Parallel String Matching

Philip Pfaffe, Martin Tillmann, Sarah Lutteropp, Bernhard Scheirle, and Kevin Zerr

Karlsruhe Institute of Technology
{philip.pfaffe, martin.tillmann}@kit.edu
{sarah.lutteropp, bernhard.scheirle, kevin.zerr}@student.kit.edu

Abstract. We explore the benefits of parallelizing 7 state-of-the-art string matching algorithms. Using SIMD and multi-threading techniques we achieve a significant performance improvement of up to 43.3x over reference implementations and a speedup of up to 16.7x over the string matching program `grep`.
We evaluate our implementations on the smart-corpora and the full human genome data set. We show scalability over number of threads and impact of pattern length.

1 Introduction

String matching is a fundamental tool in a wide range of practical software. Molecular biology, data compression and information retrieval all rely on efficient string matching algorithms on challenging amounts of input data. For over 35 years string matching algorithms have been studied extensively. Speed and memory constraints are the crucial attributes of state-of-the-art matching algorithms.

Parallelization has become an essential part of algorithm design. Multi-threading, heterogeneous computing and SIMD (single instruction stream, multiple data stream) instructions are the current tools of the trade. Due to the data-parallel nature of most string matching algorithms, these techniques can be used to achieve significant performance gains.

In this paper, we propose parallelization improvements to existing state-of-the-art string matching algorithms. We explore a chunking approach, partitioning the input data and distributing the workload with a thread pool. We utilize modern SIMD-instructions to improve throughput in computational intensive situations and optimize data structures for parallel access. Our implementations are evaluated on the smart-corpora [11] and the human genome[1] [20]. To demonstrate the effectiveness of our approach, we compare the runtime of our implementations with sequential reference implementations provided by the

[1] Dec. 2013 (GRCh38/hg38) assembly of the human genome (hg38, GRCh38 Genome Reference Consortium Human Reference 38 (GCA_000001405.2)).
See http://genome.ucsc.edu/ for details on the data set.

smart-corpora as well as the string matching program `grep`[2] of the GNU/Linux operating system.

Pattern length and alphabet size influence the effectiveness of different algorithms, choosing the optimal implementation therefore depends on those two parameters. Our evaluation considers different combinations of pattern length and alphabet size. When both parameters are known at runtime this information can be used to choose the optimal algorithm.

2 Problem Definition

We define the problem of string matching as the task of finding a pattern P of length $m = |P|$ in a text T of length $n = |T|$. Pattern and text are based on an alphabet Σ. The results are the absolute positions of every occurrence of P in T. The input is dynamic, preprocessing of pattern or text have to take place at runtime. Only exact matches are returned, approximate matches or regular expression patterns are not considered.

3 Related Work

The introduction of the Knuth-Morris-Pratt [17] and Boyer-Moore [4] algorithms which are, respectively, the first linear and the first sublinear string matching algorithms, initiated the ongoing search for ever faster matching approaches. Both of these inspired many variations. Prominent examples are Horspool [15] and QuickSearch [25], simplifying variations of Boyer-Moore, which have proven to be efficient in practice. The Rabin-Karp [16] algorithm is an alternative solution to the string matching problem, testing for matches based on hashes computed from the input text and pattern.

In more recent years, many more variations and combinations of the classical matching algorithms have been proposed. Faro and Lecroq [11] report on more than 50 new algorithms that have been published since 2000. One example is the Average Optimal Shift-Or algorithm by Fredriksson and Grabowski [12], an extension of the original Shift-Or [2], which leverages bit-parallelism within pattern and text comparison. The BNDM algorithm by Navarro and Raffinot [23] is based on the same principle, and combines it with suffix automata to find matches by efficiently identifying all subpatterns of a word. Another family of algorithms which relies on finding subpatterns is BOM [1] and its variations(cf. e.g. [10]).

For a more detailed and more complete overview of recent advances in string matching algorithms we direct the interested reader to Faro's and Lecroq's review article [11].

