping substrings coming from the same string. The first distributed algorithm which generalizes a special purpose sequential algorithm was presented in [9]. This algorithm was based on Manber & Myers [17] sequential algorithm. The algorithm achieved a good computation complexity but the communication complexity is worse than [6]. In the experimental analysis, they used 2-processor shared memory workstation to simulate a very fast network, yet they did not get a good communication time.

Recently, another parallel suffix sorting algorithm is presented in [13]. This algorithm is much more practical to implement and works well for typical input. The algorithm first partitions the suffixes into buckets, which are then assigned to individual processors for local sorting. Not much communication is involved in the sorting activity compared with the previous work. Once the bucket sorting is done, each bucket can be sorted independently. Since each bucket lies completely within a processor, no communication is necessary for local sorting. However in the bucket-sorting phase, they used All-Reduce and All-to-All collective communication operation provided by MPI standard. A communication model for collective MPI operations has been evaluated in [16]. If we consider our bandwidth ***B*** and size of the message ***S*** as key variables, we can obtain at least ***O(S/B)*** communication by considering latency and logarithmic time phase as constant parameter. ***S*** in this algorithm is equal to $|\sum|^{w}$, where ***w*** denote the first ***w*** characters involve in the bucket sorting. In the experimental analysis in [13], they evaluated their algorithm on **IBM SP-2** using genomes of several organisms. Their experimental result showed that their algorithm delivered good and scalable performance, since ***B*** is high (**IBM SP-2**) and $|\sum|$ is only 4 for any genome (i.e. {A, T, C, G}). The communication complexity depend on the size of alphabets, so no assumption can be make if the used platform is of low bandwidth network and the text with larger $|\sum|$ such as protein sequences, linguistic texts especially those written in Asian languages is used as input.

3. PARALLEL CONSTRUCTION OF SUFFIX ARRAYS

In this section we present our parallel algorithm for construct of suffix array. Our main idea is almost the same as [13], where the suffixes are partitioned into buckets and each bucket is further sorted by string sorting algorithm. However their bucket sorting involves communication while ours involve none at all.

Consider that each processor contains a string ***T***[1,***n***] and an integer array ***SA***[1, ***n***] of size ***n***. Since each processor has its own copy of ***T***, they can bucket-sort the text in parallel. Indexes of suffixes, which have the same first character, are put on a consecutive area in ***SA***. We do not distribute the bucket-sorting task among the processors. Every processor does its own bucket sorting. Some computational overheads do exist here, but it is worth, because no communication is involved and it just needs insignificant period of time compared to its total sorting time. This small bucket-sorting time will be a constant while the number of processors is increased. We still can gain good speedup by parallel the local sorting in the next phase.

Once the bucket sorting is completed, the suffix array has been partitioned into buckets. Now the problem remains to find out how each processor calculates the buckets belonging to them. Note that each processor owns a whole set of suffixes and buckets information, so no communication is needed between cooperating processors in the buckets assigning task.

Any load balancing heuristic can be used to allocate the buckets to processors. Our implementation uses the following simple scheme, which is similar, as [13]. First we partition the bucket-sorted ***SA*** into ***p*** slices of length exactly ***n/p***. If a partition falls within a bucket, we readjust the partition so that it coincides with the nearest bucket boundary. Since each processor needs only two boundary indexes, the indexes for starting and ending of the partition belong to them, they just need to calculate this two values in parallel, still no communication is needed.

At this stage, the array ***SA*** has been partitioned across processors such that each bucket lies completely within a processor. This guarantees that all suffixes within a processor are only lexicographically larger or smaller than any suffix which is allocated in the difference processors. The suffixes belonging to each partition are sorted locally within each processor. Every processor locally sorts their partition concurrently. We use Bentley and Sedgewick's [8] "*on fast algorithms for sorting strings*" for our local sorting. Any special purpose suffix sorting algorithms is not suitable to be used for low bandwidth network, since each consecutive suffix has been distributed lexicographically across processors by bucket sort, a lot communication is needed.

Execution time for the parallel algorithm is affected by load balancing among processors. However, load balancing for this problem is mainly affected by the input data, not only the partition size of ***SA*** among the processors but also the average match length (AML) [12] of the suffixes within one processor to another. However, our algorithm should perform well for typical random strings, since it expects the length of prefix sufficient to differentiate between two suffixes is O(log n) [13]. In the next section, we present some experimental analysis of our algorithm on representative data set drawn from Calgary Text Compression Corpus [2].

