HIGH-PERFORMANCE PARALLEL REGULAR EXPRESSION MATCHING IN JULIA

ZACHARY DENG

Abstract. Regular expressions are a convenient way to describe many pattern matching tasks. Most widely available regex matching libraries, including Julia's, run serially and cannot take advantage of the parallelism available on modern CPUs. Additionally, there has not been a thorough performance analysis of the parallel algorithms available in the literature that compares their relative performance and characterizes the types of patterns and input data that each processes most efficiently. This work implements several parallel regular expression matching algorithms in Julia and analyzes the performance of each.

1. Introduction. Regular expressions have played an important role in the development of the theory of computation, and are also widely used to specify pattern matching problems. For example, they can be used to validate user input, perform lexical analysis in a compiler, and detect important events, such as signs of a network attack, in log files. Serial regular expression matching libraries are available in many languages; Julia's implementation is a wrapper around the Perl Compatible Regular Expressions library [2]. However, these serial implementations do not take advantage of the increasing number of processor cores available on modern CPUs, which prevents applications that require pattern matching over large quantities of textual data from easily scaling.

In response to this limitation, several parallel algorithms for matching regular expressions (or an equivalent problem, executing deterministic finite automata) have been developed. However, implementations of these are not widely available, so developers must devise parallelization strategies for their specific applications to achieve scalability. Additionally, it is unclear how the performance characteristics of these algorithms compare to each other, which would allow the selection of the most efficient algorithm for each class of patterns and input data. Thus, this project (1) implements several parallel regular expression matching algorithms in Julia and (2) provides a performance analysis for each of them, with recommendations about the most efficient ones for each use case. We find that the algorithms generally scale well as the number of processors increases, and identify one algorithm that outperforms Julia's baseline implementation, even when run serially.

Section 2 introduces constructs from automata theory and describes how they can be used to accomplish parallel regex matching. Section 3 provides details about the implementation of these algorithms in Julia. Section 4 presents performance results for a benchmark dataset.

2. Algorithms.

2.1. Regular expressions and finite automata.

We formally define a pattern of interest as a language.

DEFINITION 2.1. *Given an alphabet Σ (a set of symbols), a language L over this alphabet is a set of strings consisting of symbols in Σ.*

Thus, each pattern matching task has a corresponding language consisting of all of the strings that match the pattern. A restricted set of languages known as regular languages can be represented by regular expressions. Regular expressions can be defined recursively, and they have the following string representation.

DEFINITION 2.2. *The language of regular expressions is defined by the following context-free grammar:*

1. `S ::= '(' S ')'`
2. `S ::= S S`
3. `S ::= S '|' S`

4. `S ::= S '*'`
5. `S ::= 'a'`
6. ...

Production 1 is used to override default operator precedence. Production 2 will match any instance of a string from the language defined by the left subexpression, followed by any instance of a string from the language defined by the right subexpression. Production 3 represents an alternation between the two options: it will match the union of the languages of the two expressions. Production 4, known as the Kleene star, represents matching the concatenation of zero or more strings from the subexpression's language. Finally, the terminals of the grammar represent languages of size 1 containing only the corresponding character.

Some regular expression libraries contain extensions for more concisely representing commonly used patterns; for example, `[abc]` represents the same expression as `'a' | 'b' | 'c'`. However, these extensions can be decomposed into combinations of the primitive operators listed in Definition 2.2. Other extensions, such as backreferences, can be used to define non-regular languages and cannot be represented as standard regular expressions. These cannot be matched using the algorithms described in this paper, and are out of scope.

A way to test if a string is present in a regular language is to use finite automata.

DEFINITION 2.3. *A nondeterministic finite automaton (NFA) consists of a set of states Q, an alphabet Σ, a set of transitions $\delta \subseteq \{(q_1, q_2, c) | q_1, q_2 \in Q, c \in \Sigma \cup \{\epsilon\}\}$, a set of starting states $I \subseteq Q$, and a set of final states $F \subseteq Q$.*

A string matches a NFA if it is possible to start at a state in I, take a sequence of edges, where each edge (q_1, q_2, c) represents a transition from state q_1 to q_2 that consumes the character c from the start of the string, or leaves the string unchanged if $c = \epsilon$, and end at a state in F when the remaining string is empty.

It is easy to convert any regular expression into an NFA that recognizes it, using the McNaughton-Yamada-Thompson algorithm [5]. Note that for a given q_1 and c, an NFA may contain edges to multiple destination states q_2, so there may be multiple paths through the graph for any given string. Additionally, it may be possible to have a choice between consuming a character or using an ϵ-transition, if it exists. Thus, it is not convenient to determine whether an NFA accepts a string. However, once the NFA is constructed, it is possible to construct another automaton, known as a deterministic finite automaton in which the path, if it exists, is unique. This process is shown in Algorithm 2.1.