Despite the global trend in industry and research to increase performance by parallelizing algorithms, to the best of our knowledge, only few parallel approaches to string matching exist, even though efficient theoretical solutions have

[2] GNU grep 2.20, Copyright (C) 2014 Free Software Foundation, Inc. http://www.gnu.org/software/grep/

been proposed: The optimal parallel algorithm for a CREW-PRAM (concurrent-read, exclusive write parallel random access machine) runs in $O(\log^2 n)$ [13]. For a CRCW-PRAM, even a constant time solution has been proposed [14]. There are, however, no practical implementations available for these theoretical algorithms. Nevertheless, there are several published approaches that in some sense rely on inherently parallel properties of string comparisons, such as by exploiting bit-parallelism [5] in comparing strings (cf. e.g. the Shift-Or algorithm [2] and its derivatives, or the works of Cantone et al. [6] or Peltola and Tarhio [24], among many others). Faro and Klekci, on the other hand, further increase the benefits of these approaches by using modern processor's SIMD extensions ([9], [19]).

Although there is a surprising lack of approaches leveraging classical threading parallelism, there are some works which explore the benefits provided by the massive parallel computing power within modern GPUs. Kouzinopoulos and Margaritis evaluate the performance of GPU implementations of the classical matching algorithms [18] and report on a possible speedup of more than 10x. Vasiliadis et al. [7] and Cascarano et al. [26] present solutions for regular expression matching in GPUs, which is a superset of the string maching problem. These approaches create finite state machines from the input patterns and execute them in parallel on partitioned input data. Another problem related to string matching is the approximate string matching problem, which allows for missing some possible matches in exchange for speed. Liu et al. [22] present GPU-based solutions and report on up to 80x speedups.

4 Implementation

We implement a general chunking approach for all of our string matching implementations. The initial text T is split into chunks of size $s = \max(2{*}m, s_a)$ where s_a is 4 MiB for the SSEF algorithm and 1 MiB for all other algorithms. A thread pool runs string matching tasks on these chunks in parallel. The string matching tasks examine an additional overlap of $m - 1$ characters after each chunk to ensure matches that cross chunk boundaries are found. This also avoids inter-chunk synchronization in the matching algorithm. If the text size is not large enough to create at least one chunk per thread, we reduce the chunk size to $s = n/\mathit{thread\_count}$. To preserve global ordering the matching results are written to a synchronized set.

We employ SSE (streaming SIMD extensions) in the appropriate implementations. We use the SSE instruction set (up to version 4.1), as it is supported by Intel and AMD CPUs. The resulting bit-parallelism is essential for high throughput on modern CPU cores.

Our implementations can be found on our project page[3]. We provide a unified C++ interface for all discussed algorithms.

The following subsections give a brief overview of the implemented algorithms. Of particular interest are our modifications to the SSEF algorithm. For a more detailed discussion we refer to the referenced articles.

[3] https://code.ipd.kit.edu/pmp/pgrep

4.1 Knuth-Morris-Pratt

The well-known Knuth-Morris-Pratt (KMP) algorithm was first published in 1977 [17]. It uses a preprocessing phase on the pattern to build a partial match table. This table can be used to skip known matching prefixes after a partial match was found. Once matched characters are therefore never visited again. The preprocessing phase runs in $O(m)$ and the actual matching in $O(n)$, resulting in an asymptotic runtime of $O(n + m)$.

4.2 Shift-Or

The Shift-Or algorithm proposed by Baeza-Yates and Gonnet in 1992 uses efficient bitwise operations [2]. For each character c in the alphabet Σ an occurrence bit-vector o_c is calculated in a preprocessing phase.

$$o_c[i] = \begin{cases} 1 & \text{, if } P[i] = c \\ 0 & \text{, otherwise} \end{cases}$$

In the matching phase a result bit-vector r is iteratively and-combined with the occurrence vector of the current character. Vector r is then bit-shifted by one position and incremented by one. A match is found when $r[m] = 1$. We use a word size of 64 bit for the bit-vectors. The runtime is deterministic and in $O(n * m)$.

4.3 Hash3

Lecroq's Hash$q$ algorithm from 2007 [21] is based on hash values for q-grams. The preprocessing phase computes a shift table for each hashed q-gram in the pattern. The search algorithm then hashes sub-strings of length q and skips characters according to the precomputed shift table. Potential matches are checked naively. Choosing $q = 3$ promises the best results for medium length patterns. Hash3 requires a minimum pattern length of $m = 3$.

