Parallel Suffix Array Construction for Shared Memory Architectures*

Vitaly Osipov

Karlsruhe Institute of Technology, Germany
osipov@kit.edu

Abstract. We present the design of the algorithm for constructing the suffix array of a string using manycore GPUs. Despite of the wide usage in text processing and extensive research over two decades there was a lack of efficient algorithms that were able to exploit shared memory parallelism (as multicore CPUs as manycore GPUs) in practice. To the best of our knowledge we developed the first approach exposing shared memory parallelism that significantly outperforms the state-of-the-art existing implementations for sufficiently large inputs. We reduced the suffix array construction problem to a number of parallel primitives such as prefix-sum, radix sorting, random gather and scatter from/to the memory. Thus, the performance of the algorithm merely depends on the performance of these primitives on the particular shared memory architecture. We demonstrate its performance on manycore GPUs, but the method can also be applied for other parallel architectures, such as multicores, CELL or Intel MIC.

1 Introduction

The *suffix tree* of a string is a compact trie of all its suffixes. It is a powerful and widely used data structure with large variety of applications in such fields as stringology, computational biology and text search. The *suffix array* and methods for constructing it were proposed by Manber and Myers in 1990 [9] as a simple and space efficient alternative to suffix trees. It is simply the lexicographically sorted array of the suffixes of a string. Suffix tree and methods for its construction were involved in hundreds of papers over the last two decades.

There are three basic techniques for constructing suffix arrays [13] that we informally denote by *prefix-doubling*, *induced copying* and *recursion*. In short, *prefix-doubling* approaches iteratively sort the suffixes by their prefixes that double in length in each iteration. *Induced copying* algorithms sort a sample of the suffixes and use them to induce the order of the non-sample suffixes. *Recursion* methods recursively reduce the input string length in each iteration. Thus, the existing algorithms can be implicitly divided into three classes according to the technique they exploit. Besides those that can be classified into a single class there exist hybrid approaches that combine at least two of the basic techniques.

* Partially supported by EU Project No. 248481 (PEPPHER) ICT-2009.3.6 and DFG grant SA 933/3-2

L. Calderón-Benavides et al. (Eds.): SPIRE 2012, LNCS 7608, pp. 379–384, 2012.
© Springer-Verlag Berlin Heidelberg 2012

On the theoretical side both the induced copying and the recursion class contain linear time algorithms. The Ko and Aluru (KA) [5], Kärkkäinen and Sanders (KS) [4] as well as the recent Nong et al. (SAIS) [12] algorithms can be referred to the hybrid recursive approaches that use the induced copying technique. Though, the underlying ideas behind inducing are different. KA and SAIS use a sample of input-dependent suffixes (SL-inducing), while KS's choice of the sample is input-independent and is merely based on the regular suffix positions in the input string (DC-inducing).

In practice the algorithms based on SL-inducing outperform their DC-inducing counterparts. Moreover, for real-world instances supralinear $\mathcal{O}(n^2 \log n)$ algorithms are often faster [13]. As long as we are concerned with practical performance and do not insist on linearity, the $\mathcal{O}(n \log n)$ Larsson and Sadakane (LS) [7] algorithm becomes competitive. LS is based on the original prefix-doubling Manber and Myers (MM) [9] algorithm with a powerful filtering criterion on top that makes it significantly (by a factor of 10 or so [13]) faster in practice.

Parallel suffix array construction solutions exist in the distributed [3,6] as well as parallel external memory settings [1,2]. The most efficient of them are based on KS algorithm (DC-inducing). As for shared memory parallel SACAs, we see almost no progress in the area. The main reasons for that are: (1) all the fastest practical sequential algorithms based on the SL-inducing technique are difficult to parallelize; (2) the DC-inducing and prefix-doubling techniques involve large overheads making parallelization using a small number of cores of little (if at all) use. Thus, we either need better parallalizable approaches involving smaller overheads, or go beyond commodity multicore machines and design solutions that would scale well with the number of cores and hence compensate for the increased overhead.

One of such emerging manycore architectures is a GPU. Sun and Ma [16] attempted to design a SACA for GPUs. They implemented the original MM algorithm and compared it to its CPU counterpart on random strings. Though their GPU implementation demonstrated a speedup of up to 10 for sufficiently large inputs, the significance of the result is questionable since for real world data MM is proven to be more than an order of magnitude slower than the currently fastest SACAs [13]. Moreover, random strings having an average longest common prefix of length 4 are easy instances for MM.

2 Preliminaries

Let $x = x_1 x_2 \dots x_n$ be a finite nonempty string of length n, where letters belong to an *indexed* alphabet Σ. That is, an alphabet that can be mapped to an integer alphabet of a limited range. Our goal is to compute a *suffix array* SA_x, or SA for short, an integer array $\mathrm{SA}[1 \dots n]$, where $\mathrm{SA}[j] = i \Leftrightarrow x_i \dots x_n$ is the jth suffix in ascending lexicographical order. For convenience, we denote $x_i \dots x_n$ as a suffix i and append the string with a sentinel \$, which we assume to be less than any letter $\lambda \in \Sigma$. An *inverse suffix array* denoted as ISA_x, or ISA for short, is an integer array $\mathrm{ISA}[1 \dots n]$, such that $\mathrm{ISA}[i] = j \Leftrightarrow \mathrm{SA}[j] = i$.

Most SACAs proceed by ordering suffixes by their prefixes of increasing length $h \geq 1$, the process that we call h-sorting. The obtained partial order is denoted as *h-ordering* of suffixes into *h-order*. Suffixes that are equal in h-order are called *h-equal*. They have the same *h-rank* and belong to the same *h-group* of h-equal suffixes. If h-sort is stable, then the h-groups for a larger h "refine" the h-groups for a smaller h. To store a partial h-order, we use an *approximate suffix array* denoted as SA_h or an *approximate inverse suffix array* denoted as ISA_h.

3 Parallel Algorithm

Due to a lack of parallel approaches exploiting SL-inducing technique, the choice of the algorithm that would suit a manycore architecture boils down to the prefix-doubling or DC-inducing based methods. Unfortunately, better asymptotic behavior of KS algorithm alone does not guarantee the better performance on real world data [13]. Moreover, practical implementation of KS algorithm, requires sorting of large tuples (up to five 32-bit integers) using comparison based sorting and merging [6,2]. Though there exist efficient comparison based GPU sorting [8] and merging [14,15] primitives, their performance is still inferior to that of GPU radix sorting [8,10,14,15]. In contrast to KS, prefix-doubling algorithms (LS and MM) require radix sorting of (32-bit key, 32-bit value) pairs only.

Nevertheless, each of prefix doubling variants has drawbacks with respect to parallelization. LS requires simultaneous sorting of a (possibly) large number of various-size chunks of data that makes load balancing difficult, while MM induces large overheads by re-sorting suffixes whose final positions in the SA are already defined.

In our approach we modify MM in a way that requires a single radix sort of (32-bit key, 64-bit value) pairs, where LS would sort independent chunks of (32-bit key, 32-bit value) pairs. On the other hand, we use the following filtering criterion that allows our approach to avoid extensive re-sorting similar to LS.

Observation 1. *If in the k-th iteration of the MM algorithm: (1) suffix $i = SA_{2^k}[j]$ forms a singleton 2^k-group; (2) $i < 2^{k+1}$ or suffix $i - 2^{k+1}$ also forms a singleton 2^k-group, then for all further iterations $j > k$ either $i < 2^j$ or suffix $i - 2^j$ forms a singleton 2^j-group.*

The Algorithm 1 contains a high-level description of our approach. The procedure sorts all suffixes by 4 characters initially and initializes the corresponding approximate SA_4 and ISA_4. Further, the algorithm proceeds in phases until all suffixes get sorted. It generates tuples containing suffix index $i - h$ accompanied with its h-rank and h-rank of the suffix i. By stable sorting of these tuples by h-rank of $i - h$ we obtain SA_{2h}. The h-rank of i stored in tuples allows us to refine h-groups and, thus, get ISA_{2h}. Finally, we filter SA_{2h} using Observation 1 and compact it accordingly.

It is not difficult to see that the proposed algorithm allows its reduction to a number of widely-used parallel primitives. Indeed, steps 1 and 6 involve radix sorting of integer tuples. Refinement of h-heads and compaction (lines 8 and 11)

```
1  initialize SA_4 by sorting suffixes by their first 4 characters
2  initialize ISA_4[i] with the 4-rank of i = head of i's 4-group in SA_4
3  size = n, h = 4
4  while size > 0 do
5  |   Scan SA_h and generate tuples (SA_h[j] − h, ISA_h[SA_h[j] − h], ISA_h[SA_h[j]])
6  |   Sort tuples by 2nd component stably /* contains SA_2h          */
7  |
8  |   Refine h-heads of h-groups
9  |   Update ISA_h /* contains ISA_2h                                 */
10 |
11 |   Filter and Compact SA_2h
12 |   size = size of SA_2h
13 |   h = h * 2
14 end
```

Algorithm 1: high-level description of the algorithm

can be implemented using prefix-sum operation. While updating and filtering (lines 9 and 11) involves random gather and scatter from and to the memory.

We should mention that the running time was our primary goal. Therefore, our implementation is not particular lightweight in memory consumption and requires for a string of length n a total of $32n$ byte storage in GPU memory.

4 Performance Evaluation

Our experimental platform is an Intel i7 920 2.67 GHz quad-core machine with 6 GB of memory. We used a commodity NVidia Fermi GTX 480 featuring 15 multiprocessors, each containing 32 scalar processors, for a total of 480 CUDA cores on chip. The GPU RAM is 1.5 GB. We compiled all implementations using CUDA 4.1 RC 2 and Microsoft Visual Studio 2010 on 64-bit Windows 7 Enterprise with maximum optimization level.

We do not include the time for transferring the data from host CPU memory to GPU memory as suffix array construction is often a subroutine in a more complex algorithm. Therefore, we expect applications to reuse the constructed data structure for the further processing on GPU.

We performed the performance analysis on widely used benchmark sets of files including Calgary Corpus, Canterbury Corpus, Large Canterbury Corpus, Manzini's Large Corpus, Maximum Compression Test Files, Protein Corpus and The Gauntlet [11]. Due to the GPU memory capacity and the memory requirements of our implementation we include into the benchmark strings of size at most 45 MB.

In Figure 1 (left) we show the relative speedup of our implementation over the original LS Algorithm [7]. For instances under 10^5 characters the GPU performance is inferior to the serial CPU implementation. Such short instances are not capable to saturate the hardware and efficiently exploit available parallelism. On the other hand, the CPU is able to realize the full potential of its cache that fits the whole input.

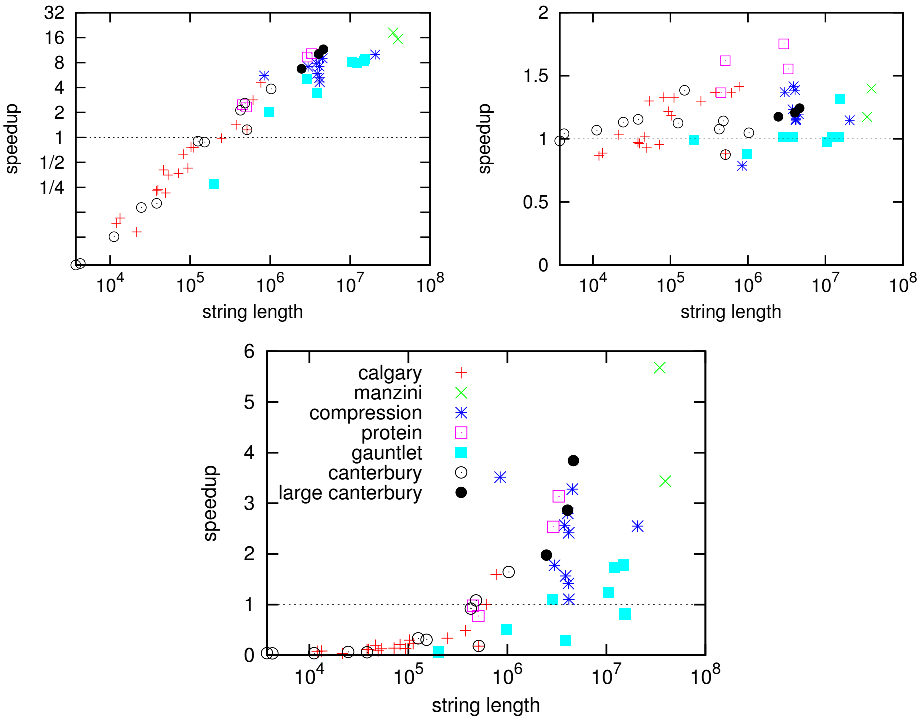

Fig. 1. The relative speedup of GPU SACA compared to serial LS algorithm (left), 4-core divsufsort compared to its sequential version (right) and GPU SACA compared to 4-core divsufsort (bottom)

Though for larger instances our implementation achieves a considerable speedup of up to 18 over its sequential counterpart. Sufficiently small fluctuations in speedup for approximately equally sized instances suggest that the behavior of our MM variant is similar to LS, thus showing efficiency of our filtering criterion.

We also compare the performance of our implementation to Yuta Mori's highly tuned, OpenMP assisted CPU implementation divsufsort 2.01 [11] using 4 cores, see Figure 1 (bottom). Being not a fully parallel algorithm, divsufsort scales suboptimally with the number of processors, see Figure 1 (right).

We observe, that the relative speedup fluctuates significantly depending on the instance. This is due to different techniques that are used in the algorithms. For example, three instances that are simply multiple concatenation of some string from the Gauntlet set are still faster on a CPU. The reason is, that these are the most difficult inputs for prefix doubling algorithms. The filtering criterion is also of little help here, since most of the suffixes get fully sorted only on the last few iterations of the algorithm. While for the class of induced copying algorithms, which includes divsufsort, this kind of instances are not particularly hard.

Nevertheless, our implementation achieves a speedup of up to 6 for the majority of significantly large instances.

References

1. Beckmann, A., Dementiev, R., Singler, J.: Building a parallel pipelined external memory algorithm library. In: Proc. Int'l Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–10 (May 2009)
2. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. J. Exp. Algorithmics 12, 3.4:1–3.4:24 (2008)
3. Futamura, N., Aluru, S., Kurtz, S.: Parallel suffix sorting. Electrical Engineering and Computer Science, paper 64 (2001)
4. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)
5. Ko, P., Aluru, S.: Space Efficient Linear Time Construction of Suffix Arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 200–210. Springer, Heidelberg (2003)
6. Kulla, F., Sanders, P.: Scalable parallel suffix array construction. Parallel Computing 33(9), 605–612 (2007)
7. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Theoretical Computer Science 387(3), 258–272 (2007)
8. Leischner, N., Osipov, V., Sanders, P.: GPU sample sort. In: Proc. of the IEEE Int'l Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–10 (April 2010)
9. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. In: Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Philadelphia, PA, USA, pp. 319–327 (1990)
10. Merrill, D., Grimshaw, A.S.: High performance and scalable radix sorting: a case study of implementing dynamic parallelism for gpu computing. Parallel Processing Letters 21(2), 245–272 (2011)
11. Mori, Y.: Suffix array construction algorithms benchmark set, `http://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks`
12. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure Induced-Sorting. In: Proc. of Data Compression Conference (DCC), pp. 193–202. IEEE (March 2009)
13. Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2) (July 2007)
14. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore gpus. In: Proc. Int'l Symposium on Parallel & Distributed Processing, IPDPS (2009)
15. Satish, N., Kim, C., Chhugani, J., Nguyen, A.D., Lee, V.W., Kim, D., Dubey, P.: Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In: Proc. of the Int'l conference on Management of Data, pp. 351–362. ACM (2010)
16. Sun, W., Ma, Z.: Parallel lexicographic names construction with CUDA. In: Proc. of the 15th International Conference on Parallel and Distributed Systems (ICPADS), pp. 913–918 (December 2009)