On the energy efficiency of suffix array construction algorithms

Antonio Brogi[1], Luigi di Micco[1,*] and Giovanni Manzini[1]

[1]Department of Computer Science, University of Pisa, Pisa, Italy

Abstract

The suffix array is a fundamental data structure in string processing, with applications in text indexing, data compression, and bioinformatics. Its construction has been studied extensively, leading to many suffix array construction algorithms (SACAs) based on different algorithmic strategies and with different theoretical and practical properties. However, better theoretical guarantees do not always translate into better practical performance, as modern architectures make implementation details, memory locality, input characteristics, and parallelism crucial factors. This paper studies suffix array construction from an energy-efficiency perspective. Starting from the SacaBench framework, we extend the experimental methodology with support for energy measurements, enabling a systematic comparison of sequential and parallel SACAs across running time, peak memory usage, and energy consumption. The evaluation considers both heterogeneous real-world datasets and highly repetitive inputs, highlighting how dataset characteristics, memory behaviour, and thread-level parallelism influence the energy profile of different algorithms. The results show that energy efficiency cannot be inferred from running time alone. Although faster algorithms often consume less total energy, the relation between running time and energy consumption also depends on memory behaviour and execution configuration. We also report preliminary results on a parallel variant of the classical qsufsort algorithm, used as a case study to investigate the impact of parallelism on running time and energy consumption.

Keywords

suffix array construction, parallel algorithms, string algorithms, energy efficiency, experimental algorithmics

1. Introduction

Suffix arrays are among the most important data structures in string processing [1]. Given a text, the suffix array stores the lexicographic order of all suffixes of the text and provides the basis for efficient pattern matching, compressed indexes, and several applications in computational biology and data compression [2, 3].Many suffix array construction algorithms have been proposed, using different algorithmic strategies and therefore exhibiting different computational and memory-access patterns, ranging from theoretically linear-time algorithms to highly engineered practical implementations [4, 5]. However, modern hardware makes the practical performance of these algorithms depend on more than asymptotic complexity alone. Memory traffic, cache behaviour, and parallel scalability all contribute to the real cost of suffix array construction, ultimately affecting energy consumption. This paper reports preliminary results along two related directions. First, we extend the SacaBench framework [4] with energy measurements and use it to compare representative SACAs across running time, memory usage, and energy consumption. Second, we introduce a novel parallel variant of `qsufsort` [6], i.e., `PQSufsort`, and use it as a case study to investigate how thread-level parallelism affects both running time and energy consumption.

2. Background

Suffix arrays were introduced by Manber and Myers in 1993, along with algorithms for their construction and use, as a space-efficient alternative to suffix trees [1]. Since then, suffix arrays have become one of the most important data structures in string processing and compressed indexing. Let S be a string

ICTCS 2026: 27th Italian Conference on Theoretical Computer Science, September 7–9, 2026, Udine, Italy

*Corresponding author.

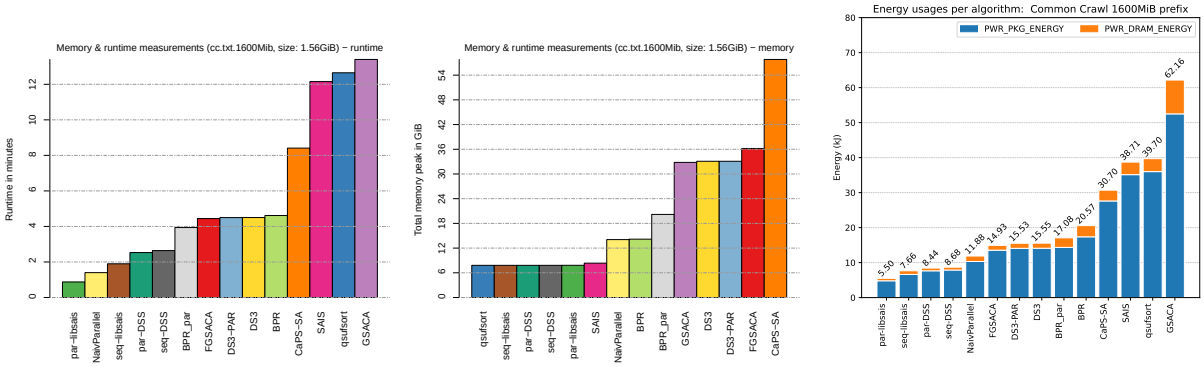
antonio.brogi@unipi.it (A. Brogi); luigi.dimicco.it@gmail.com (L. di Micco); giovanni.manzini@unipi.it (G. Manzini)