DEFINITION 2.4. *A deterministic finite automaton (DFA) consists of a set of states Q, an alphabet Σ, a set of transitions $\delta \subseteq \{(q_1, q_2, c) | q_1, q_2 \in Q, c \in \Sigma\}$, a starting state q_i, and a set of final states $F \subseteq Q$. Additionally, there must be at most one transition for every pair (q_1, c).*

With this definition, each transition consumes exactly one character from the string, and there is a unique path through the string, if it exists. Algorithm 2.2 shows a simple algorithm for matching a string, which takes $O(|s|)$ time.

2.2. Parallel DFA computation. Now that we have described the construction of a DFA from any regex, we have reduced the problem of parallel regex matching to that of parallelizing the execution of DFAs. Each algorithm discussed in the following section either takes as input a DFA directly, or constructs a DFA through the process described above. Thus, we will describe only the way each executes these DFAs in parallel.

The simple DFA execution algorithm does not easily parallelize because the computation of the transition for each character requires knowing the ending state at the previous character. A variety of algorithms have been developed to work around this problem.

Algorithm 2.1 Converting NFA to DFA (subset construction)

Input: NFA $(Q, \Sigma, \delta, q_i, F)$
Output: DFA $(Q', \Sigma, \delta', q_i', F')$ that recognizes the same language as the NFA

```
worklist ← {{q_i}}
visited ← {{q_i}}
Q' ← {}
δ' ← {}
q_i' ← {q_i}
while worklist nonempty do
  S ← dequeue(worklist)
  next ← {}
  for c ∈ Σ do
    for q ∈ S do
      if δ has transition from q with character c to q_d then
        next ← next ∪ {q_d}
      end if
    end for
    if next ∉ visited then
      q' ← new DFA state for next
      Q' ← Q' ∪ {q'}
      visited ← visited ∪ {next}
      worklist ← worklist ∪ {next}
    else
      q' ← existing DFA state corresponding to next
    end if
    v' ← DFA state corresponding to S
    δ' ← δ' ∪ {(v', q', c)}
  end for
end while
F' = {S|S ∈ Q' and at least one element of S is in F}
```

2.2.1. Enumeration method [4]. One of the earliest methods of parallelizing DFA execution involves speculative execution. First, the input string is split into one contiguous chunk for each processor. Then, each processor executes the DFA starting from every possible initial state on the corresponding chunks. This results in a map from the starting state at the start of the chunk to the ending state after the chunk, for every chunk. Finally, paths through the entire DFA are constructed by combining the solutions of adjacent chunks. The full algorithm is shown in Algorithm 2.3.

Each chunk is processed in full $|Q|$ times, once for each starting state, and processing each chunk takes time linear in the length of the chunk. Additionally, it takes time linear in the number of processors to reduce the results. It is theoretically possible to accomplish this reduction in logarithmic time with respect to the number of processors, by reducing adjacent pairs of results in a bottom-up manner, for a total reduction runtime of $O(|Q| \lg p)$, but since we assume the number of processors is small relative to the size of the string, this would likely be slower than the serial reduction due to synchronization and communication overheads. Thus, the runtime of this algorithm is $O(|Q||s|/p + p)$. Note that if $|Q| \gg p$, the work overhead of this algorithm is very large and it will likely execute more slowly than the serial algorithm.

Algorithm 2.2 Executing DFA on a string

Input: DFA $(Q, \Sigma, \delta, q_i, F)$, string s
Output: whether DFA accepts s

```
q ← q_i
while s nonempty do
  if (q_i, q_2, s[0]) ∈ δ for some q_2 then
    q ← q_2
    remove first character from s
  else
    return false
  end if
end while
return q ∈ F
```

2.2.2. PaREM [6]. The algorithm introduced in this paper improves upon the naive speculative execution algorithm by pruning the set of initial states that each processor must consider. This set is calculated using the set of transitions that could occur between adjacent chunks. In particular, the starting state for a chunk must be one that can be reached after transitioning along an edge corresponding to the previous chunk's last character, and it must be possible to transition from the starting state using an edge corresponding to the current chunk's first character. It would be possible to generalize this pruning to consider the k characters at the end of the previous chunk and the start of the current chunk; however, it is likely that considering additional characters would result in diminishing returns.

The pruning process is shown in Algorithm 2.4, and is done by each individual processor. Note that the pruned states depend on the input string s; thus, they must be computed online, which takes resources away from the actual executions of the DFA. The rest of the algorithm is the same as in Algorithm 2.3.

In the worst case, this algorithm is equivalent to Algorithm 2.3 because no starting states can be pruned from each chunk. However, in practice it is possible that a large portion of states are impossible to reach from a given chunk boundary, and thus this method would give a large speedup.

2.2.3. Simultaneous finite automata [7]. This method also takes the approach of splitting an input string into one chunk per processor, executing on the chunks in parallel, and finally reducing the results. However, instead of executing the original DFA once starting from each possible starting state (or a pruned subset of the states), a new automaton is constructed, called a simultaneous finite automaton (SFA), which is itself a DFA. In the SFA, states correspond to functions $Q \mapsto Q$ in the original DFA, so that executing the SFA once computes the same mapping that Algorithm 2.3 used $|Q|$ executions of the original DFA to compute. Once the SFA is computed once on each chunk, the resulting functions can be composed to produce a function that when applied to the starting state of the original DFA, gives the final state after executing it on the entire string.

From a DFA, an SFA can be constructed in a very similar manner as the NFA-to-DFA construction. In the NFA-to-DFA construction, DFA states correspond to sets of NFA states, while in the DFA-to-SFA construction, SFA states corresponding to functions $Q \mapsto Q$ over the original DFA states. The construction is shown in Algorithm 2.5.

The SFA can be precomputed ahead of time, because it only depends on the DFA. After it is constructed, matching a string is accomplished in a very similar way as Algorithm 2.3, except each processor only needs to start at the SFA start state, which is the identity map.

Algorithm 2.3 Speculative execution of DFA

Input: DFA $(Q, \Sigma, \delta, q_i, F)$, string s, number of processors p
Output: whether DFA accepts s

```
results ← p × |Q| array
parallel for worker ∈ {1, ..., p} do
  // speculatively execute from all starting states
  for q ← Q do
    q_i ← q
    s_i ← chunk for this worker
    while s_i nonempty do
      if (q_i, q_2, s[0]) ∈ δ for some q_2 then
        q_i ← q_2
        remove first character from s_i
      else
        q_i ← null
        break
      end if
    end while
    results[worker][q] ← q_i
  end for
end for
// reduce chunk results
q ← q_i
for worker ∈ {1, ..., p} do
  q ← results[worker][q]
  if q is null then
    return false
  end if
end for
return q ∈ F
```

Computing the final result is also accomplished with the same reduction. This is illustrated in Algorithm 2.6.

This algorithm has the advantage that each processor only needs to execute its automaton once, while the other approaches can in the worst case take time proportional to the number of DFA states. It requires $O(|s|/p + p)$ time, where the first term represents the computation of the ending SFA state for each chunk, and the second term represents the reduction time (again, it's possible to accomplish the reduction in $O(|Q| \lg p)$ time if using a very large number of processors, which may occur in a distributed system). However, the number of SFA states can be exponential in the number of DFA states, which may result in poor cache performance.

3. Implementation.

Each of these algorithms was implemented in a Julia package.

3.1. Regex parsing.

Regular expressions are generally specified in the string representation given in Definition 2.2, so they must be parsed into an abstract syntax tree before conversion into an NFA. This is accomplished with the ParserCombinator.jl package [3], which provides convenient syntax for specifying a recursive descent parser. Only the most common regular expression extensions are supported, because the focus of this work is on the parallel evaluation of automata, and the algorithms to be evaluated only substantially differ once a DFA has been constructed.

Algorithm 2.4 PaREM pruning

Input: DFA $(Q, \Sigma, \delta, q_i, F)$, previous chunk s, current chunk t
Output: set of starting states that must be considered

```
S ← ∅
if no previous chunk then
  S ← {q_i}
else
  for q ∈ Q do
    if δ contains a transition from q using the last character of the previous chunk then
      S ← S ∪ {q}
    end if
  end for
end if
T ← ∅
for q ∈ Q do
  if δ contains a transition from q using the first character of the current chunk then
    T ← T ∪ {q}
  end if
end for
return S ∩ T
```

3.2. Automata construction and representation. The automata for each regular expression can be computed ahead of time and stored for later use, so the efficiency the construction algorithms is not the greatest priority. Automata are represented in a pointer-based graph structure, where each state stores a lookup table, indexed by a single byte, that contains either a pointer to the destination of the corresponding transition, or a `Missing` value to indicate that the transition does not exist. This, rather than a sparse representation such as an adjacency list, was chosen because SFA graphs are dense.

3.3. Parallel matching procedures. Each of the three parallel DFA execution algorithms accepts as input the corresponding precomputed data structure and a string represented as a byte vector. The chunk boundaries are computed by the main thread, but all workers read from the original vector to prevent memory copies from being a bottleneck. Parallelism is achieved by using the Threads.jl library on a loop that computes the result for each subtask. After this completes, the reduction is performed by the main thread and the final result is returned.

