SACABench: Benchmarking Suffix Array Construction

Johannes Bahne[1], Nico Bertram[1], Marvin Böcker[1], Jonas Bode[1], Johannes Fischer[1], Hermann Foot[1], Florian Grieskamp[1], Florian Kurpicz[1], Marvin Löbel[1], Oliver Magiera[1], Rosa Pink[1], David Piper[1], and Christopher Poeplau[1]

Technische Universität Dortmund, Department of Computer Science, Germany
{`firstname.lastname`}`@tu-dortmund.de` (where ö is *oe*) and
`johannes.fischer@cs.tu-dortmund.de`

Abstract. We present a practical comparison of suffix array construction algorithms on modern hardware. The benchmark is conducted using our new benchmark framework *SACABench*, which allows for an easy deployment of publicly available implementations, simple plotting of the results, and straight forward support to include new construction algorithms. We use the framework to develop a construction algorithm running on the GPU that is competitive with the fastest parallel algorithm in our test environment.

Keywords: Suffix Array · Practical Survey · Text Indexing.

1 Introduction

The suffix array (SA) [28] is one of the most versatile and well-researched full-text indices. Given a text T of length n, the SA is the permutation of [1,n], such that $T[\mathsf{SA}[i]..n] < T[\mathsf{SA}[i+1]..n]$ for all $i \in [1, n-1]$, i. e., the starting positions of all suffixes of the text in lexicographical order.

There exist extensive surveys on SA construction algorithms (SACAs), starting with the one by Puglisi et al. [42] and ending currently with the one by Bingmann [4, p. 163–192]. However, none of these surveys address any practical results for SACAs in main memory. There are 24 main memory SACAs that we are aware of. However, not all SACAs have been implemented. It is generally accepted that the *Divsufsort* [12,33] is the fastest SACA—despite it having a superlinear running time. Different models of computation have also been considered for this problem: external memory, e. g., [5,9,18,19,20,38], shared memory, e. g., [20,20,25], distributed memory, e. g., [1,6,13,14,20,32,36], and GPGPU, e. g., [10,41,46,47].

In this paper, we first present a practical comparison of SACAs that have a publicly available implementation. This comparison has been conducted using our new SACA benchmark framework called *SACABench*, which allows for (a) an easy comparison of SACAs including the output of the results (running time and memory peak) in form of raw data in JSON format, as PDF, or LaTeX file, (b) a simple way to include new SACAs, such that the features mentioned before can be used, and (c) fast development of new SACAs due to a variety of building blocks needed for SACAs (such as prefix sorting, renaming techniques, etc.). The framework is available from `https://github.com/sacabench/sacabench`. It is coded in C++17 and contains 13 SACA implementations, which are to our best knowledge *all* SACAs having a publicly available implementation. See Fig. 1 for a list (and also the historical development) of the SACAs that are included in the framework. We then use the building blocks of SACABench to implement a new GPU-based SACA, which is competitive with the fastest parallel (shared memory) SACA par_DivSufSort [25]. Here, our GPU SACA achieves a speedup between 0.93 and 1.69 compared to par_DivSufSort for inputs fitting into the GPU's memory.

2 SACABench: A Suffix Array Construction Benchmark

In Fig. 1 we give an overview of different SACAs in main memory. There are four general types of SACAs: *Prefix Doubling* algorithms sort the length-2^i prefixes of all suffixes by using the length-2^{i-1} prefixes as keys, and stopping when all considered prefixes are unique. If carefully implemented, this results in a running time of $\mathcal{O}(n \lg n)$. *Induced Copying* algorithms first sample certain suffixes and only sort those suffixes. Based on the sorted sample, the lexicographical order of all other suffixes can be computed in a second phase, which usually has linear running time. Depending on which algorithms are used to sort the sample, induced copying algorithms have either linear or slightly superlinear running time. *Recursive* algorithms reduce the problem size during each recursive step until the problem is trivially solvable (e.g., when all suffixes start with unique characters). They can achieve linear running time and are sometimes used in induced copying algorithms in the first phase (to achieve linear running time). *Grouping* is a new approach somehow similar to induced copying. Here, all suffixes are first grouped together by presorting them according to some prefix (in the only algorithm using grouping [3], Lyndon words determine this prefix). Those groups are then refined using already sorted suffixes, similar to induced copying algorithms.