of length n, where each symbol is drawn from a finite alphabet Σ equipped with a total order. The suffix array of S is an array of integers $SA[0 \dots n-1]$ storing a permutation of $\{0, 1, \dots, n-1\}$ such that $S_{SA[i]} < S_{SA[i+1]}$ for all $0 \leq i < n-1$ where $S_{SA[i]}$ denotes the suffix of S starting at position $SA[i]$. In other words, the suffix array stores the starting indices of all suffixes of S in lexicographic order. The suffix array construction problem consists of computing the suffix array of an input string efficiently in terms of runtime and memory usage. The natural naive approach sorts all the suffixes of S using a general comparison-based sorter, resulting in $O(n^2 \log n)$ time. Smarter SACAs exploit the strong overlap between suffixes to achieve better time complexity. SACAs can be classified into several families [5, 4] based on their algorithmic strategies: prefix-doubling algorithms, induced-sorting algorithms, recursive algorithms, and grouping-based algorithms [4, 5]. These families differ not only in their theoretical complexity, but also in their practical behaviour on modern hardware. For example, algorithms with excellent asymptotic bounds may require complex memory access patterns, whereas asymptotically suboptimal algorithms can sometimes perform competitively because of better locality or lower overhead. It is not trivial to predict the practical performance of these algorithms and their energy consumption. Thus, starting from an existing benchmarking framework, we extended it to include energy measurements, enabling a systematic comparison of SACAs across multiple metrics. The energy consumption has been measured through hardware-supported energy counters, in particular using the Running Average Power Limit (RAPL) interface on Intel processors [7], which provides estimates of energy consumption for CPU and DRAM components. Motivated by the observation that the parallel versions reduce the runtime without increasing energy consumption, we further investigated parallel suffix array construction algorithms. Our focus has been the `qsufsort` algorithm, which adopts a prefix-doubling strategy and is known for its simplicity and practical robustness, especially on large alphabets, and shows a structure suitable for parallelization. Finally, we present a possible approach towards the parallelization of `qsufsort`, comparing it with a state-of-the-art parallel SACA, showing that it is possible to achieve competitive running times and energy efficiency by carefully designing the parallelization strategy.

3. Energy-aware experimental evaluation

3.1. Methodology

The experimental analysis builds on the SacaBench framework [4], which provides implementations of several suffix array construction algorithms, a collection of benchmark datasets, and a methodology for measuring running time and memory usage. Since SacaBench was released in 2019 and has not been actively updated since then, a first step consisted of extending the framework with more recent sequential and parallel SACAs. The newly included algorithms cover different recent directions in suffix array construction. The DS-family [8] algorithms and FGSACA [9] are inspired by Uwe Baier's GSACA [10] and exploit Lyndon-word-based grouping techniques. The DS-family algorithms replace parts of Baier's induced-copying procedure with faster integer-sorting techniques, sacrificing worst-case theoretical guarantees in favour of improved practical performance. In contrast, FGSACA exploits additional structural properties of Lyndon words in order to retain linear-time construction while improving the practical efficiency of the original approach. We also include CaPS-SA [11], a recent parallel and cache-friendly algorithm for constructing both the suffix array and the LCP (Longest Common Prefix) array. CaPS-SA is inspired by parallel sample sort and combines sample-based partitioning with LCP-aware merging. SacaBench was then extended with support for energy measurements, enabling a systematic comparison of SACAs across running time, peak memory usage, and energy consumption. Energy measurements are collected through hardware-supported energy counters using the Running Average Power Limit (RAPL) interface available on Intel processors. RAPL provides estimates for several power domains, including the processor package and DRAM, and has been widely adopted in experimental studies as a low-overhead mechanism for comparative energy analysis. Consequently, the reported measurements should be interpreted primarily as comparative indicators rather than absolute energy costs. Experiments were conducted on a dual-socket Intel Xeon Gold 6132 platform with 28



(a) Running time. (b) Peak memory usage. (c) Energy consumption.

Figure 1: Running time, peak memory usage, and energy consumption on the CC dataset.

physical cores (56 logical hardware threads), 377 GB of RAM distributed across two NUMA nodes, AVX2/AVX-512 support, and Ubuntu 22.04 using g++ 11.4.0. All parallel experiments were performed using up to 56 logical hardware threads, corresponding to the maximum degree of hardware parallelism available on the experimental platform. The evaluation uses the datasets included in SacaBench [4], which are designed to expose different algorithmic behaviours. They include both real-world datasets, such as natural-language texts and genomic sequences, and highly repetitive datasets. This distinction is important because suffix array construction algorithms can behave very differently depending on alphabet size, repetitiveness, and longest-common-prefix structure.

