Regular Expression Search in Compressed Text

Extended Abstract

Pedro Valero*
IMDEA Software Institute, Spain
pedro.valero@imdea.org

Pierre Ganty
IMDEA Software Institute, Spain
pierre.ganty@imdea.org

Javier Esparza
Technical University of Munich, Germany
esparza@in.tum.de

ABSTRACT

Companies like Facebook, Google or Amazon store and process huge amounts of data. Lossless compression algorithms such as *brotli*[1] and *zstd*[2] are used to store textual data in a cost-effective way. On the other hand, it is common to process textual data using regular expression engines as evidenced by the number of highly-performant engines under development such as *Hyperscan*[3] or *RE2*[4]. For the scenario in which the input to the regular expression engine is only available in compressed form, the state of the art approach is to feed the output of the decompressor into the regular expression engine. We challenge this approach by searching directly on the compressed file. The experiments show that our purely sequential implementation, called *zearch*, outperforms the state of the art even when decompressor and search engine each have a distinct computing unit. Based on theoretical studies that suggest *zearch*'s algorithm is easily parallelizable, we take the challenge to boost the performance of *zearch* using parallel processing capabilities of modern architectures: SIMD instructions, multi-threading, multi-processing and general-purpose computing on GPUs.

CCS CONCEPTS

• **Theory of computation → Regular languages**; *Shared memory algorithms*; *Vector / streaming algorithms*; *Massively parallel algorithms*;

ACM Reference Format:

: Extended Abstract. In *Proceedings of 12th EuroSys Doctoral Workshop (EuroDW'18).* ACM, New York, NY, USA, 3 pages.

INTRODUCTION

Lossless compression of textual data is achieved by finding repetitions and replacing them by references to the repeated string. These strings are then encoded together with the remaining text as the output of the compression.

The state of the art approach to search on compressed text is to feed the output of the decompressor into the regular expression engine. Observe that with this approach the regular expression engine has no information about repetitions even though this information was computed by the compression algorithm.

Zearch's novelty is to take advantage of the information about repetitions to save on search work. As we describe next, *zearch* relies on well-established algorithms from Language Theory to report all lines of the original text containing a match of the regular expression. Our experiments, summarized in Table 1, show that *zearch* outperforms the state of the art approach even when comparing our sequential implementation with decompressing and searching in parallel.

*Corresponding author.

[1]https://github.com/google/brotli
[2]https://github.com/facebook/zstd
[3]https://github.com/intel/hyperscan
[4]https://github.com/google/re2

ZEARCH

Zearch operates over grammar-compressed text. Grammar-based compression algorithms, such as LZ78 [6], LZW [5], Recursive Pairing [2] and Sequitur [3], use repetitions in the text to build a context-free grammar as shown in Fig. 1.

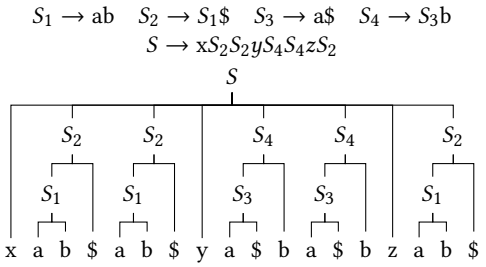

Figure 1: List of grammar rules (on top) generating the string "xab abyababzab$" (and no other) as evidenced by the parse tree (bottom).

In the settings of grammar-based compressed text, finding regular expression matches is conceptually close to the problem of deciding whether the languages of a context-free grammar (the compressed text) and an automaton (the regular expression) intersect. Indeed, if the two intersect it means there is a match of the regular expression in the original text.

Esparza et al. [1] proposed the *saturation construction* which allows to decide whether or not the intersection of a context-free grammar and an automaton is empty. The saturation construction processes grammar rules sequentially adding, for each rule, zero or more transitions to the automaton. We illustrate the construction by considering the rule $S_2 \to S_1\,\$$ of Fig. 1. If the automaton contains $(q_1) \xrightarrow{S_1} (q_2) \xrightarrow{\$} (q_3)$ for some q_1, q_2, q_3, then the saturation adds a transition $(q_1) \xrightarrow{S_2} (q_3)$. Intuitively, $(q_1) \xrightarrow{S_2} (q_3)$ means that the string "a b $" generated by variable S_2 labels a path in the automaton from state q_1 to state q_3. Thus deciding the emptiness of the intersection amounts to check whether there exists a transition from initial to final states with label S. Observe that the construction adds no states, only transitions.

	Subtitles			CSV			Log		
	zearch	Zgrep	PZgrep	zearch	Zgrep	PZgrep	zearch	Zgrep	PZgrep
"pedro"	239	304	**228**	454	397	**355**	**107**	258	183
"."	285	364	**269**	558	529	**405**	**125**	221	188
"I .* you"	380	382	**270**	517	372	**346**	**125**	261	185
"[a-z]{4}"	394	492	**347**	661	611	**466**	**155**	259	182
" [a-z]{4} "	**386**	688	518	532	698	**467**	**132**	403	255
"[a-z]{6}"	480	592	**430**	727	614	**469**	**172**	294	188
"[a-z]*[a-z]{10}"	567	658	**493**	852	616	**466**	**191**	436	293
"([a-z]{5})*"	400	357	**252**	540	519	**391**	**125**	219	189

Table 1: Running time (milliseconds) required for regular expression search on compressed text. Each column shows the running times required to search with a regular expression on a 100 MB long (uncompressed) file.* Zearch *uses a single process to search on the* repair*-compressed file.* Zgrep *and* PZgrep *decompress the file with* zstd *and search on the output with* grep*.* Zgrep *forces the decompression and search to be carried on in a single CPU while* PZgrep *imposes no such restriction. All tools are asked to report the number of lines on the uncompressed text containing a match of the regular expression. We underline the runtime of the fastest tool working on a single CPU and highlight in bold text the runtime of the fastest tool when there is no limitation in the number of CPUs used.