2.1 Experimental Setup

We conducted our experiments on a computer with two Intel E5-2640v4 (10 physical cores, Hyper-Threading is disabled (per default on the cluster that can not be changed by users), with frequencies up to 3.4 GHz, and cache sizes of 320 KiB (L1I and L1D), 2.5 MiB (L2) and 25 MiB (L3)), one NVidia Tesla K40 graphics card (2880 stream processors with frequencies up to 875 MHz and 12 GB GDDR5 SDRAM) and 64 GB of RAM. We compiled the code using `g++` 8.3.0 and compiler flags `-O3` and `-march=native`. Note that *Cilk* support was removed from `g++` 8.0.0. Hence, we use *OpenMP* to express parallelism.

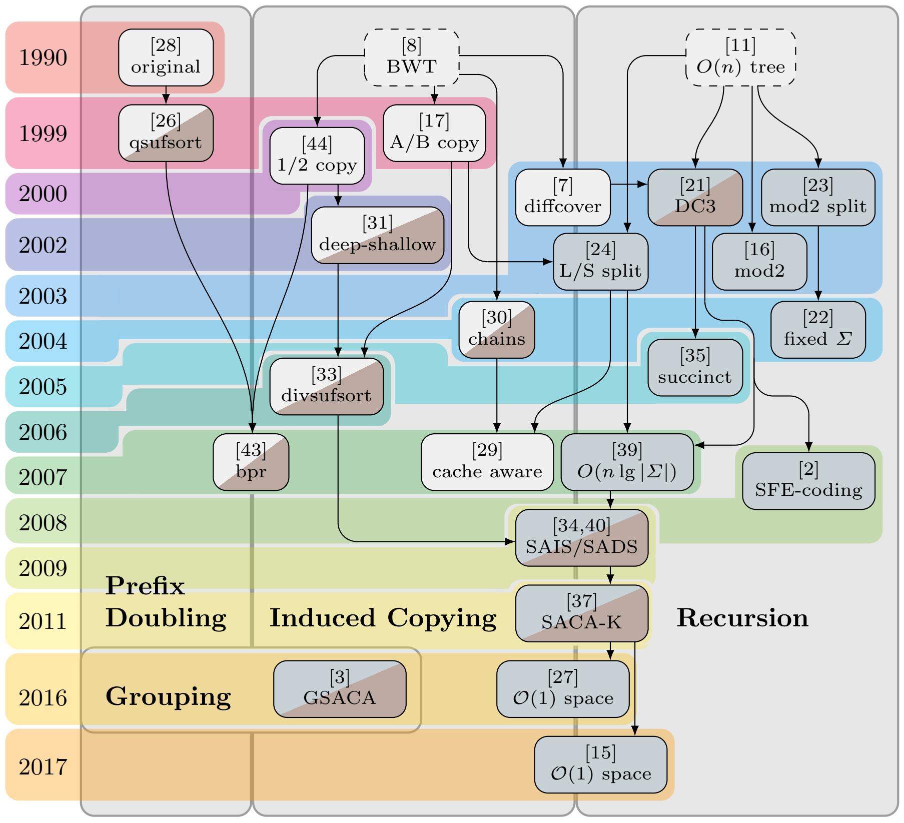

Fig. 1. Historical development of SACAs in main memory (enhanced and updated, based on [4,42]). For each algorithm, we cite its most recent publication, and the years on the left hand side show the year of its first publication. In some cases these years may not match, e. g., due to a later journal publication. SACAs are marked with a grey background (⬜), if they have linear running time, and a partly brown background (🟫), if an implementation is publicly available. All of the latter are also part of SACABench.