4.4 SSEF

The SSEF algorithm [19] precomputes 65536 filter lists based on the kth bit of each character on the pattern. These filters are then applied efficiently, utilizing SSE instructions, on shifting alignments of pattern and text. SSEF is restricted to patterns with a minimum length of $m \geq 32$. The worst case runtime is in $O(n*m)$. If we consider the probability to filter possible matches, SSEF achieves an average runtime in $O(n * m/65536)$.

In the original SSEF algorithm parameter k has to be specified by the user. The smart-corpora implementation chooses a fixed value of $k = 7$. We improved on this by finding the bit that carries the most information in the pattern. We count the set bit positions in each character of the pattern and choose the bit

Character	Bits						
	7	6	5	4	3	2	1
a	1	1	0	0	0	0	1
c	1	1	0	0	0	1	1
a	1	1	0	0	0	0	1
f	1	1	0	0	1	1	0
Ratio	1.00	1.00	0.00	0.00	0.25	0.50	0.75

Table 1: Finding the bit that carries the most information. For the pattern 'acaf' the second bit is set in 50% of the characters.

that carries the most information, see table 1 for an example. Optimally the kth bit is set 50% of the time.

A second optimization is the filter list itself. The original algorithm and the smart-corpora implementation use a linked list and allocate each entry dynamically. The reference performs separate heap allocations for each individual entry. As the number of entries in this linked list is fixed for a given pattern size, we only allocate a single chunk of memory. This allows us to use simple offsets (instead of pointers) to address the list entries. Also we minimize the total memory footprint of the filter list by automatically using the smallest data type possible to store the offsets inside the list. This has the fortunate side effect of improved cache locality.

4.5 Variants of the Backward-Oracle-Matching

Faro and Lecroq presented Extended-Backward-Oracle-Matching (EBOM) and Forward-Simplified-Backward-Nondeterministic-DAWG-Matching (FSBNDM) in 2009 [10]. Both are variants of the Backward-Oracle-Matching algorithm and based on finite automata.

Extended-Backward-Oracle-Matching
The EBOM algorithm extends Backward-Oracle-Matching with a fast-loop. The fast-loop technique iterates a matching heuristic in a non-branching cycle. This is used to quickly locate the last character of the pattern in the currently observed text window. In each iteration two consecutive characters are handled. EBOM requires a preprocessing phase in $O(|\Sigma|^2)$.

Forward-Simplified-Backward-Nondeterministic-DAWG-Matching
The FSBNDM algorithm uses bit-parallelism to implement a non-deterministic forward automaton on the reversed pattern. The preprocessing phase can be performed in $O(|\Sigma| + m)$.

4.6 Exact-Packed-String-Matching

Exact-Packed-String-Matching (EPSM) was presented in 2013 by Faro and Külekci [8]. EPSM makes use of bit-parallelism by packing several characters into

a bit-word and partitioning text T into chunks T_i. These bit-word sized chunks are compared with a packed pattern bit-word. Shift and bitwise-and operations are used to efficiently compare text chunks with the pattern. Our implementation uses SSE registers as 128 bit words. We limit the usage of EPSM to cases with short patterns ($m \leq 8$). Under these restrictions EPSM is very fast and runs in $O(n)$. The asymptotic runtime for the general case remains $O(n * m)$.

5 Evaluation

In the following section we present the evaluation of the performance of our parallelized string matching algorithms. We show experimental results obtained from two benchmarks using the smart-corpora [11] and the human genome [20]. The human genome benchmark input text is the assembly of the human genome, which is 3.1 GB in size and uses an alphabet of four characters. The smart-corpora benchmark is comprised of seven input texts from the smart-coprora archive:

- The text of the English King James Bible, containing natural English language with a complete alphabet of 63 characters.
- A set of genome sequences for the E. Coli bacterium. The DNA is encoded over an alphabet of size 4.
- Four protein sequences `hi`, `hs`, `mj`, `sc`, with an alphabet of 20 characters (19 characters for the `hs` protein).
- The CIA world fact book. Natural English language with a few special characters. Alphabet size of 94.

For both benchmarks, we generate 10 patterns for every input file of the lengths 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024. The patterns for an input file are generated by randomly picking sequences of the respective length from the file, thus ensuring that there actually are matches for every file and pattern. The benchmark results shown in the remainder of this chapter are averaged over all 10 patterns for every file and configuration. To assess the benefits of parallelization, all experiments are conducted using 1,2,4, and 8 threads.

Additionally, we compare the performance results of our implementations with sequential reference implementations of the respective algorithm provided by the smart-corpora as well as the string matching program `grep` of the GNU/Linux operating system.

Input files are directly mapped into the application's memory to reduce I/O latencies. We ensured that input files are completely cached by the operating system. To get comparable results we used an equivalent memory-mapping interface for the smart-corpora algorithms. Memory-mapping is used in `grep` as well. We invoke `grep` with the parameters `grep <pattern> <file> -c`. To benchmark the actual string matching we use the additional switch `-c` to suppresses output of the individual matches and instead print the count of matching lines. The runtimes of `grep` and our implementations thus encompass the matching algorithm including all synchronization but minimize file and screen I/O. To run

the benchmarks we used `temci` [3], a benchmarking helper tool, in combination with `perf`, a tool for profiling with performance counters. All experiments were performed on an Intel Xeon E5 system, with 4 CPU cores (8 hardware threads) at 3.7 GHz.

In the following subsection we discuss an excerpt of our result data.

5.1 Results

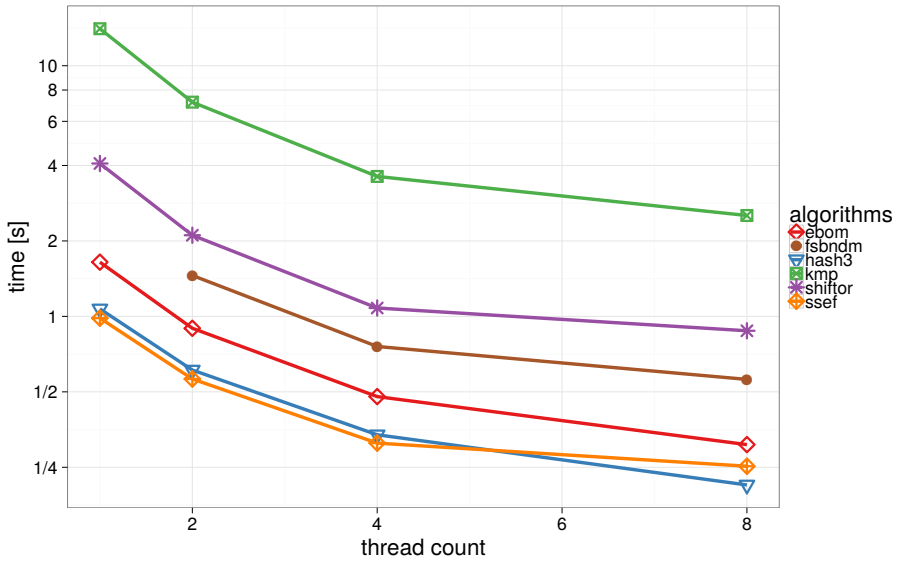


Fig. 1: Runtime scalability of six algorithms for 1,2,4, and 8 threads. Human genome data set, pattern length of 32.

Figure 1 shows the average time to match a pattern of length 32 on the genome data set for six algorithms on a logarithmic scale. We can observe linear scalability with increased thread count. Our FSBNDM implementation requires a minimum of two threads due to space limitations exceeded by the genome data set and the EPSM algorithm is not applicable due to $m > 8$. The content of the patterns has an insignificant impact on performance. The maximum relative standard deviation over the patterns is 3% with a relative range of 9%.

Figure 2 shows the average absolute runtimes of six algorithms on the smart-corpora. The algorithms use up to 8 threads. The pattern length is 32. Both SSEF and Hash3 are consistently fast on all seven texts. The relative performance between the algorithms is surprisingly stable.

In Figure 3 we see the average performance of seven algorithms over different pattern lengths. We use the bible text and our implementations use up to 8

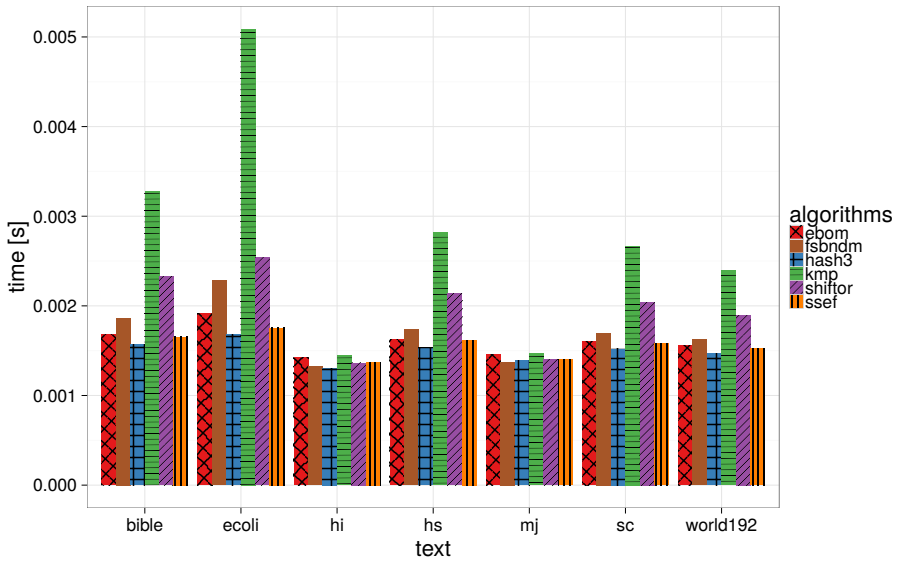


Fig. 2: Absolute runtimes of six parallelized algorithms for the seven texts of the smart-corpora. Pattern length is 32.

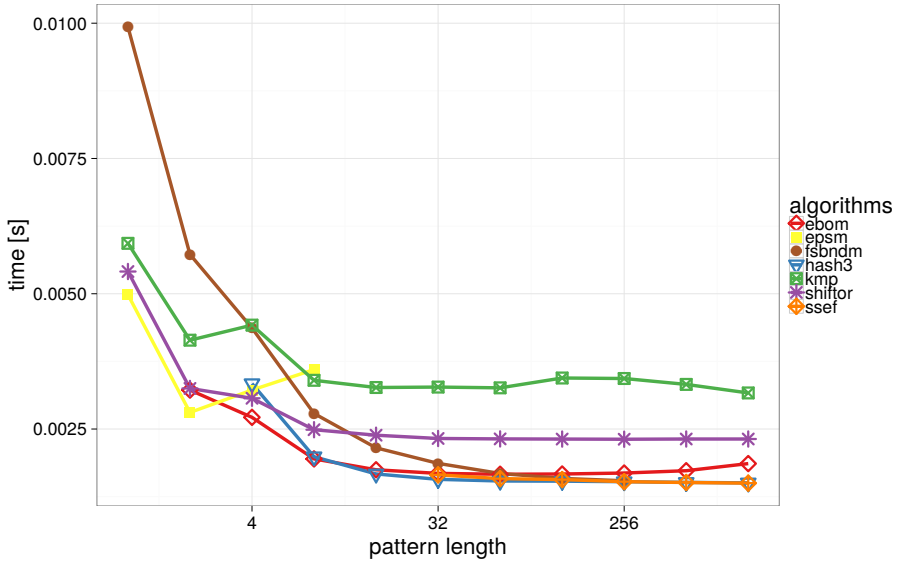


Fig. 3: Performance over pattern length on natural language text (bible).

threads. Several algorithms are restricted to specific pattern sizes. With increasing pattern length algorithm performance increases as well with the exception of EPSM which is optimal for $m = 2$. The maximum relative standard deviation over the pattern contents is 26%. This increase compared to the genome data set is explained by the relative small runtime influenced by measuring fluctuations.

To assess the practicality of our implementations we compare our runtimes against the performance of `grep`. In Figure 4 we show the relative speedups over different pattern lengths on the human genome data set on a logarithmic scale. In the case where we are limited to one thread, we can achieve a performance increase for pattern lengths between 4 and 128. However `grep` outperforms our implementations for patterns with $m \leq 2$ or $m \geq 256$. If we utilize eight threads we can achieve significant speedups of up to 16.7x for all patterns with $m \geq 2$. SSEF, EBOM and Hash3 all perform consistently well on this data set.

Figure 5 shows the speedups of our implementations over the reference implementations found in the smart-corpora. The speedups are displayed on a logarithmic scale. The baseline for each algorithm is the corresponding reference implementation. In contrast to speedups on the human genome data set, only the EPSM, KMP and Shift-Or implementations benefit from an increased thread count on this smaller data set. However our modifications to the SSEF implementation result in a significant speedup even in the sequential case.

6 Conclusion

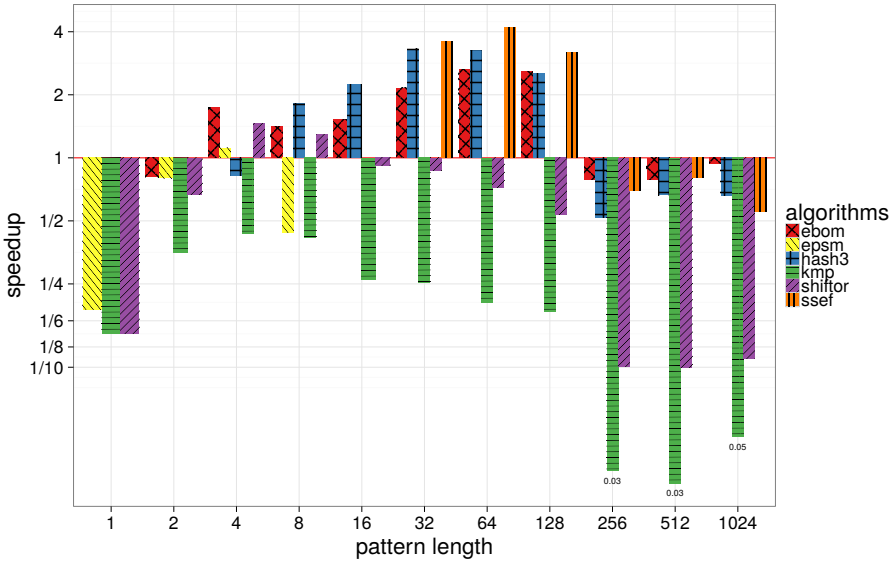
We used a chunking approach to parallelize seven state-of-the-art string matching algorithms. We have shown linear scalability on the number of threads for large input data. We observed the influence of pattern size on string matching algorithms. For short patterns EPSM and EBOM are the algorithms of choice, while bigger patterns favor Hash3, SSEF and FSBNDM.

SSEF is consistently fast over different alphabet sizes and the supported pattern lengths. With our modifications to SSEF we achieved a 43x speedup over the reference implementation. Compared with `grep` we achieve significant speedups in all cases where the pattern has two or more characters. On the human genome data set the maximal speedup of SSEF compared to `grep` is 15x.

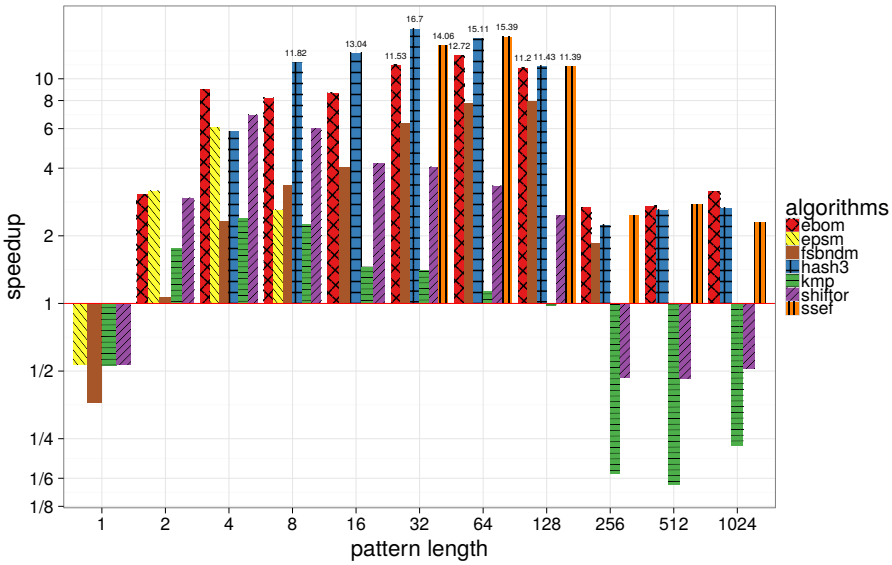
In the future we plan to explore a heterogeneous approach by distributing text chunks on CPUs, GPUs and Intel MICs.

References

1. Allauzen, C., Crochemore, M., Raffinot, M.: Factor oracle: A new structure for pattern matching. In: Theory and Practice of Informatics. Springer (1999)
2. Baeza-Yates, R., Gonnet, G.H.: A New Approach to Text Searching. Communications of the ACM 35(10) (1992)
3. Bechberger, J.: temci (2016), http://temci.readthedocs.io
4. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM 20(10) (1977)

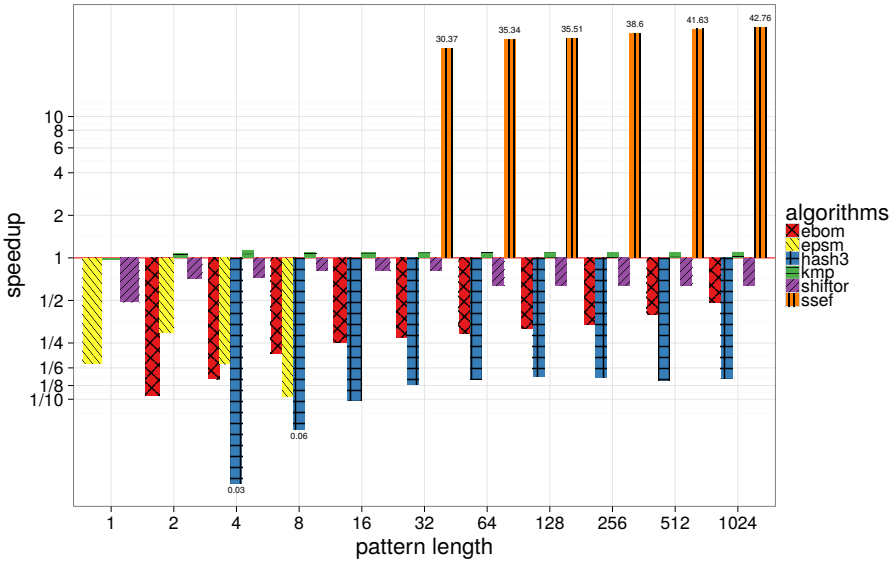


(a) Sequential execution on 1 thread.

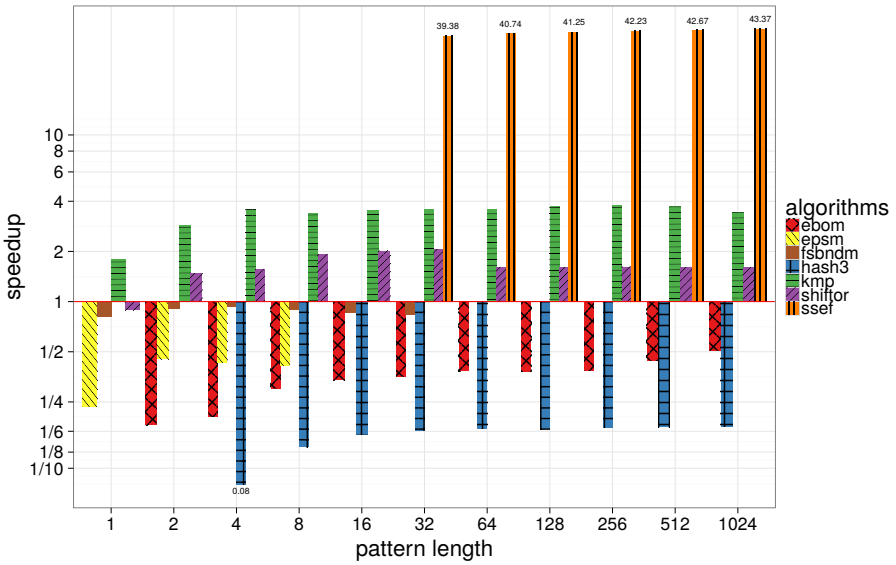


(b) Parallel execution on 8 threads.