4. Evaluation. Past work has not provided a clear analysis of the performance characteristics of these algorithms, especially as compared to each other. Thus, we are interested in several attributes. First, speculative execution incurs a large work overhead, proportional to the number of DFA states $|Q|$, to achieve parallelism. We will evaluate whether realistic regular expressions can be converted into sufficiently small DFAs to make parallelism effective. Additionally, we will evaluate whether the pruning done by PaREM provides a large enough speedup in general scenarios to outweigh the cost of computing the reduced starting state sets.

We will also evaluate the empirical scalability of each of these methods. In theory, each should achieve close to linear speedup as the number of processors increases, since the cost of reduction is negligible, but there may be other bottlenecks such as memory bandwidth that prevents this from being realized. A related question is the cache usage of each method; SFAs can be exponentially larger in the worst case than the DFAs they are constructed from, so executing an SFA may cause the graph data structure to move into a slower level of the cache

Algorithm 2.5 Converting DFA to SFA

Input: DFA $(Q, \Sigma, \delta, q_i, F)$
Output: SFA $(Q', \Sigma, \delta', q_i', F')$

```
worklist ← {I} // start with the identify map, which is the state function when zero characters are consumed
visited ← {I}
Q' ← {}
δ' ← {}
q_i' ← I
while worklist nonempty do
  f ← dequeue(worklist)
  next ← I
  for c ∈ Σ do
    for q ∈ S do
      if f(q) ≠ null then
        next(q) ← ending state of transition using c from f(q) in δ, or null if not present
      else
        next(q) ← null
      end if
    end for
    if next ∉ visited then
      q' ← new SFA state for next
      Q' ← Q' ∪ {q'}
      visited ← visited ∪ {next}
      worklist ← worklist ∪ {next}
    else
      q' ← existing SFA state corresponding to next
    end if
    v' ← SFA state corresponding to f
    δ' ← δ' ∪ {(v', q', c)}
  end for
end while
F' = ∅
```

hierarchy, limiting scalability.

Finally, the absolute runtimes will be compared to the baseline, which is Julia's serial PCRE wrapper, to determine whether the magnitude of the speedup is worth accepting the more limited functionality of these DFA-based methods.

4.1. Datasets and hardware. To evaluate the performance of each of these algorithms, the benchmark dataset is the Dotstar subset of the ANMLzoo automata benchmark suite [8]. This consists of a set of regular expressions based on those used in a network intrusion detection system. Additionally, the dataset contains two test strings containing network traces, one of size 1MB and one of size 10MB. The benchmarks are run on an AMD EPYC 7R13 CPU and the Ubuntu 20.04 operating system. All timings are measured using the BenchmarkTools.jl library [1] to reduce variance due to just-in-time compilation and caching.

4.2. Scalability. Each of the four algorithms (speculative execution, PaREM, SFA, and the PCRE baseline) was used to match regular expressions sampled from the dataset to both the 1MB and 10MB test strings, using 1, 2, 4, 8, and 16 threads to measure whether the

Algorithm 2.6 Execution of SFA

Input: SFA $(Q, \Sigma, \delta, q_i, F)$, DFA $(Q', \Sigma, \delta', q_i', F')$, string s, number of processors p
Output: whether corresponding DFA accepts s

```
results ← length p array
parallel for worker ∈ {1, ..., p} do
  q_i ← I // start only from the identity map
  s_i ← chunk for this worker
  while s_i nonempty do
    if (q_i, q_2, s[0]) ∈ δ for some q_2 then
      q_i ← q_2
      remove first character from s_i
    else
      // this branch is never reached because the SFA has a transition for every character
      at each state
    end if
  end while
  results[worker] ← q_i
end for
// reduce chunk results
q ← q_i'
for worker ∈ {1, ..., p} do
  q ← results[worker](q)
  if q is null then
    return false
  end if
end for
return q ∈ F'
```

implementations achieve the theoretical linear speedup. The speedup results for each regex are aggregated by taking the geometric mean. Figure 1 shows the speedup for each method compared to its speed when run with only one thread on the 1MB string and Figure 2 shows the speedup when run on the 10MB string.

On the 1MB string, both the speculative and PaREM methods display linear speedup up to 8 threads, but this diminishes as the thread count is increased to 16. The SFA algorithm does not achieve linear speedup even for two threads, and increasing the number of threads beyond two has no effect on the runtime. This difference is likely due to the small size of the tested task; since the speculative and PaREM methods have a large work overhead, their effective problem size is larger than the SFA method, so that the overhead associated with launching threads and collecting results is negligible compared to each thread's subtask. Another contributing factor may be that SFAs are generally larger in terms of the number of states than the corresponding DFAs; thus, executing SFAs can cause more cache misses and use more memory bandwidth, which can become a bottleneck as the number of processors increases.

However, when the problem size is increased to 10MB, we observe that the speculative and PaREM methods scale close to linearly even up to 16 threads, and the scaling of SFA is much closer to linear. This indicates that memory bandwidth is not a significant bottleneck for the execution of SFAs of this magnitude. We conclude that parallel regex matching is more applicable to larger problems (having at least an order of magnitude of tens of megabytes).

FIG. 1. *Speedup over single-thread execution for 1MB string*

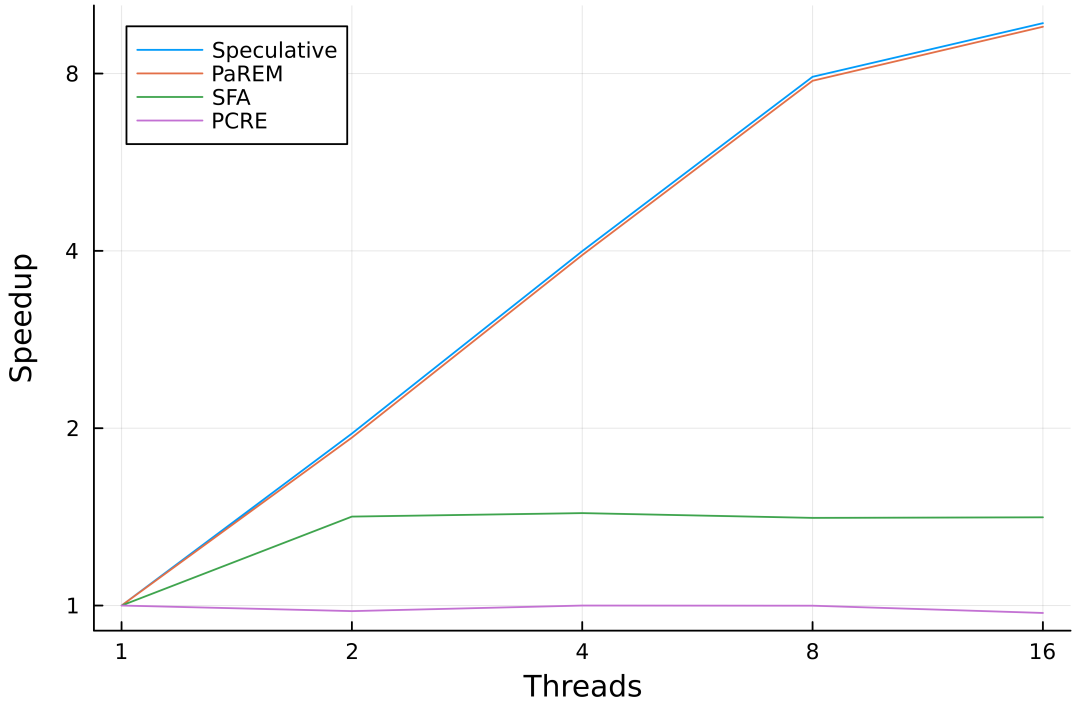


FIG. 2. *Speedup over single-thread execution for 10MB string*

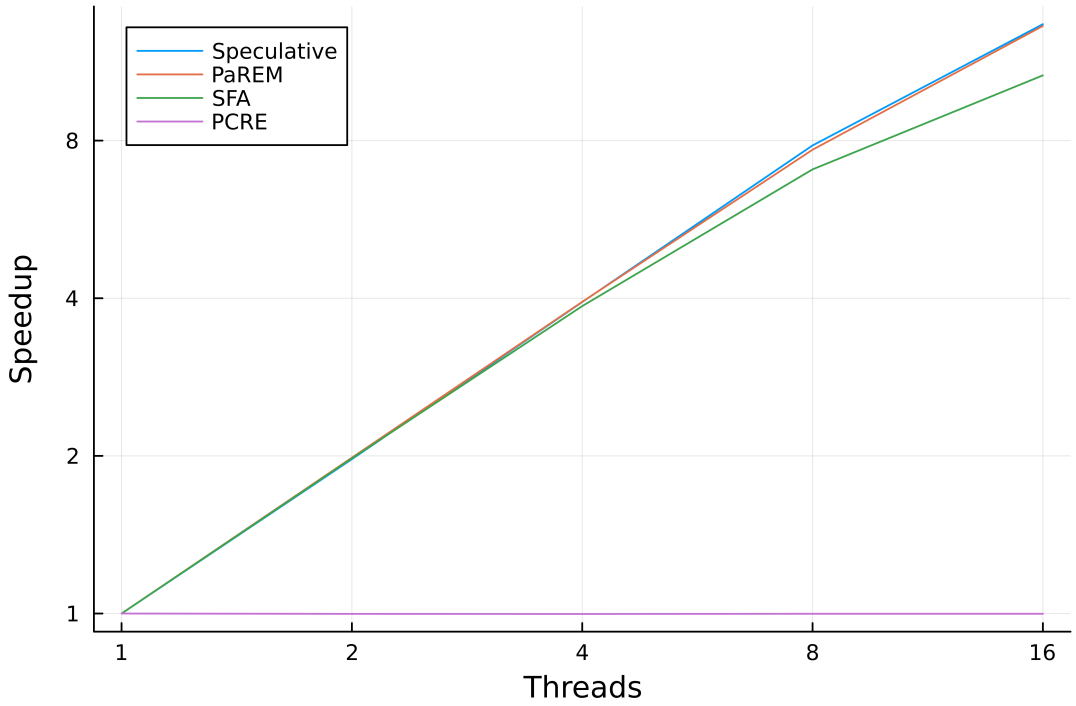
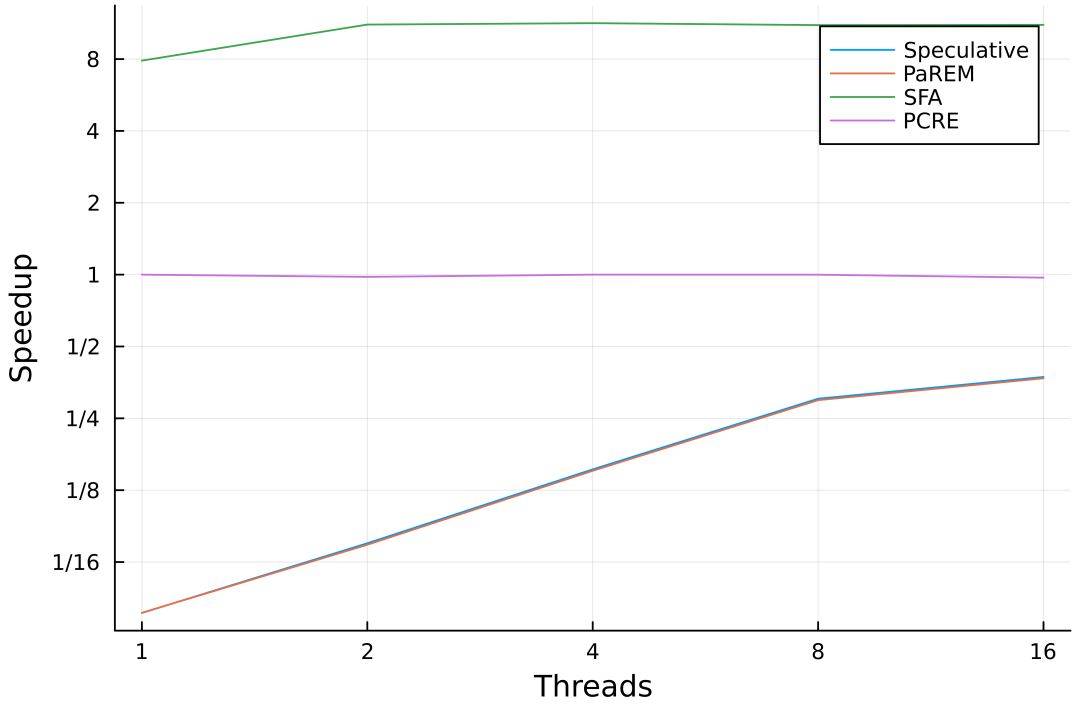


FIG. 3. *Speedup over PCRE for 1MB string*



4.3. Runtime. Each of the experiments of the previous section also recorded absolute runtimes, which are now compared to the serial baseline in Figures 3 and 4.

We observe that the work overhead of the speculative and PaREM methods is quite large in practice, and that even with 16 threads, the runtime remains worse than the serial baseline. This indicates that the regexes in our benchmark dataset form DFAs that are relatively large compared to the number of processors available. Additionally, the pruning performed by PaREM does not appear to result in a measurable speedup. The cases it is able to prune may occur too infrequently in our benchmark dataset, or the cost of performing this pruning (which must happen at runtime) might exceed the savings of the pruned states. However, even when run serially, the SFA method outperforms the baseline by a factor of 8, which is likely due to the extremely simple inner loop that processes a single character. The regex extensions that PCRE supports requires it to track more state during execution, which incurs a performance penalty. Thus, this method provides a speedup even if it is not known in advance whether multiple processors are available, so users do not need to implement a fallback to PCRE in the case of serial execution.