2.2 Evaluation of Sequential Suffix Array Construction Algorithms

For the evaluation of the sequential SACAs we use 1600 MiB prefixes of three texts. Note that we encode each symbol of the text using one byte, as this is required by most implementations. *1000G* ($\sigma = 4, \text{avg\_lcp} = 24, \text{max\_lcp} = 353$), which is a concatenation of DNA sequences provided by the 1000 Genomes Project (`https://internationalgenome.org`). We removed every character but `A`, `C`, `G`, and `T`. *CommonCrawl* ($\sigma = 242, \text{avg\_lcp} = 3,995, \text{max\_lcp} = 605,632$), which is a crawl of the web done by the CommonCrawl Corpus (`http://commoncrawl.org`) without any HTML tags. Here, we also removed all annotations added. Last, *Wiki* ($\sigma = 209, \text{avg\_lcp} = 32, \text{max\_lcp} = 25,063$), which is a concatenation of recent Wikipedia dumps in XML format (`https://dumps.wikimedia.org`).

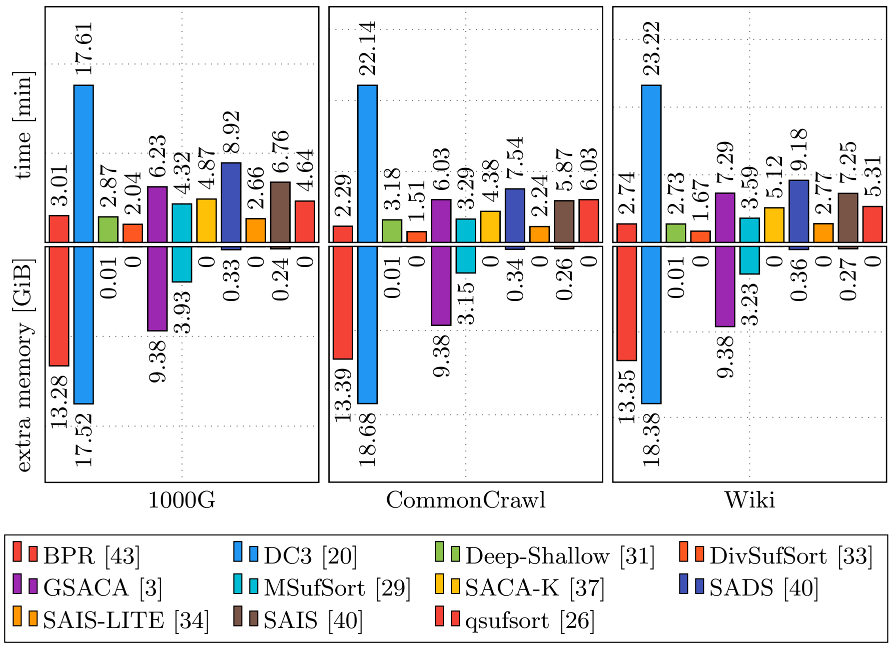

Fig. 2. Running times and extra memory usage (memory required in addition to the SA and input text) for all sequential SACAson real world inputs. The LaTeX code of the plot was generated using SACABench (legend and size slightly modified to fit in this layout).

Here, max_lcp denotes the maximum size of a common prefix of two suffixes that are consecutive in the SA, and avg_lcp is the average of if all these sizes (rounded down).

We used this set of texts, as more popular corpora (e. g., Pizza & Chili `http://pizzachili.dcc.uchile.cl` or the Lightweight corpus `http://people.unipmn.it/manzini/lightweight`) do only contain one files larger than 1600 MiB and we want to test on larger inputs.