Fig. 4: Speedup of our implementations over `grep` on different pattern lengths on the human genome data set.



(a) Sequential execution on 1 thread.



(b) Parallel execution on 8 threads.

Fig. 5: Speedup of our implementations over the smart-corpora reference implementations on different pattern lengths on a natural language text (bible).

5. Cantone, D., Faro, S., Giaquinta, E.: Bit-(parallelism) 2: Getting to the next level of parallelism. In: Fun with Algorithms. Springer (2010)
6. Cantone, D., Faro, S., Giaquinta, E.: A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. In: Combinatorial Pattern Matching. Springer (2010)
7. Cascarano, N., Rolando, P., Risso, F., Sisto, R.: iNFAnt: NFA Pattern Matching on GPGPU Devices. SIGCOMM Computer Communication Review 40(5) (2010)
8. Faro, S., Külekci, M.O.: Fast packed string matching for short patterns. In: Proceedings of the Meeting on Algorithm Engineering & Expermiments. Society for Industrial and Applied Mathematics (2013)
9. Faro, S., Külekci, M.O.: Fast and flexible packed string matching. Journal of Discrete Algorithms 28 (2014)
10. Faro, S., Lecroq, T.: Efficient variants of the backward-oracle-matching algorithm. International Journal of Foundations of Computer Science 20(06) (2009)
11. Faro, S., Lecroq, T.: The exact online string matching problem: A review of the most recent results. ACM Computing Surveys 45(2) (2013)
12. Fredriksson, K., Grabowski, S.: Practical and optimal string matching. In: String Processing and Information Retrieval. Springer (2005)
13. Galil, Z.: Optimal parallel algorithms for string matching. Information and Control 67(1) (1985)
14. Galil, Z.: A Constant-time Optimal Parallel String-matching Algorithm. Journal of the ACM 42(4) (1995)
15. Horspool, R.N.: Practical fast searching in strings. Software: Practice and Experience 10(6) (1980)
16. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2) (1987)
17. Knuth, D.E., Morris, Jr, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM journal on computing 6(2) (1977)
18. Kouzinopoulos, C.S., Margaritis, K.G.: String Matching on a Multicore GPU Using CUDA. In: 13th Panhellenic Conference on Informatics, 2009. PCI '09 (2009)
19. Külekci, M.O.: Filter Based Fast Matching of Long Patterns by Using SIMD Instructions. In: Stringology (2009)
20. Lander, E.S., Linton, L.M., Birren, B., Nusbaum, C., Zody, M.C., Baldwin, J., Devon, K., Dewar, K., Doyle, M., FitzHugh, W., others: Initial sequencing and analysis of the human genome. Nature 409(6822) (2001)
21. Lecroq, T.: Fast exact string matching algorithms. Information Processing Letters 102(6) (2007)
22. Liu, Y., Guo, L., Li, J., Ren, M., Li, K.: Parallel Algorithms for Approximate String Matching with k Mismatches on CUDA. In: Parallel and Distributed Processing Symposium Workshops PhD Forum (2012)
23. Navarro, G., Raffinot, M.: Fast and Flexible String Matching by Combining Bit-parallelism and Suffix Automata. ACM Journal of Experimental Algorithmics 5 (2000)
24. Peltola, H., Tarhio, J.: Alternative algorithms for bit-parallel string matching. In: String Processing and Information Retrieval. Springer (2003)
25. Sunday, D.M.: A very fast substring search algorithm. Communications of the ACM 33(8) (1990)
26. Vasiliadis, G., Polychronakis, M., Ioannidis, S.: Parallelization and characterization of pattern matching using GPUs. In: IEEE International Symposium on Workload Characterization (2011)