Finding all matches.

Tools like *grep*[5] or *ripgrep*[6] and others go beyond answering an emptiness problem and report all the lines in the original text containing a match of the regular expression. For the sake of conciseness we omit the description of how *zearch* reports all the matching lines and focus on how it counts them.

To count all matching lines, *zearch* attaches to each transition in the automaton extra information about potential match. Consider the grammar of Fig. 1 and let symbol "$" be an end-of-line delimiter so that the string generated by S consists of 5 lines. For the regular expression "a|b" we have that both symbols S_2 and S_4 generate a string matching against the expression. Indeed they both generate two matches of the expression although the string "ab$" generated by S_2 produces a single matching line while the string "a$b" generated by S_4 produces two.

Attaching to each transition information about the number of matching lines its label generates and defining the proper combination rules enables *zearch* to report the exact count of all matching lines.

Empirical Evaluation

Table 1 summarizes the results of searching compressed text with *zearch* and with the state of the art approach which feeds the output of the decompressor *zstd* into the regular expression engine *grep*.

We considered three types of textual data representative of the inputs fed into regular expression engines: English text, CSV and a computer generated log. Each table column corresponds to one text file that is compressed using the grammar-based compression algorithm *repair* [2] and *zstd* to feed our approach and the state of the art, respectively.

For the regular expressions we also aimed at a representative sample of commonly used regular expressions. The expressions considered range from simple patterns with few occurrences on the files to more complex expressions that match almost all lines of the uncompressed text.

Finally, we contrived two experiments to identify the situations where our approach vastly outperforms the state of the art. In the first experiment, we contrived textual data by repeating the same line over and over. Since the compressor turns 100 MB of repeated lines into a few bytes, *zearch* process it in little time (less than 3 milliseconds). In the second, we contrived a set of regular expressions of the form "[0-1]*1[0-1]{n}2" whose representation by a deterministic automaton is exponentially larger (in n) than representations by non-deterministic automata. By working directly on the non-deterministic representation *zearch* outperforms the state of the art which relies on the deterministic one. For instance, searching on a randomly generated string of the form "$\{0,1\}^{10^8}2$", *zearch* reports a match of the expression "[0-1]*1[0-1]{11}2" within 1.3 seconds while *grep* requires 7 min.

We have carried out more experiments varying the regular expressions used, the size of the uncompressed files and the type of files on which the search is done. The results of these experiments are publicly available through *zearch*'s website[7] as interactive graphs.

In summary *zearch*, which is purely sequential, outperforms the state of the art even when decompression and search are done in parallel. These results evidence the benefits of performing the search directly on the compressed representation of the data.

PARALLEL ZEARCH

Intuitively, rules like $S_2 \rightarrow S_1\$$ and $S_4 \rightarrow S_3 b$ from Fig. 1 can be processed in parallel since it will cause no concurrent reads and writes of any shared memory location. The same happens with rules $S_1 \rightarrow ab$ and $S_3 \rightarrow a\$$. In general, any set of rules such that the sets of symbols on the left and on the right hand side are disjoint can be processes simultaneously. Also each symbol could yield to parallel processing if the corresponding transitions in the automaton share no common state.

On the other hand, a theoretical result by Ullman et al. [4] on the parallelization of Datalog queries can be interpreted in our

[5] https://www.gnu.org/software/grep/
[6] https://github.com/BurntSushi/ripgrep
[7] https://pevalme.github.io/zearch/graphs/index.html

setting to show that the regular expression search on grammar-compressed text is in $\mathcal{NC}$[8] when the automaton built from the expression is acyclic. This result indicates some variant of our problem has efficient parallel solution.

Future Work

Nowadays, numerous options are available to compute in parallel: SIMD[9] instructions, multiple processes, multiple threads, compute kernel for GPU or FPGA…However, each comes with different restrictions and different guarantees which are difficult to compare, especially without prior experience. Incidentally, the success of moving to a particular parallel architecture often depends on subtleties: computing on GPU might result in no gain for small data because of the latency of the data bus.

Moreover, committing to a parallel computing platform often implies rethinking the algorithm and its data structures. For instance, the data layout can have a dramatic influence on the effectiveness of SIMD instructions. Thus to achieve a performance gain using SIMD one might have to re-define data structures.

REFERENCES

[1] Javier Esparza, Peter Rossmanith, and Stefan Schwoon. 2000. A Uniform Framework for Problems on Context-Free Grammars. *Bulletin of the EATCS* 72 (2000), 169–177.
[2] N Jesper Larsson and Alistair Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732.
[3] Craig G Nevill-Manning and Ian H Witten. 1997. Compression and explanation using hierarchical grammars. *Comput. J.* 40, 2_and_3 (1997), 103–116.
[4] Jeffrey D Ullman and Allen Van Gelder. 1988. Parallel complexity of logical query programs. *Algorithmica* 3 (1988), 5–42.
[5] Terry A. Welch. 1984. A technique for high-performance data compression. *Computer* 6, 17 (1984), 8–19.
[6] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory* 24, 5 (1978), 530–536.

[8] Solvable in poly-logarithmic time using a polynomial number of processors
[9] Single Instruction, Multiple Data