In addition, we also tested the algorithms on highly repetitive texts that are available from the Pizza & Chili corpus, as some suffix array construction algorithms behave differently on this kind of input. To be precise, we use *Cere* ($\sigma = 5, n = 461,286,644, \text{avg\_lcp} = 7,066, \text{max\_lcp} = 303,204$), *Einstein.en.txt* ($\sigma = 139, n = 467,626,544, \text{avg\_lcp} = 59,074, \text{max\_lcp} = 935,920$), and *Para* ($\sigma = 5, n = 429,265,758, \text{avg\_lcp} = 3,273, \text{max\_lcp} = 104,177$).

Running time and memory usage are automatically measured by the framework for each included algorithm. To this end, we use the timing functionality of C++ and have overwritten the malloc, realloc, and free functions to track the memory usage of all components and also already coded algorithms.

The running times and the additional memory required are shown in Fig. 2. It is easy to see that DivSufSort is the fastest sequential SACA running in main

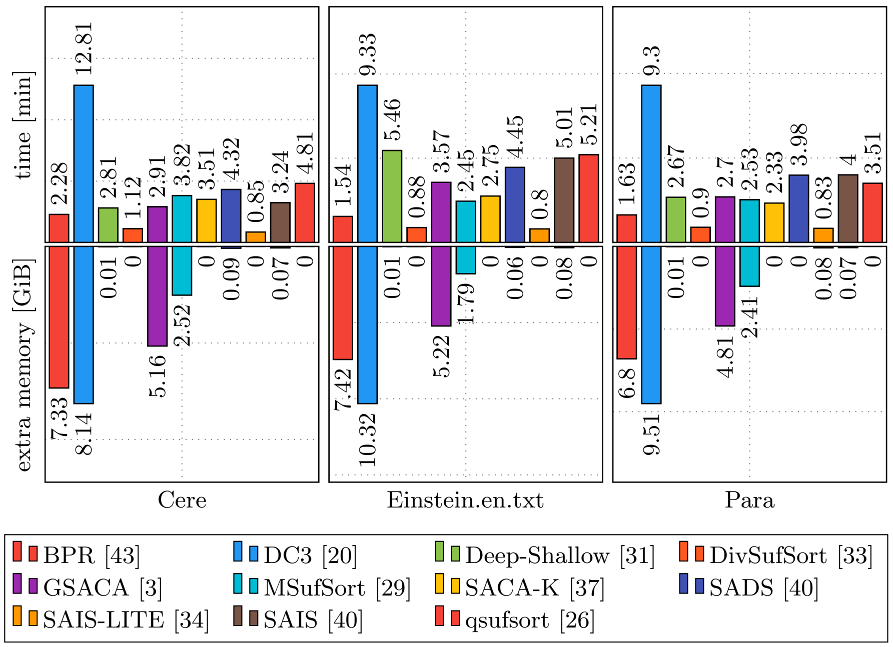

Fig. 3. Running times and extra memory usage (memory required in addition to the SA and input text) for all sequential SACAs on highly repetitive inputs. The LaTeX code of the plot was generated using SACABench (legend and size slightly modified to fit in this layout).

memory on all input texts. Also, it is among the SACAs that require nearly no memory in addition to the space for the SA and the input text. Overall, DivSufSort is 1.3, 1.61, and 1.63 times faster than the second fastest SACA on DNA, CommonCrawl, and Wiki. SAIS-LITE, which also does not require additional memory, is the second fastest SACA on DNA and CommonCrawl. It is noteworthy that both DivSufSort and SAIS-LITE have been coded by Yuta Mori. On Wiki, Deep Shallow is the second fastest SACA, but it is just 0.01 seconds faster than BPR and 0.04 seconds faster than SAIS-LITE. Those two SACAs (BPR and Deep Shallow) are also the third and fourth fastest algorithm on CommonCrawl and the fourth and third fastest on DNA. BPR is the only algorithm among the fast ones that requires an extensive amount of additional memory. More than 13 GiB for an input of size 1600 MiB. BPR, DC3, GSACA, and MSufSort require more additional memory than the size of the input.

For the highly repetitive texts, we have similar results regarding the running time and memory peaks. We show the results of our experiments in Fig. 3. Surprisingly, SAIS-LITE is faster than DivSufSort on this kind of inputs. On Cere it is 24.11 % faster, on Einstein.en.txt it is 9.1 % faster, and on Para it is 8.78 % faster. All this while requiring the same memory as DivSufSort.

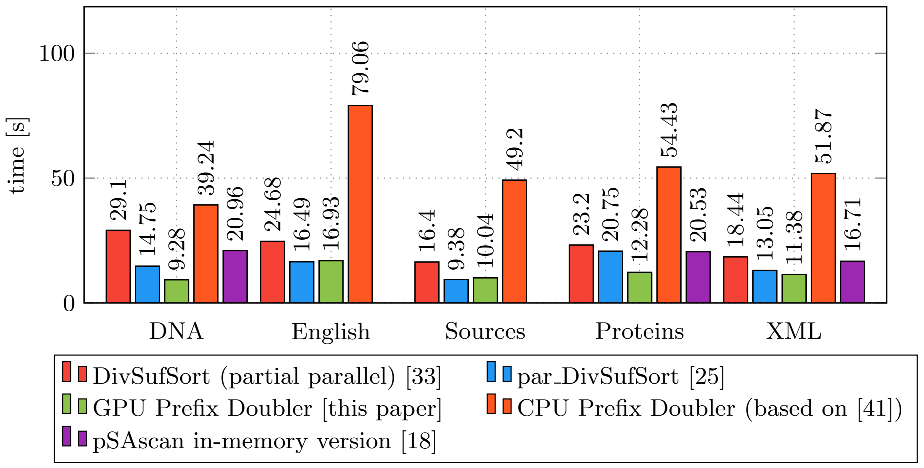

Fig. 4. Running time of the parallel (shared memory on 20 cores and GPU) SACAs on 200 MiB texts. The LaTeX code of the plot was generated using SACABench (legend and size slightly modified to fit in this layout).

3 Suffix Array Construction on the GPU

Next to the well tuned SACAs compared above, SACABench also contains many experimental SACA implementations. The best performing one is a parallel prefix doubling algorithm that runs on GPUs and is based on Osipov's GPU SACA [41, p. 44–51]. The main idea is similar to the general *prefix doubling* approach as used by Manber/Myers [28] and Larsson/Sadakane [26]. In iteration i, we consider the length-2^i prefixes of all suffixes and group equal (using the prefix as key) suffixes together into *buckets*. To refine the new buckets in the next iteration, we use the bucket numbers of the suffixes starting 2^{i-1} text positions to the right. This allows us to compute the new buckets without additional access to the text. A bucket is *sorted* if it contains only a single suffix. Larsson and Sadakane [26] added a clever mechanism to ignore already sorted buckets, which is a practical improvement. However, this can lead to load imbalance when parallelizing the algorithm. Our implementation combines techniques from both approaches such that sorted buckets can be ignored, but load imbalance is avoided by marking sorted groups and making heavy use of parallel prefix sums to compute the number of smaller groups for each group.

The prefix doubling technique has proven to be effective in other models of computation, e. g., distributed memory [6,13,14] and external memory [9].

3.1 Evaluation of Parallel Suffix Array Construction Algorithms

We compare our GPU SACA with three shared memory parallel SACAs. We could not compare our algorithm with the most recent GPU-algorithm by Wang et al. [47], we could only run it successfully for inputs smaller than 100 KiB for our text collection. (To test their code, they use somehow meaningless *random* input texts, which we could get to work in our test environment for sizes up to 170 MiB. However, even if we reduced the alphabet size of our real world texts to match the alphabet size of the random texts, we could not get this algorithm to work with inputs larger than 100 KiB.) Likewise, we could not compare against Osipov's CPU-SACA [41], as it does not have publicly available code and the author seems to be have left research and did not reply to our code requests. We are also aware of *parallelKS*, *parallelRange* that are available from the *Problem Based Benchmark Suite* [45], however we were not able to make them compute the correct suffix array on short notice for the final version of this paper.

As inputs we use the *Pizza & Chili* corpus, as it offers a variety of smaller text that have size at least 200 MiB: DNA ($\sigma = 16, \text{max\_lcp} = 14,836$), English ($\sigma = 225, \text{max\_lcp} = 109,394$), Sources ($\sigma = 230, \text{max\_lcp} = 71,651$), Proteins ($\sigma = 25, \text{max\_lcp} = 45,704$), and (dblp.)XML ($\sigma = 96, \text{max\_lcp} = 1,084$). More characteristics of the texts are available from `http://pizzachili.dcc.uchile.cl`. Again, max_lcp denotes the maximum size of a common prefix of two suffixes that are consecutive in the SA. We only use inputs of 200 MiB due to the memory requirements of our algorithm. On the given hardware it cannot compute the suffix array for larger inputs.

The results of our experiments are shown in Fig. 4, where *par_DivSufSort* denotes the *fully* parallel version of DivSufSort by Labeit et al. [25]. The *partially parallel* DivSufSort is Mori's [33] implementation of DivSufSort, where only the first phase is be parallelized. The *GPU Prefix Doubler* is the algorithm presented in this paper. The *CPU Prefix Doubler* is the same as the GPU one but it only uses the CPU, which we included as sanity check to see the speedup of the GPU. The running time of prefix doubling SACAs is $\mathcal{O}(n \lg \text{max\_lcp})$.

Our new algorithm is the fastest on DNA, Proteins, and XML, where max_lcp is comparatively small. Here, we are 1.58 (DNA), 1.69 (Proteins), and 1.15 (XML) times faster than par_DivSufSort. On English and Sources, par_DivSufSort is 1.03 (English) and 1.07 (Sources) times faster than our GPU Prefix Doubler on inputs with large max_lcp. Hence, it is only slightly faster.

We also included the in-memory version of the external memory suffix array construction algorithm *pSAscan* [18] in our framework. The available implementation could not handle all inputs by design, as it cannot handle text that contain the character 255, which occurs in English and Sources. It is 2.25 times, 1.67 times, and 1.46 times slower than our GPU Prefix Doubler.

Although a fair comparison against [41] is difficult due to the problems mentioned above, we hypothesize the following: Osipov [41] used an NVidia Fermi GTX 480 graphics cards with 480 and 1.5 GB RAM and an Intel i7 920 CPU with 4 cores and frequencies up to 2.93 GHz, where they achieved a speedup of at most 5.8 against partially sequential DivSufSort in the best case, but often a speedup of only around 2.5. Our speedup against the partial parallelized DivSufSort. varies between 1.45 (English) and 3.14 (DNA). Given that the ratio between the GPU and CPU cores is nearly the same in both setups (120:1 in their experiment and 144:1 in ours), but that our CPU cores have a higher frequency, we speculate that our implementation is of similar speed as Osipov's original one.

4 Conclusion

We presented a framework for SACAs that allows for an easy comparison of SACAs regarding time and memory consumption during construction. The result of this comparison is an empirical proof that *DivSufSort* is still the fastest SACA. It also has (in practice) optimal space requirements, as the additional memory only depends on the size of the alphabet. In addition, new algorithms can effortless be included in the framework allowing all features of the framework to be used. We also presented a GPU SACA that is the fastest parallel SACA, but is limited by the memory size of the graphics card, and part of the framework.

Recently, linear time SACAs that require only a constant number of computer words in addition to SA and the input text have been presented [15,27], which is optimal. Now, the only open question regarding SACAs in main memory is: is there a SACA faster than DivSufSort, which is the fastest since 2006? And if there is a faster algorithm than SAIS-LITE for highly repetitive texts, as it is even faster than DivSufSort on those.

Acknowledgment. We would like to thank the anonymous reviewer who pointed us to additional parallel suffix array construction algorithms that we had not previously included in the framework.

References

1. Abdelhadi, A., Kandil, A., Abouelhoda, M.: Cloud-based parallel suffix array construction based on MPI. In: Middle East Conference on Biomedical Engineering (MECBME). pp. 334–337. IEEE (2014)
2. Adjeroh, D.A., Nan, F.: Suffix sorting via Shannon-Fano-Elias codes. In: Data Compression Conference (DCC). p. 502. IEEE (2008)
3. Baier, U.: Linear-time suffix sorting - A new approach for suffix array construction. In: 27th Annual Symposium on Combinatorial Pattern Matching (CPM). LIPIcs, vol. 54, pp. 23:1–23:12. Schloss Dagstuhl Leibniz Center for Informatics (2016)
4. Bingmann, T.: Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools. Ph.D. thesis, Karlsruhe Institute of Technology, Germany (2018). https://doi.org/10.5445/IR/1000085031
5. Bingmann, T., Fischer, J., Osipov, V.: Inducing suffix and LCP arrays in external memory. ACM J. Exp. Algorithmics **21**(1), 2.3:1–2.3:27 (2016)
6. Bingmann, T., Gog, S., Kurpicz, F.: Scalable construction of text indexes with thrill. In: IEEE International Conference on Big Data. pp. 634–643. IEEE (2018)
7. Burkhardt, S., Kärkkäinen, J.: Fast lightweight suffix array construction and checking. In: 14th Annual Symposium on Combinatorial Pattern Matching (CPM). LNCS, vol. 2676, pp. 55–69. Springer (2003)
8. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. rep., Digital Equipment Corporation (1994)

9. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. ACM J. Exp. Algorithmics **12**, 3.4:1–3.4:24 (2008)
10. Deo, M., Keely, S.: Parallel suffix array and least common prefix for the GPU. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). pp. 197–206. ACM (2013)
11. Farach, M.: Optimal suffix tree construction with large alphabets. In: 38th IEEE Annual Symposium on Foundations of Computer Science (FOCS). pp. 137–143. IEEE (1997)
12. Fischer, J., Kurpicz, F.: Dismantling divsufsort. In: Prague Stringology Conference (PSC). pp. 62–76. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague (2017)
13. Fischer, J., Kurpicz, F.: Lightweight distributed suffix array construction. In: 21st Workshop on Algorithm Engineering and Experiments (ALENEX). pp. 27–38. SIAM (2019)
14. Flick, P., Aluru, S.: Parallel distributed memory construction of suffix and longest common prefix arrays. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC). pp. 16:1–16:10. ACM (2015)
15. Goto, K.: Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. CoRR **arXiv:1703.01009** (2017)
16. Hon, W., Sadakane, K., Sung, W.: Breaking a time-and-space barrier in constructing full-text indices. SIAM J. Comput. **38**(6), 2162–2178 (2009)
17. Itoh, H., Tanaka, H.: An efficient method for in memory construction of suffix arrays. In: 6th International Symposium on String Processing and Inforation Retrieval (SPIRE). pp. 81–88. IEEE (1999)
18. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Parallel external memory suffix sorting. In: 16th Annual Symposium on Combinatorial Pattern Matching (CPM). LNCS, vol. 9133, pp. 329–342. Springer (2015)
19. Kärkkäinen, J., Kempa, D., Puglisi, S.J., Zhukova, B.: Engineering external memory induced suffix sorting. In: 19th Workshop on Algorithm Engineering and Experiments (ALENEX). pp. 98–108. SIAM (2017)
20. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: 30th International Colloquium on Automata, Languages, and Programming (ICALP). LNCS, vol. 2719, pp. 943–955. Springer (2003)
21. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006)
22. Kim, D.K., Jo, J., Park, H.: A fast algorithm for constructing suffix arrays for fixed-size alphabets. In: 3rd International Workshop on Experimental and Efficient Algorithms (WEA). LNCS, vol. 3059, pp. 301–314. Springer (2004)
23. Kim, D.K., Sim, J.S., Park, H., Park, K.: Constructing suffix arrays in linear time. J. Discrete Algorithms **3**(2-4), 126–142 (2005)
24. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. J. Discrete Algorithms **3**(2-4), 143–156 (2005)
25. Labeit, J., Shun, J., Blelloch, G.E.: Parallel lightweight wavelet tree, suffix array and fm-index construction. J. Discrete Algorithms **43**, 2–17 (2017)
26. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Theor. Comput. Sci. **387**(3), 258–272 (2007)
27. Li, Z., Li, J., Huo, H.: Optimal in-place suffix sorting. In: Data Compression Conference (DCC). p. 422. IEEE (2018)
28. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)

29. Maniscalco, M.A., Puglisi, S.J.: An efficient, versatile approach to suffix sorting. ACM J. Exp. Algorithmics **12**, 1.2:1–1.2:23 (2007)
30. Manzini, G.: Two space saving tricks for linear time LCP array computation. In: 9th Scandinavian Workshop on Algorithm Theory (SWAT). LNCS, vol. 3111, pp. 372–383. Springer (2004)
31. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica **40**(1), 33–50 (2004)
32. Metwally, A.A., Kandil, A.H., Abouelhoda, M.: Distributed suffix array construction algorithms: Comparison of two algorithms. In: Cairo International Biomedical Engineering Conference (CIBEC). pp. 27–30. IEEE (2016)
33. Mori, Y.: divsufsort, `https://github.com/y-256/libdivsufsort` (2006)
34. Mori, Y.: sais, `https://sites.google.com/site/yuta256/sais` (2008)
35. Na, J.C.: Linear-time construction of compressed suffix arrays using o(n log n)-bit working space for large alphabets. In: 16th Annual Symposium on Combinatorial Pattern Matching (CPM). LNCS, vol. 3537, pp. 57–67. Springer (2005)
36. Navarro, G., Kitajima, J.P., Ribeiro-Neto, B.A., Ziviani, N.: Distributed generation of suffix arrays. In: 8th Annual Symposium on Combinatorial Pattern Matching (CPM). LNCS, vol. 1264, pp. 102–115. Springer (1997)
37. Nong, G.: Practical linear-time O(1)-workspace suffix sorting for constant alphabets. ACM Trans. Inf. Syst. **31**(3), 15 (2013)
38. Nong, G., Chan, W.H., Hu, S.Q., Wu, Y.: Induced sorting suffixes in external memory. ACM Trans. Inf. Syst. **33**(3), 12:1–12:15 (2015)
39. Nong, G., Zhang, S.: Optimal lightweight construction of suffix arrays for constant alphabets. In: 10th International Symposium on Algorithms and Data Structures (WADS). LNCS, vol. 4619, pp. 613–624. Springer (2007)
40. Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Trans. Comput **60**(10), 1471–1484 (2011)
41. Osipov, V.: Parallel suffix array construction for shared memory architectures. In: 19th International Symposium on String Processing and Inforation Retrieval (SPIRE). LNCS, vol. 7608, pp. 379–384. Springer (2012)
42. Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. **39**(2) (2007)
43. Schürmann, K., Stoye, J.: An incomplex algorithm for fast suffix array construction. Software: Practice and Experience **37**(3), 309–329 (2007)
44. Seward, J.: On the performance of BWT sorting algorithms. In: Data Compression Conference (DCC). pp. 173–182. IEEE (2000)
45. Shun, J., Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Kyrola, A., Simhadri, H.V., Tangwongsan, K.: Brief announcement: The problem based benchmark suite. In: 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). pp. 68–70. ACM (2012)
46. Sun, W., Ma, Z.: Parallel lexicographic names construction with CUDA. In: 15th IEEE International Conference on Parallel and Distributed Systems (ICPADS). pp. 913–918. IEEE (2009)
47. Wang, L., Baxter, S., Owens, J.D.: Fast parallel skew and prefix-doubling suffix array construction on the GPU. Concurrency and Computation: Practice and Experience **28**(12), 3466–3484 (2016)