

Parallelising Brzozowski's DFA Construction Algorithm in Haskell

Conventional research project

Joshua Clark

537660

Supervised by Harald Søndergaard



COMP90055: Computing Project (25 points)

The University of Melbourne

Semester 1, 2019

Parallelising Brzozowski’s DFA Construction Algorithm in Haskell

Abstract

Regular expressions are used in a variety of different domains. They provide an expressive syntax for describing formal languages, i.e., sets of strings. Brzozowski gives an elegant algorithm for directly constructing a deterministic finite automaton from a regular expression, based on the concept of a regular expression derivative. The functional programming paradigm is already a natural choice for an implementation of Brzozowski’s algorithm, given the inductive specification of the derivative function. Exploring what resources the functional programming paradigm can also offer attempts at parallelism is the primary objective of the present discussion. Algorithms dedicated to converting regular expressions into equivalent automata have a long, well-studied history. However, relatively less attention has been paid to the benefits that might be gained from parallel approaches to this task. Paying attention to parallelism is important at a time where computers become more powerful from parallel hardware, not higher clock speeds. We evaluate the performance of three implementations in the purely functional language Haskell in terms of speed-up and efficiency, and compare the findings to prior work presenting a concurrency-based implementation. The results suggest that while both approaches are capable of achieving increased performance through parallelism, their particular characteristics may make them most appropriate in different scenarios.

Contents

1	Introduction	1
2	Preliminaries	2
2.1	Regular expressions	2
2.2	A note on “extended” regular expressions	3
2.3	Deterministic finite automata	3
3	Derivatives of regular expressions	4
3.1	Similarity of regular expressions	5
3.2	An example	6
4	Towards parallelism	8
4.1	Parallelism versus concurrency	8
4.2	The concurrent, process-oriented specifications of Strauss et al.	9
5	Shifting to a functional context	10
5.1	Evaluation strategies	11
5.2	Dataflow parallelism and the <code>Par</code> monad	12
6	Performance evaluation	14
6.1	Experimental setup	14
6.2	Observations	15
7	Conclusions	21
7.1	Future work	21

1 Introduction

Regular expressions are of fundamental importance to a variety of computational endeavours including lexical analysis, text processing and information retrieval amongst others. They provide an expressive syntax to describe formal languages, i.e., sets of strings. In this paper we are concerned with approaches to implementing regular expression matching – a concrete means for determining whether a given input belongs to the language described by a regular expression. There are, broadly speaking, two divergent approaches to implementing such matching which amount to either using recursive backtracking or constructing a finite automaton capable of recognising the same language.

The main focus of this paper lies with automaton based implementations, specifically Brzozowski's method based on derivatives of regular expressions. As can be imagined for an area with as long a history, much research has been done into the techniques for implementing regular expression matching. Consequently, there are a number of well-studied algorithms dedicated to converting regular expressions into equivalent automata. Indeed, the method we are considering was first introduced in 1964. Despite this long history, relatively less attention has been paid to the benefits that might be gained from parallel approaches to this task.

Potential for parallelism is an important consideration for the current computational climate where individual processors are not increasing in power, but computers are becoming potentially more powerful by utilising multiple processors. We can describe this situation as one in which computers are becoming wider, rather than faster. To benefit from this situation we need to consider algorithms with potential to scale out rather than scale up.

Exploring the resources offered by the functional programming paradigm for achieving parallelism is the primary objective of the present discussion – specifically, how a paradigm which draws a distinction between pure and side-effecting code allows us to draw a stronger distinction between *concurrency* and *parallelism* than is typical of other paradigms. In order to address this primary aim we also consider the effect on attempts at parallelism introduced by different algorithm and input parameters, namely, regular expression complexity, alphabet size and notions of regular expression similarity. We focus on efforts to parallelise one particular algorithm, but it is expected that doing so allows more general insights into the behaviour of functional parallelism to be gained. We survey two Haskell libraries offering different perspectives of realising deterministic parallelism, namely evaluation strategies and the `Par` monad. These two approaches provide the foundation of our own parallel implementations of Brzozowski's algorithm.

We find that utilising the parallelism facilities available to the Haskell programmer we are able to obtain increased performance over our test cases. Compared to a sequential baseline, the maximum speed-up achieved by any of our parallel implementations is 2.95 utilising 4 cores. In terms of effect of input characteristics, considerable difference in performance was observed relative to alphabet size. Our findings in this respect run contrary to those put forth in a previous study [14], and we discuss possible causes underlying this discrepancy. Overall, the results suggest that while approaches utilising concurrent specifications and those using deterministic parallelism are capable of achieving increased performance, their particular characteristics may make them most appropriate for different use cases.

The reader is assumed to have some familiarity with Haskell. Fundamentals of the language such as syntax of function applications and type definitions are not covered, and section 5 makes explicit use of code examples in explaining the different approaches to parallelism we consider. The approaches themselves depend on language features such as type classes and monads, but for our present purposes familiarity with `do` notation and understanding type classes as a way to implement polymorphism should be enough to follow the discussion.

The remainder of the paper proceeds according to the following structure. Section 2 introduces required background and theoretical preliminaries. Specifically, we introduce regular expressions and deterministic finite automata (DFA) and the notations we use to describe them. Section 3 explains how Brzozowski's concept of derivatives of a language can be used as the basis for a DFA construction algorithm. We also discuss the main parameter of the algorithm we consider, namely

how regular expressions should be tested for equivalence. Section 4 turns attention to parallelism and concurrency. We consider the prior work of Strauss et al. [13, 14, 15], which is aimed at defining and evaluating a concurrency-based specification of Brzozowski’s algorithm – and provides a major point of comparison for our own implementations. Section 5 presents the case for differentiating between the sometimes conflated concepts of concurrency and parallelism, especially in a functional context. The relationship of parallelism to Haskell’s *purity* and *laziness* is considered in our survey of two parallelism-oriented libraries. Section 6 describes our experimental methodology for evaluating the various implementations and details the observations that were made. Finally, Section 7 presents conclusions drawn from the research and suggests avenues for further research lying outside the scope of the present paper.

2 Preliminaries

2.1 Regular expressions

In what follows we consider a language, L , to be a set of strings over some finite alphabet Σ . In other words, $L \subseteq \Sigma^*$. In the present context, we are concerned with *regular languages*, i.e., languages which can be expressed by a regular expression. We use uppercase R, S and T to stand for arbitrary regular expressions and lowercase r, s, t for arbitrary strings.

A regular expression consists of two different and disjoint classes of characters. Firstly, we have \emptyset meaning the empty language, ε meaning the set containing only the empty string and the literal symbols taken directly from the given alphabet, $a \in \Sigma$ meaning the singleton set of a string that is a single instance of a . