Scalable Parallel Suffix Array Construction

Fabian Kulla and Peter Sanders

Universität Karlsruhe, 76128 Karlsruhe, Germany
sanders@ira.uka.de

Abstract. Suffix arrays are a simple and powerful data structure for text processing that can be used for full text indexes, data compression, and many other applications in particular in bioinformatics. We describe the first implementation and experimental evaluation of a scalable parallel algorithm for suffix array construction. The implementation works on distributed memory computers using MPI, Experiments with up to 128 processors show good constant factors and make it look likely that the algorithm would also scale to considerably larger systems. This makes it possible to build suffix arrays for huge inputs very quickly. Our algorithm is a parallelization of the linear time DC3 algorithm.

1 Introduction

The suffix array [1,2], a lexicographically sorted array of the suffixes of a string, has numerous applications, e.g., in string matching [1,2], genome analysis [3] and text compression [4]. For example, one can use it as full text index: To find all occurrences of a pattern P in a text T, do binary search in the suffix array of T, i.e., look for the interval of suffixes that have P as a prefix. A lot of effort has been devoted to efficient construction of suffix arrays, culminating recently in three direct linear time algorithms [5,6,7]. One of the linear time algorithms, *DC3* [8] is very simple and can also be adapted to different models of computation. An external memory version of the algorithm [9] already makes it possible to construct suffix array for huge inputs. However, this takes many hours and hence a scalable *parallel algorithm* might be more interesting. This is the subject of the present paper. We describe the algorithm, *pDC3*, in Section 2 and experimental results in Section 3. Section 4 concludes with an outline of possible additional questions.

Related Work

There are numerous theoretical results on parallel suffix *tree* construction (e.g., refer to the references given in [10,11]). Suffix trees can be easily converted to suffix arrays. However, these algorithms are fairly complicated. We are not aware of any implementations. Recently, a trend is to use simpler suffix array construction algorithms even as a means of constructing suffix trees. Parallel conversion algorithms are described in [10]. The basic ideas for parallel suffix array construction based on the DC3 algorithm are already given in [8,11] for several theoretical models of parallel computation. Here, we concentrate on the detailed description of a practical

B. Mohr et al. (Eds.): PVM/MPI 2006, LNCS 4192, pp. 22–29, 2006.

algorithm with particular emphasis on implementation and experimental evaluation. We are only aware of a single implemented parallel algorithm for suffix array construction [12]. This algorithm is practical but based on string sorting and thus needs quadratic work in the worst case. From experiments with sequential algorithms, it is also known that algorithms based on string sorting are not very fast even for some real world inputs with long common prefixes (e.g. [13]). Furthermore it seems that all processing elements (PEs) need access to the complete input. This is an impediment for scaling to large numbers of PEs and large inputs since there might not be enough space on distributed memory machines and since this implies an execution time of $\Omega(n)$, i.e., the maximal speedup is bounded by a constant independent of the number of PEs.

2 The pDC3 Algorithm

We use the shorthands $[i,j] = \{i,\ldots,j\}$ and $[i,j) = [i,j-1]$ for ranges of integers and extend to substrings as seen below. The **input** of a suffix array construction algorithm is a *string* $T = T[0,n) = t_0 t_1 \cdots t_{n-1}$ over the alphabet $[1,n]$, that is a sequence of n integers from the range $[1,n]$. For convenience, we assume that $t_j = 0$ for $j \geq n$. For $i \in [0,n]$, let S_i denote the *suffix* $T[i,n) = t_i t_{i+1} \cdots t_{n-1}$. We explain the algorithm using pseudocode manipulating sequences of tuples. For example, for $T = abcdef$, $\langle (T[i,i+2],i) : i \bmod 3 = 0\rangle$ denotes $\langle (abc,0),(def,3)\rangle$. The goal is to sort the sequence $\langle S_0,\ldots,S_n\rangle$ of suffixes of T, where comparison of substrings or tuples assumes the lexicographic order throughout this paper. The **output** is the *suffix array* $SA[0,n)$ of T, a permutation of $[0,n)$ satisfying $S_{SA[0]} < S_{SA[1]} < \cdots < S_{SA[n-1]}$. Let p denote the number of processors (PEs). PEs are numbered from 0 to $p-1$.

At the most abstract level, the DC3 Algorithm is very simple and completely independent of the model of computation: It first constructs the suffix array of the suffixes starting at positions $i \bmod 3 \neq 0$. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively. Then this information is used to annotate the original input. With this annotation, two *arbitrary* suffixes S_i and S_j can be compared by looking at $T[i,i+2]$ and the annotations at positions $[i,i+2]$. For a more detailed explanation refer to [11].

Fig. 1 gives a more detailed pseudocode which exposes parallelism and which we will then refine to the full parallel algorithm. Line 1 extracts the information needed for building the recursive subproblem which consists of two concatenated substrings of length $n/3$ representing the mod1 suffixes and mod2 suffixes respectively. This length reduction is achieved by finding *lexicographic names* for triples of characters, i.e., integers that reflect the lexicographic order of these character triples. To find these names, the triples (annotated with their position in the input) are sorted in Line 2 and named in Line 3 using a subroutine to be discussed. If all triples are unique, no recursion is necessary (Line 4). Otherwise, Line 5 assembles the recursive subproblem, Line 6 solves it, and Line 7 brings it into a form compatible with the output of the naming routine. Line 8 permutes

Function *pDC3*(T)
S:= $\langle((T[i, i+2]), i) : i \in [0, n), i \bmod 3 \neq 0\rangle$ 1
sort S *by the first component* 2
P:= *name*(S) 3
if *the names in* P *are not unique* **then** 4
permute the $(r, i) \in P$ *such that they are sorted by* $(i \bmod 3, i \operatorname{div} 3)$ 5
SA^{12}:= *pDC3*$(\langle c : (c, i) \in P\rangle)$ 6
P:= $\left\langle (j+1, SA^{12}[j]) : j \in [0, 2n/3)\right\rangle$ 7
permute P *such that it is sorted by the second component* 8
S_0:= $\langle(T[i], T[i+1], c', c'', i) : i \bmod 3 = 0, (c', i+1), (c'', i+2) \in P\rangle$ 9
S_1:= $\langle(c, T[i], c', i) \quad : i \bmod 3 = 1, (c, i), (c', i+1) \in P\rangle$ 10
S_2:= $\langle(c, T[i], T[i+1], c'', i) \; : i \bmod 3 = 2, (c, i), (c'', i+2) \in P\rangle$ 11
S:= *sort* $S_0 \cup S_1 \cup S_2$ *using comparison function:* 12
$(c, \ldots) \in S_1 \cup S_2 \leq (d, \ldots) \in S_1 \cup S_2 \Leftrightarrow c \leq d$
$(t, t', c', c'', i) \in S_0 \leq (u, u', d', d'', j) \in S_0 \Leftrightarrow (t, c') \leq (u, d')$
$(t, t', c', c'', i) \in S_0 \leq (d, u, \quad d', j) \in S_1 \Leftrightarrow (t, c') \leq (u, d')$
$(t, t', c', c'', i) \in S_0 \leq (d, u, u', d'', j) \in S_2 \Leftrightarrow (t, t', c'') \leq (u, u', d'')$
return $\langle$*last component of* $s : s \in S\rangle$ 13

Fig. 1. High level pseudo code for pDC3

the resulting tuples into the order of the input string. Now, Lines 9–11 use the input string and the result of the recursion to build 5-tuples and 4-tuples that contain all the information needed to compare the suffixes they represent. These are sorted in Line 12. Line 13 extracts the suffix array from the result.

The basic idea behind parallelization is that input, output, and intermediate tuple sequences are uniformly or almost uniformly distributed over all PEs. Lines 1,7, 9–11, and 13 are then straight forward to parallelize. The only necessary communication is between PE i and PE $i+1$ to retrieve values that are one or two places to the right in the sequence currently processed. Permutations (Lines 5 and 8) are mapped to personalized all-to-all communications with variable message lengths but balanced or almost balanced total communication volume at each PE. Sorting (Lines 2 and 12) can be implemented using any parallel sorting algorithm. The naming step in Line 3 is interesting since its sequential implementation scans S assigning a fresh name to any new triple found. On the first glance this looks inherently sequential. However consider replacing the naming step by the following two lines.

Δ:= $\langle[S[i] \neq S[i+1]] : 0 \leq i < 2n/3\rangle$
P:= $\left\langle(1 + \sum_{j<i} \Delta[j], i) : 0 \leq i < 2n/3\right\rangle$

The first line is a simple local computation. The second line computes a *prefix sum*, an operation easily done in time $\mathcal{O}(n/p + \log p)$. Finally, to implement Line 4 in Fig. 1, PE $p-1$ just needs to check whether the total sum over Δ was n and broadcast this information to all PEs.

This level of abstraction is the most appropriate for an analysis of the algorithm.

Theorem 1. *The suffix array of a string of size n can be computed in time $\mathcal{O}(T_{\text{parsort}}(n,p) + T_{\text{allall}}(n/p,p) + f(p)\log(n))$ where $T_{\text{parsort}}(n,p)$ is a bound on the execution time of sorting n elements on p processors with the property $T_{\text{parsort}}(2n/3,p) \leq \frac{2}{3}T_{\text{parsort}}(n,p) + f(p)$ and $T_{\text{allall}}(\ell,p)$ is a bound on the execution time of personalized all-to-all communication such that no PE sends or receives more than ℓ words of data with the property that $T_{\text{allall}}(2\ell/3,p) \leq \frac{2}{3}T_{\text{allall}}(\ell,p) + f(p)$.*

The term $f(p) \in \Omega(\log p)$ in Theorem 1 is a bottleneck term that does not decrease when the input size decreases.

Proof. (Outline) The algorithm goes through $\mathcal{O}(\log n)$ levels[1] of recursion. The involved data volumes are decreasing geometrically. Thus, up to constant factors, we can bound the total execution time of sorting, all-to-all, and local operations by the cost of the first level of recursion, plus $\mathcal{O}(\log n)$ times the bottleneck term $f(p)$. Further communication operations all take time $\mathcal{O}(\log p) = \mathcal{O}(f(p))$ in each level of recursion. ∎

The usual implementation of all-to-all directly delivers all messages to their destination. It has $T_{\text{allall}}(\ell,p) = \mathcal{O}(\ell T_{\text{byte}} + pT_{\text{start}})$ on a machine with full interconnection network and time $kT_{\text{byte}} + T_{\text{start}}$ for point-to-point communication of a message of size k.

In our implementation we have

$$T_{\text{parsort}}(n,p) = \mathcal{O}\big((n/p + p^2)\log p\big) + T_{\text{allall}}(n/p,p)$$

using a simple variant of comparison based sample sort [14]: The input is first sorted locally. Each PE takes $\mathcal{O}(p)$ sample elements. The sample is gathered and sorted at a single PE. The sorted samples are used to obtain splitter elements $s_1,\ldots,s_{p-1}$ that are equally spaced in the sorted sample. These splitters are broadcast to all other PEs. Define $s_0 = -\infty$ and $s_p = +\infty$. Now each processor partitions the elements into buckets where the i-th buckets gets elements between s_i and s_{i+1}. All Elements from bucket i are then sent to PE i using an all-to-all personalized communication. Finally, each PE merges the received pieces of its bucket. In summary, sorting is reduced to local sorting, multiway merging, and further standard communication operations: gather of a small sample, splitter broadcast, and a single personalized all-to-all communication.

We get a bottleneck term of $f(p) = \mathcal{O}\big(p^2\log p + p^2T_{\text{byte}} + pT_{\text{start}}\big)$ and a total execution time of

$$\mathcal{O}\big((n/p\log p + (p^2(\log p + T_{\text{byte}}) + pT_{\text{start}})\log n\big)$$

[1] One can get a slight improvement of the theoretical bound by switching to a sequential algorithm after $\mathcal{O}(\log p)$ levels of recursion. But this is irrelevant from a practical perspective.

Asymptotically better bounds are obtained in [11] using more sophisticated implementations of sorting and all-to-all. However, these algorithms are considerably more complicated and in Section 3 we will give evidence that on machines with a moderate number of processors no significant improvements can be expected from these theoretical algorithms.

All the required communication operations (point-to-point, prefix sum, broadcast, all-to-all, gather) are available in communication libraries such as MPI [15].

3 Experiments

We have implemented pDC3 with deterministic sample sort using C++ and MPI [15]. Most measurements were performed on a HP Integrity rx2620 running under Linux with 64 dual processor nodes using Itanium 2 processors with 1.5 GHz and 6 MByte Cache. The machine has 64×12 GByte of main memory. The nodes are connected by a Quadrics QSNet II network with 800 MByte/s communication bandwidth.

We have used the big real world inputs from [9]: The human genome, 3.125 GByte of books from the Gutenberg project, and 522 MByte of source code. In addition, we use the artificial inputs a^n and $(abc)^{n/3}$. Timing is started when all PEs have initialized MPI and hold n/p characters of the input each.

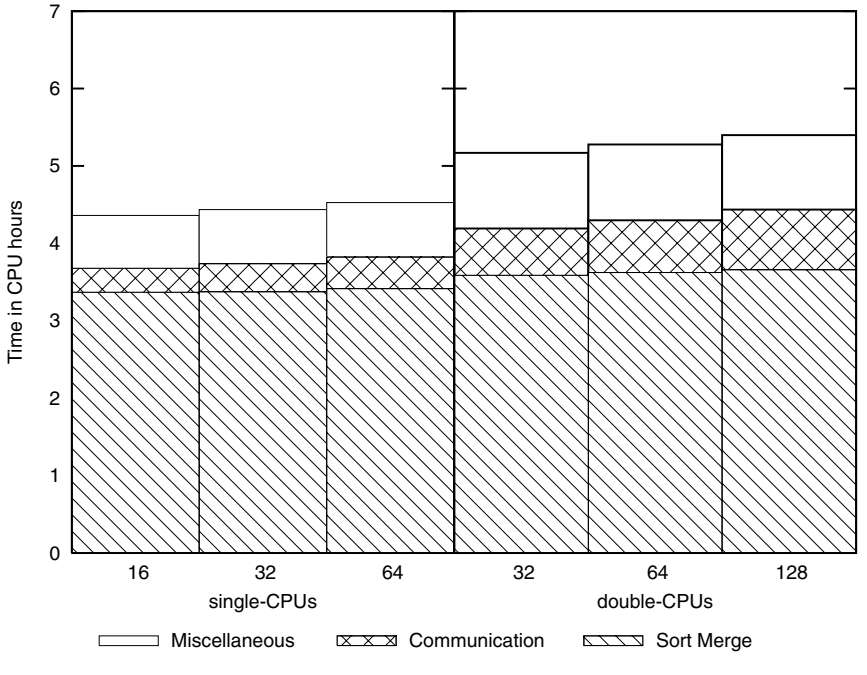

Fig. 2. The distribution of the execution time between sorting, communication and the remaining local operations for the Gutenberg instance

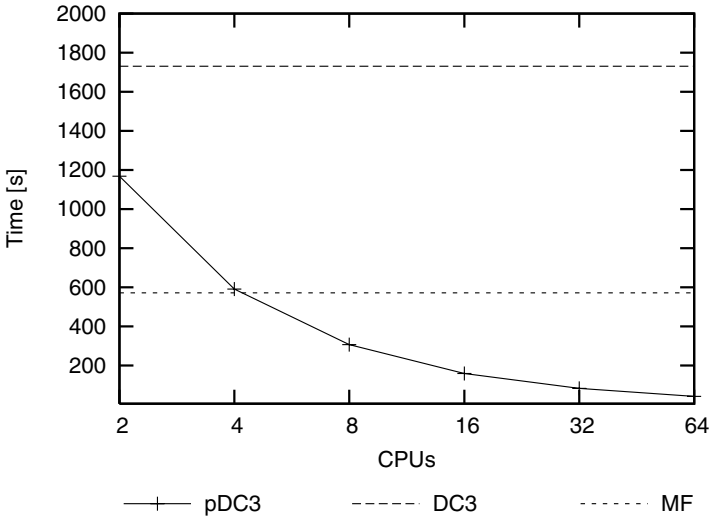

Fig. 3. Execution time of pDC3 compared to the sequential DC3 algorithm from [8] and to the sequential algorithm from [13]

Fig. 2 shows the work performed for the Gutenberg input using 16–128 PEs using one or two CPUs on each node. We see that sorting and merging takes most of the time. Communication time (mainly all-to-all) takes only a small fraction of the total execution time. However, it should be noted that low cost machines with switched Gigabit Ethernet have an order of magnitude smaller communication bandwidth than our machine. On such machines, communication would take about half of the time. (Which might still be acceptable considering that such machines are much cheaper). The overall work increases only slightly when increasing the number of processors. This indicates good scalability. As to be expected, using both CPUs increases internal work and total communication time since the CPUs have to share the main memory and the network interface.

We cannot give speedups for big inputs since no single node has enough memory to solve the problem. Therefore Fig. 3 compares pDC3 with two sequential algorithms for the source code instance. DC3 is the simple sequential linear time implementation from [5].[2] MF is one of the fastest practical algorithms [13]. With the minimal number of two processors our parallel algorithm already outperforms the simple sequential algorithm significantly although it has a factor $\Theta(\log n)$ disadvantage in its asymptotic execution time. The break even point to [13] is at four processors. The work per processor is about half as much as for the external algorithm from [9] on a 2GHz Intel Xeon processor. Unfortunately, a direct comparison with the parallel implementation from [12] is not possible since this paper does not specify the clock speed of the machine used.

[2] There are faster sequential implementations of DC3 by now [16] but they still do not beat implementations such as [13].

Table 1. Average (Ø) versus bottleneck (max) execution times of major parts of pDC3. Timings in second. Top part: 64×1 CPU. Bottom part: 64×2 CPUs.

Input	Size	Total	quicksort		mergesort		p-merge		All2all	Com	sample
			max	Ø	max	Ø	max	Ø	Ø	Ø	
Source	522	37.8	16.6	15.9	28.6	27.9	10.5	9.6	4.2	0.14	0.29
Genome	2928	282.0	160.3	115.0	182.6	178.7	62.8	58.0	22.2	0.36	1.24
Gutenberg	3125	254.6	124.0	119.5	197.4	195.6	68.1	66.5	22.2	0.36	1.30
a^n	3815	520.7	411.4	271.3	168.9	165.7	49.6	32.1	22.2	0.42	2.16
a^n	2000	259.7	202.2	130.6	85.2	83.4	25.8	16.6	11.5	0.37	1.78
$(abc)^{n/3}$	2000	263.7	198.2	98.5	85.2	83.2	33.3	16.4	13.8	0.38	1.54
Source	522	24.2	7.8	7.4	14.9	14.4	6.2	5.3	4.9	0.23	0.37
Genome	2928	180.8	94.3	53.8	99.6	95.7	39.0	37.2	21.9	0.67	1.25
Gutenberg	3125	151.8	58.7	55.8	107.5	105.7	44.2	40.9	21.1	0.53	1.31
a^n	3815	280.9	193.1	120.0	99.0	96.1	45.0	26.6	21.2	0.91	2.12
a^n	2000	140.7	93.4	56.9	49.4	47.8	23.2	13.7	11.2	0.53	1.76
$(abc)^{n/3}$	2000	146.1	92.3	42.7	49.5	47.8	30.9	13.5	13.4	0.56	1.49

Table 1 gives a more detailed breakdown of the execution time of pDC3 for different inputs. The STL quicksort used for local sorting shows considerable load imbalances, i.e., the slowest PE does much more work than the average PE. This is not due to significantly different amounts of data assigned to PEs but because quicksort has highly data dependent execution times in particular for the artificial inputs like a^n. In contrast, if we use mergesort, there is much less load imbalance. Here, the artificial inputs turn out to be *easier* to solve than the real world inputs. There is also some load imbalance for the p-way merging in sample sort for artificial inputs. However, this is not very critical since it only means that some PEs do less work than in the worst case.

4 Conclusions

We have demonstrated that pDC3 is a practicable and scalable way to build huge suffix arrays. Several practical improvements could be considered. pDC3 might scale even to machines with thousands of processors if we use parallel sorting for sorting the sample. The DC3 algorithm can be generalized to larger difference covers that imply a different recurrence relation. Using this scheme in the first level of recursion could save a constant factor of time for small alphabets. A $\log n$ term in the execution time could be removed by switching from comparison based sorting to integer sorting. However, we are not aware of an algorithm that would really remove the $\log n$ in the worst case and would bring improvements in practice. For example, the implementation from [8] gets slightly *faster* when its linear time sorting algorithm is replaced by quicksort. There are also further opportunities for tuning. For inputs that are so large that they do not even fit in the main memory of a parallel computer, a parallel external algorithm could be developed by combining the results of the present paper with [9].

References

1. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing **22** (1993) 935–948
2. Gonnet, G., Baeza-Yates, R., Snider, T.: New indices for text: PAT trees and PAT arrays. In Frakes, W.B., Baeza-Yates, R., eds.: Information Retrieval: Data Structures & Algorithms. Prentice-Hall (1992)
3. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: The enhanced suffix array and its applications to genome analysis. In: Proc. 2nd Workshop on Algorithms in Bioinformatics. Volume 2452 of LNCS., Springer (2002) 449–463
4. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report 124, SRC (digital, Palo Alto) (1994)
5. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proc. 30th International Conference on Automata, Languages and Programming. Volume 2719 of LNCS., Springer (2003) 943–955
6. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Proc. 14th Annual Symposium on Combinatorial Pattern Matching. LNCS, Springer (2003) 186–199 To appear.
7. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Proc. 14th Annual Symposium on Combinatorial Pattern Matching. Volume 2676 of LNCS., Springer (2003) 200–210
8. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: 30th International Colloquium on Automata, Languages and Programming. Number 2719 in LNCS (2003) 943–955
9. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. In: Workshop on Algorithm Engineering & Experiments, Vancouver (2005) 86–97
10. Iliopoulos, C.S., Rytter, W.: On parallel transformations of suffix arrays into suffix trees. In: 15th Australasian Workshop on Combinatorial Algorithms (AWOCA). (2004)
11. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. Journal of the ACM (2006) to appear.
12. Futamura, N., Aluru, S., Kurtz, S.: Parallel suffix sorting. In: Proc. 9th International Conference on Advanced Computing and Communications, Tata McGraw-Hill (2001) 76–81
13. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. In: Proc. 10th Annual European Symposium on Algorithms. Volume 2461 of LNCS., Springer (2002) 698–710
14. Shi, H., Schaeffer, J.: Parallel sorting by regular sampling. Journal of Parallel and Distributed Computing **14** (1992) 361–372
15. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI – the Complete Reference. MIT Press (1996)
16. Smyth, B., Turpin, A.: The performance of linear time suffix sorting algorithms. In: IEEE Data Compression Conference. (2005)