4.4. Automata size. To better understand the performance characteristics of these algorithms, it is useful to consider the size of the underlying automata, which impacts how many states can be preserved in each level of cache during execution. The sizes of the constructed DFA (used for the speculative execution and PaREM methods) and the corresponding SFA (used by the SFA method) were computed for each of the regexes tested above. The relationship between the two is displayed in Figure 5. Although in theory a DFA with $|Q|$ states may require an SFA with $|Q|^{|Q|}$ states in the worst case [7], in practice this exponential behavior does not seem to occur. In fact, the SFA size appears to grow polynomially, with a growth

FIG. 4. *Speedup over PCRE execution for 10MB string*

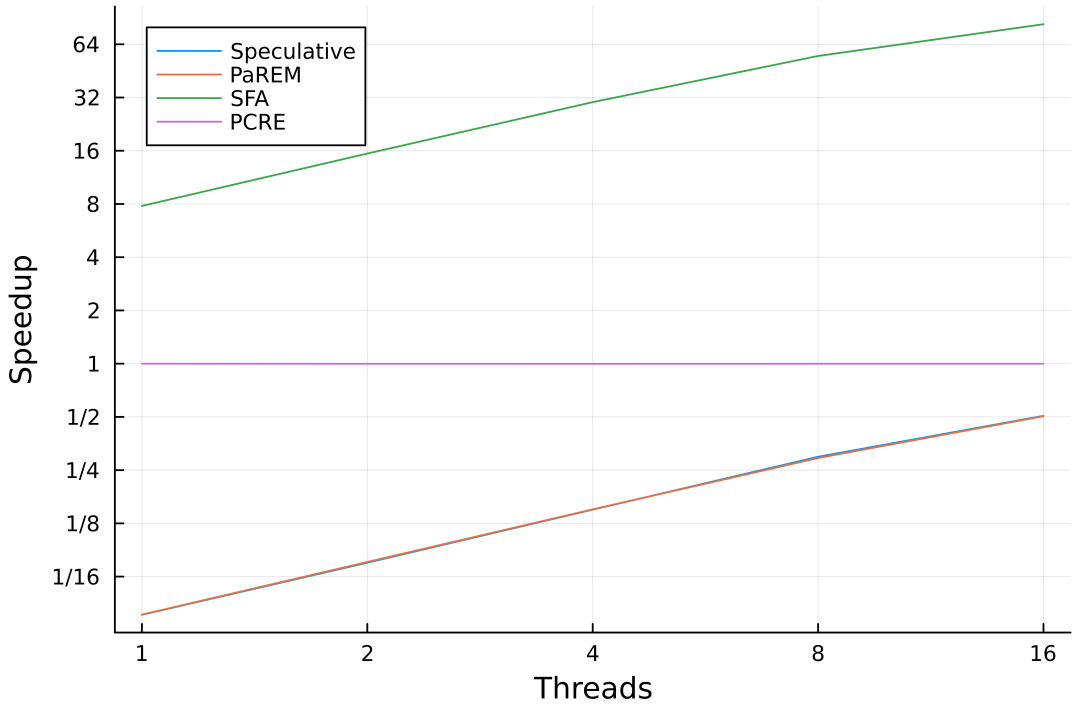
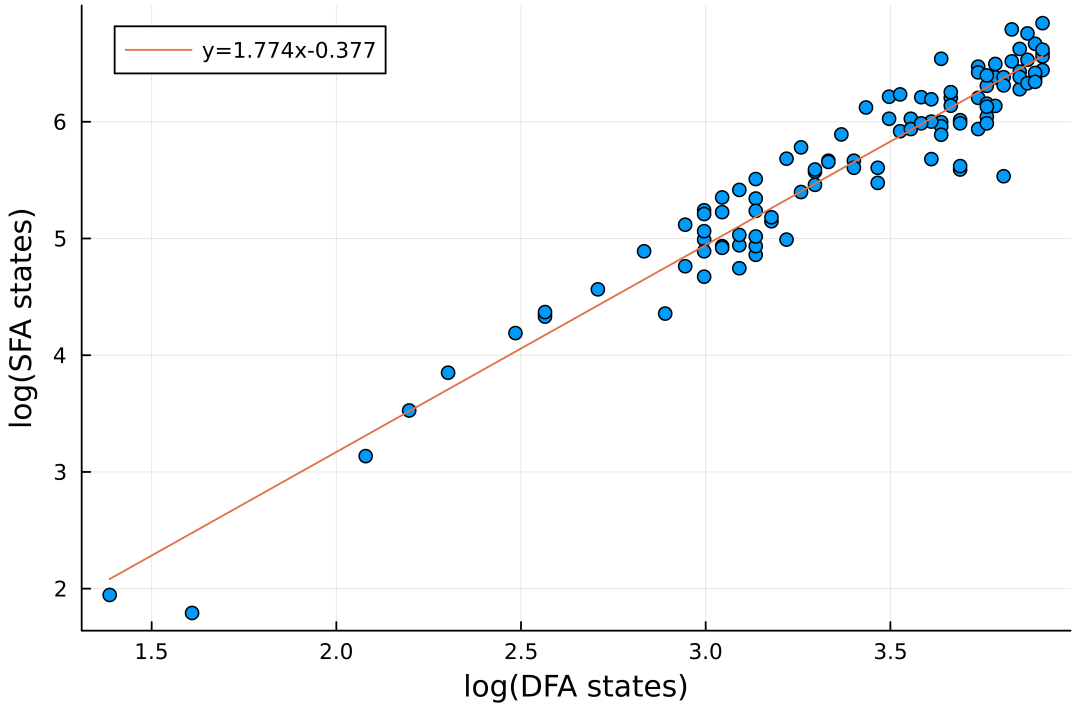


FIG. 5. *SFA size vs DFA size*



rate of approximately $O(|Q|^{1.77})$. Thus, this method should not have significant issues scaling to larger, more complex regular expressions.

5. Future work. A direction for future work may be to extend the frontend of the library (which parses regex strings into an abstract syntax tree) to support a greater subset of commonly used syntactic sugar. This would allow the library to support a greater variety of regular expressions, without needing any preprocessing to desugar these constructs. Another useful feature would be selecting an optimal algorithm to use at runtime, depending on the characteristics of the input string and the pattern. For example, very simple expressions with few DFA states may run faster using speculative execution than the SFA method even after accounting for work overhead, since the DFA may fit entirely in a faster level of cache than the SFA. Further experimentation can also be done with the representations of SFAs in memory; for example, storing the DFA mapping separately from the lookup table will change the memory access pattern. Finally, the algorithms for constructing the various automata from each regex can be further optimized. This was not in scope of this work, but doing so would improve usability for very large patterns and decrease latency in the scenario where the patterns are not known ahead of time.

6. Conclusions. This work implements several parallel regular expression matching algorithms in Julia. We confirmed that given sufficient data, all three algorithms achieve close to the theoretical linear speedup as the number of processors increases, and showed that one of the algorithms, SFA matching, significantly outperforms Julia's built-in regex library, even when running serially. This speedup results from a compact data structure representation, keeping the hot loop as simple as possible, and taking advantage of parallelism when available. Additionally, we found that the work overhead of the speculative execution based methods generally results in worse performance than the serial baseline, so they should only be used on very simple patterns. Finally, we replicated the finding in [7] that SFAs empirically grow much more slowly than in the theoretical worst case.

The implementations are available at https://github.mit.edu/dengzac/18.337-project.

REFERENCES

[1] *BenchmarkTools.jl*, https://github.com/JuliaCI/BenchmarkTools.jl (accessed 2023-05-17).

[2] *Strings · The Julia Language*, https://docs.julialang.org/en/v1/manual/strings/#man-regex-literals (accessed 2023-03-24).

[3] A. COOKE, *ParserCombinator*, https://github.com/andrewcooke/ParserCombinator.jl (accessed 2023-05-17).

[4] J. HOLUB AND S. ŠTEKR, *On Parallel Implementations of Deterministic Finite Automata*, in Implementation and Application of Automata, S. Maneth, ed., Lecture Notes in Computer Science, Springer, pp. 54–64, https://doi.org/10.1007/978-3-642-02979-0_9.

[5] R. MCNAUGHTON AND H. YAMADA, *Regular Expressions and State Graphs for Automata*, EC-9, pp. 39–47, https://doi.org/10.1109/TEC.1960.5221603.

[6] S. MEMETI AND S. PLLANA, *PaREM: A Novel Approach for Parallel Regular Expression Matching*, in 2014 IEEE 17th International Conference on Computational Science and Engineering, pp. 690–697, https://doi.org/10.1109/CSE.2014.146, http://arxiv.org/abs/1412.1741 (accessed 2023-03-24), https://arxiv.org/abs/1412.1741.

[7] R. SINYA, K. MATSUZAKI, AND M. SASSA, *Simultaneous Finite Automata: An Efficient Data-Parallel Model for Regular Expression Matching*, in 2013 42nd International Conference on Parallel Processing, pp. 220–229, https://doi.org/10.1109/ICPP.2013.31.

[8] J. WADDEN, V. DANG, N. BRUNELLE, T. T. II, D. GUO, E. SADREDINI, K. WANG, C. BO, G. ROBINS, M. STAN, AND K. SKADRON, *ANMLzoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures*, in 2016 IEEE International Symposium on Workload Characterization (IISWC), pp. 1–12, https://doi.org/10.1109/IISWC.2016.7581271.