4. EXPERIMENTAL RESULTS

We implemented our parallel algorithm using **C** and **PVM**. For input texts with $|\Sigma| < p$,we transform the input text according to the algorithm provided in [14]. This transformation is to ensure that there are enough buckets to assign to each processor. We tested the program on following data: Complete genome of *E.coli* (length 4, 638, 690 bases), linguistic texts; bible.txt (4, 047, 392 characters), world192.txt (2,473,400 characters) and combine.txt (6520792 characters, *concatenation of bible.txt and world192.txt*)

Our timing is based on the slowest processors. The timing is terminated using barrier function provided by PVM.

First, we experimentally evaluate our program on cluster of SUN Workstation with 10 bases hub. Figure 1 shows our algorithm works well and almost obtained a linear speedup.

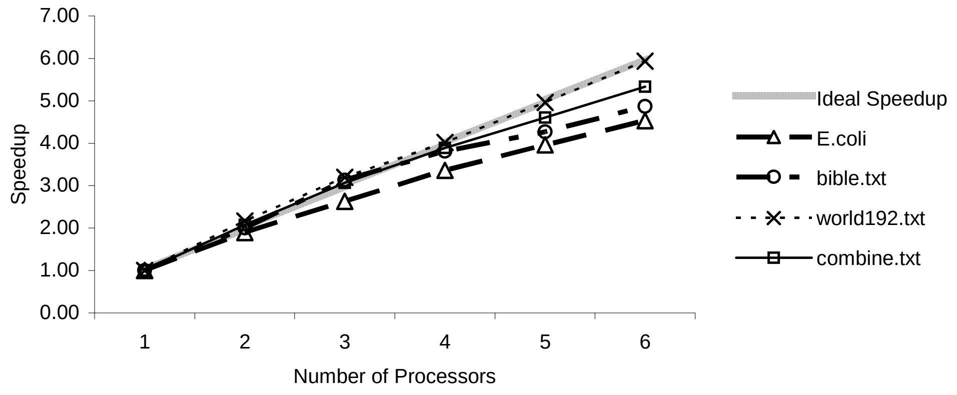
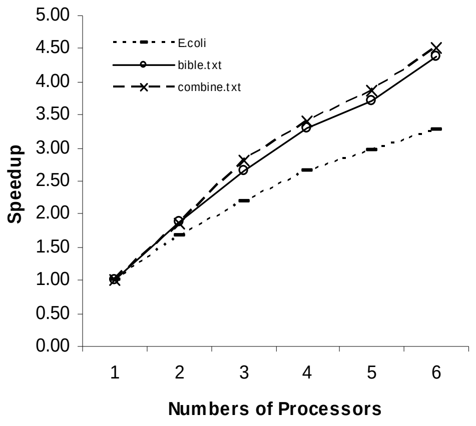


Figure 1: Speedup as a function of the number of processors for difference data

For our second experiment, we run our program on PC-cluster of Intel Pentium IV (1.8GHz). First, we connected the PC using 3com Switch 24-port with 100 base and for the second experiment we connected this cluster of PC using 3com Hub 8-port with 10 base.



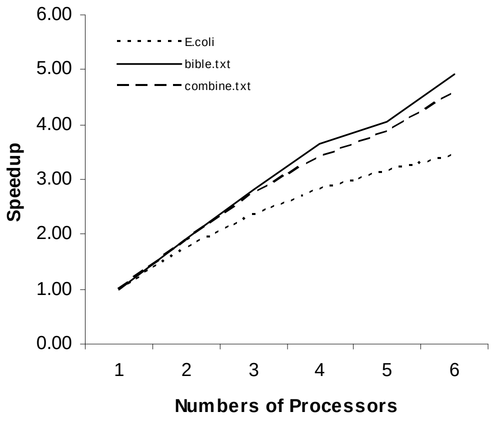


Figure 2: Speedup as a function of the number of processors for difference data on high bandwidth network (100Mbps Switch)

Figure 3: Speedup as a function of the number of processors for difference data on low bandwidth network (10Mbps hub)

From the figures 2 and 3, we get almost linear speedup for both different bandwidth of connection are obtained. This indicates our algorithm is not platform dependent and work well even in low bandwidth network connection. Figure 4 shows that we obtained almost overlapping speedup on both bandwidths using combine.txt as the input data.