3.2. Running time, memory usage, and energy consumption

Figure 1 reports the results on the Common Crawl (CC) dataset, which is derived from large-scale web-crawl data and exhibits extensive repeated regions and LCPs among suffixes [4], and is used here as a representative example due to space constraints. The same analysis was conducted on the remaining datasets, and the discussion below summarizes the main trends observed across the full experimental campaign. Across the evaluated datasets, the parallel version of `libsais` [12] achieves the best overall behaviour when running time, peak memory usage, and total energy consumption are considered together. Its excellent running time is combined with one of the smallest memory footprints among all evaluated algorithms in both its sequential and parallel variants, substantially less than algorithms such as DS3, FGSACA, and CaPS-SA. This combination directly translates into the best energy results.

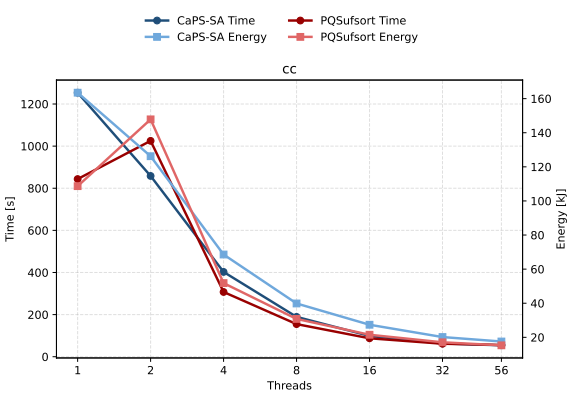
Overall, running time and total energy consumption are strongly correlated: algorithms that finish earlier generally consume less total energy. However, this correlation does not explain all observed behaviours. Peak memory usage and memory traffic also affect the final energy profile. For this reason, total energy consumption, defined as the sum of processor-package and DRAM energy, provides a more informative metric than either component alone, and it is reported in Figure 1c. The comparison between `libsais` and `qsufsort` illustrates that low memory usage alone is not sufficient to guarantee energy efficiency. Like `libsais`, `qsufsort` exhibits a compact memory footprint and remains remarkably memory efficient despite its age. Nevertheless, its running time is substantially worse than that of modern implementations. Conversely, algorithms with competitive running times may still be penalized by large working-memory requirements and memory-system pressure. DS3 and DS3-PAR achieve reasonably competitive running times, but they require substantially more memory than the induced-sorting implementations. A similar effect can be observed for CaPS-SA. Although it achieves competitive running times, it exhibits by far the largest memory footprint among the evaluated SACAs, reaching approximately 58 GiB across all datasets. This is mainly due to the auxiliary structures required by its sample-sort-based parallel merging strategy. Its high memory requirements are accompanied by relatively high total energy consumption: despite its parallel design and scalability-oriented structure, its runtime and energy efficiency remain worse than that of modern induced-sorting approaches such as

libsais and DivSufSort. The behaviour of DivSufSort [13] (denoted by seq-DSS and par-DSS in the figures) confirms the importance of considering running time and memory footprint jointly. Both the sequential and parallel versions remain remarkably efficient despite their age. Although they are slower than the newest implementations, they maintain one of the smallest memory footprints among all evaluated SACAs and often consume less energy than newer but more memory-intensive algorithms. Finally, NaivParallel, based on a general-purpose parallel sorter, performs well on the CC dataset and other real-world datasets, achieving good running times with a moderate memory footprint, though its energy consumption is slightly higher than expected from runtime alone. On highly repetitive datasets, however, it is not competitive, and its energy consumption closely tracks running time. The best results are obtained by implementations that combine short running times, compact memory usage, and favourable memory-access behaviour. In this sense, libsais represents the strongest overall point in the design space, while DivSufSort remains a robust and efficient baseline. Algorithms such as DS3, FGSACA, and CaPS-SA demonstrate that competitive running times may come at the cost of larger working memory and higher energy consumption. For the parallel algorithms, these effects are often amplified when scalability becomes constrained by the memory subsystem. Overall, four main trends emerge. First, the most optimized modern SACA (e.g., libsais) significantly outperforms older implementations in both running time and energy consumption. Second, running time is strongly correlated with total energy, but it does not fully determine energy efficiency. Third, memory footprint and memory-system behaviour play an important role, especially for memory-intensive and parallel algorithms. Finally, despite non-ideal scalability, the parallel implementations generally achieve lower total energy consumption than their sequential counterparts. In most cases, the reduction in running time outweighs the increased instantaneous use of computational resources, resulting in improved overall energy efficiency.

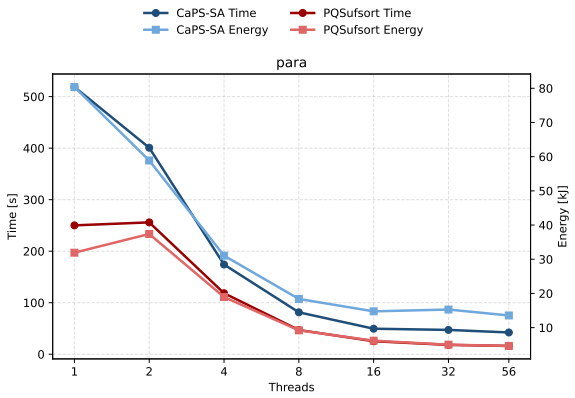
4. Parallelism and energy: parallel qsufsort

Another contribution of our research is the design and implementation of a parallel variant of qsufsort. qsufsort [6] is a classical suffix sorting algorithm based on iterative rank refinement. Although it is not a modern linear-time SACA, it remains interesting because of its simplicity, practical robustness, and natural support for large alphabets. The qsufsort algorithm by Larsson and Sadakane is based on a prefix-doubling strategy, which iteratively refines the order of suffixes by considering increasingly longer prefixes. This is achieved by maintaining a group array that assigns an integer identifier to each suffix based on the first k characters, where k doubles at each iteration. The very first step of the algorithm consists of sorting the suffixes according to their first character, which can be done efficiently using a bucket sort, then creating the initial groups based on the sorted order. After this, there is a refinement loop where the algorithm considers the next k characters of the suffixes, ending up refining the groups; this is done until all the groups are made of single suffixes, which means that the suffix array is fully sorted. Our parallelization of qsufsort introduced parallelism in both the initial sorting step and the iterative refinement loop, reducing the sequential portion of the algorithm. The first sorting step is parallelized using a parallel bucket sort, which divides the input into blocks and processes them in parallel to compute local histograms, which are then combined to produce the final sorted order. To parallelize the doubling refinement loop, we adopted a block-based decomposition strategy: the suffix array is partitioned into a fixed number of logical blocks that can be processed independently by workers. The algorithm parallelization has been designed to preserve group boundaries: no unresolved suffix group may be processed concurrently by multiple workers, since refinement updates group identifiers collectively for all suffixes belonging to the same group. The results show that the adopted parallelization strategy is effective, making it competitive with modern parallel SACAs. We compare PQSufsort with CaPS-SA [11], one of the most recent and best-performing parallel SACAs reported in the literature. Although libsais-par achieved the best overall results in our experimental evaluation, it is only briefly described in its GitHub repository [12] and, to the best of our knowledge, has not been studied in detail in the scientific literature. From a theoretical perspective, CaPS-SA is output-sensitive

and may exhibit $O(n^2)$ worst-case time complexity on adversarial inputs [11]. In contrast, PQSufsort preserves the $O(n \log n)$ worst-case complexity of qsufsort, while exploiting thread-level parallelism to reduce wall-clock execution time in practice. Nevertheless, CaPS-SA remains one of the strongest practical parallel SACAs currently available and serves as a relevant baseline for comparison. A formal analysis of the parallel complexity of PQSufsort, including a tighter characterization of achievable speedups as a function of p, namely the number of workers, is left as future work. As shown in Figure 2, the parallel qsufsort implementation, i.e., PQSufsort, achieves competitive running times and energy efficiency compared to state-of-the-art parallel SACAs such as CaPS-SA. On the CC dataset, PQSufsort outperforms CaPS-SA in running time for most evaluated thread counts, with the exception of two-worker configurations and high thread counts where CaPS-SA becomes competitive or slightly faster. A similar behaviour is observed on the other real-world datasets, namely DNA and Wiki. The highly repetitive datasets, chosen to model a class of inputs that frequently arise in practice, include Para (Figure 2b), which we use as a representative example. The Para dataset contains 36 sequences of *Saccharomyces paradoxus*. On these datasets, PQSufsort consistently achieves lower running times than CaPS-SA across all tested thread counts. It provides speedups of up to approximately $3\times$ over CaPS-SA, depending on the number of threads. The energy measurements follow the same general trend: in most configurations, the reduction in running time obtained through parallel execution also translates into lower total energy consumption. The plots show that runtime scalability and



(a) Comparison of CaPS-SA and PQSufsort time and energy varying the number of threads on the CC dataset.



(b) Comparison of CaPS-SA and PQSufsort time and energy varying the number of threads on the para dataset.

Figure 2: Comparison of the running time and energy consumption of CaPS-SA and PQSufsort as a function of the number of threads.

energy efficiency are closely related, but not equivalent. Increasing the number of threads generally reduces both running time and total energy consumption, although the improvement is not perfectly proportional because of synchronization overheads and memory-system effects.

5. Conclusion and future work

In this work, we studied suffix array construction algorithms from an energy-efficiency perspective. We extended SacaBench with energy measurements and compared several sequential and parallel SACAs. The results show that although running time is strongly correlated with total energy consumption, memory behaviour and the degree of parallelism also play a significant role in determining energy efficiency. As a complementary case study, we presented preliminary results on a parallel variant of qsufsort. The experiments indicate that PQSufsort can be competitive with modern parallel SACAs while also reducing total energy consumption. Future work includes a deeper microarchitectural analysis, a more detailed investigation of memory-locality effects on energy consumption, further optimization of PQSufsort, and a formal analysis of its parallel complexity.

Funding. Work partially supported by the INdAM-GNCS 2026 Project CUP_E53C25002010001, and by the project "Hub multidisciplinare e interregionale di ricerca e sperimentazione clinica per il contrasto alle pandemie e all'antibioticoresistenza (PAN-HUB)" funded by the Italian Ministry of Health (Project ID: T4-AN-07, CUP: I53C22001300001).

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT in order to: Grammar and spelling check, Improve writing style. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of this publication.

References

[1] U. Manber, G. Myers, Suffix Arrays: A New Method for On-Line String Searches, SIAM Journal on Computing 22 (1993) 935–948. doi:10.1137/0222058.

[2] P. Ferragina, G. Manzini, Indexing compressed text, Journal of the ACM 52 (2005) 552–581.

[3] R. Grossi, G. F. Italiano, et al., Suffix trees and their applications in string algorithms, in: Proceedings of the 1st South American Workshop on String Processing, 1993, pp. 57–76.

[4] J. Bahne, N. Bertram, M. Böcker, J. Bode, J. Fischer, H. Foot, F. Grieskamp, F. Kurpicz, M. Löbel, O. Magiera, R. Pink, D. Piper, C. Poeplau, SACABench: Benchmarking Suffix Array Construction, in: N. R. Brisaboa, S. J. Puglisi (Eds.), String Processing and Information Retrieval, Springer International Publishing, Cham, 2019, pp. 407–416. doi:10.1007/978-3-030-32686-9_29.

[5] S. J. Puglisi, W. F. Smyth, A. H. Turpin, A taxonomy of suffix array construction algorithms, ACM Computing Surveys 39 (2007) 4. doi:10.1145/1242471.1242472.

[6] N. J. Larsson, K. Sadakane, Faster suffix sorting, Theoretical Computer Science 387 (2007) 258–272. doi:10.1016/j.tcs.2007.07.017.

[7] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, Z. Ou, RAPL in Action: Experiences in Using RAPL for Power Measurements, ACM Transactions on Modeling and Performance Evaluation of Computing Systems 3 (2018) 1–26. doi:10.1145/3177754.

[8] N. Bertram, J. Ellert, J. Fischer, Lyndon Words Accelerate Suffix Sorting, in: ESA, volume 204, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, 2021, pp. 15:1–15:13. doi:10.4230/LIPIcs.ESA.2021.15.

[9] J. Olbrich, E. Ohlebusch, T. Büchler, Generic Non-recursive Suffix Array Construction, ACM Trans. Algorithms 20 (2024) 18:1–18:42. doi:10.1145/3641854.

[10] U. Baier, Linear-time Suffix Sorting – A New Approach for Suffix Array Construction, Proc. CPM (2016) 1–12. doi:10.4230/LIPIcs.CPM.2016.23.

[11] J. Khan, T. Rubel, E. Molloy, L. Dhulipala, R. Patro, Fast, parallel, and cache-friendly suffix array construction, Algorithms for Molecular Biology 19 (2024) 16. doi:10.1186/s13015-024-00263-5.

[12] I. Grebnov, libsais, https://github.com/IlyaGrebnov/libsais, 2026. GitHub repository, accessed 2026-05-17.

[13] J. Fischer, F. Kurpicz, Dismantling DivSufSort, 2017. doi:10.48550/ARXIV.1710.01896.