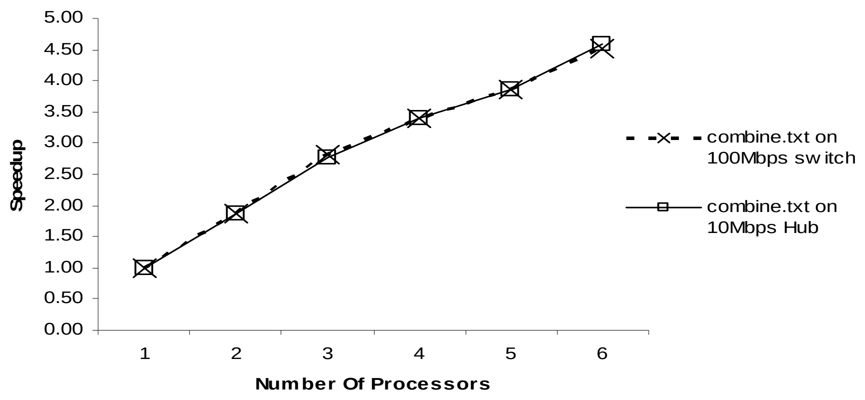


Figure 4: Speedup as a function of the number of processors for combine.txt text data on PC-cluster with 10Mbps hub and 100Mbps switch connection.

5. CONCLUSIONS

We presented a simple parallel algorithm for suffix arrays construction. We have created a fast implementation of our algorithm using Bentley & Sedgewick's code and N.J. Larsson & K.Sadakane's code. We have experiment our program using data set downloaded from [2]. The evaluation of our program on low bandwidth network proved to produce similar results to that obtained on high bandwidth network, hence indicating that the parallel algorithm is platform independent.

6. REFERENCES

A. Macêdo, M.A.Cristo, E.S.Silva, D.M.Barbosa, J.Kitajima, B.Ribeiro, G.Navarro and N. Ziviani.1998 .Experimental Analysis of Parallel Quicksort-Based Algorithm for suffix Array Generation .*Proceedings of 3rd International Meeting on Vector and Parallel Processing (VECPAR'98),* pages 1049-1062 [1]

Calgary Text Compression Corpus. (online)ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/ (5 June 2003) [2]

D. Gusfield, 1997. Algorithms on Strings, Trees, and Sequences- Computer Science and Computational Biology. Cambridge University Press. [3]

D. Knuth, 1973. The Art of Computer Programming, Vol 3: Sorting and Searching. Addison-Wesley. [4]

G. Manzini and P.Ferragina, Aug 2002. Engineering a Lightweight Suffix Array Construction Algorithm. *Proceedings of the 10th European Symposium on Algorithms (ESA '02), Rome, Italy. Springer Verlag Lecture Notes in Computer Science n. 2461,* pages 698-710. [5]

G.Navarro, J.Kitajima, B.Ribeiro and N.Ziviani, June 1997. Distributed generation of suffix arrays. *Proceedings of the 8ht Symposium on Combinatorial Pattern Matching (CPM'97), LNCS 1264*, pages 102-115. [6]

H. Itoh and H. Tanaka, 1999. An efficient method for in Memory construction of suffix arrays. *Proceedings of the 6th Symposium on String Processing and Information Retrieval (SPIRE'99)*, pages 81-88. [7]

J.Bentley and R.Sedgewick, 1997. Fast Algorithms for Sorting and searching Strings. *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms,* pages 360-369.[8]

J. Kitajima, G.Navarro. A fast Distributed suffix Array generation Algorithm. *Proceedings of the 6th Symposium on String Processing and Information Retrieval (SPIRE'99),* pages 97-104 [9]

J. Kitajima, G.Navarro, B.Ribeiro and N.Ziviani, 1997. Distributed generation of suffix arrays: A quicksort-based approach. *Proceedings of 4th South American Workshop on String Processing (WSP'97),* pages 53-69. [10]

K. Sadakane, 1998. A fast algorithm for making suffix arrays and for burrows-wheeler transformation.*Proceedings of the IEEE, Data Compression Conference (DCC'98),* pages 129-138. [11]

K. Sadakane and H. Imai, 1998. Constructing Suffix Arrays Of large Texts transformation**.** *Proceedings of the IEICE 9th Data Engineering Workshop (DEWS '98)* [12]

N. Futamura, S. Aluru, and S. Kurtz, 2001. Parallel Suffix Sorting, *Proceedings of the 9th International Conference on Advanced Computing and Communications (ADCOM2001),* pages 76-81 [13]

N.J. Larsson and K.Sadakane,1999. Faster suffix sorting, *Technical Report in Department of Computer Science, Lund University, sweeden,* LU-CS-TR: pages 99-214. [14]

P.M. McIlroy and K. Bostic, 1993. Engineering radix sort. Computing Systems, 6(1): pages 5-27. [15]

S. Girona, J. Labarta and R.M. Badia, 2000. Validation of Dimemas communication model for MPI collective operations. *Proceedings of the Recent Advances in Parallel Virtual Machine and Message Passing Interface (Euro-PVM/MPI 2000,.* pages 39-46. [16]

U. Manber and Myers, Oct 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing, 22:* pages 935-948. [17]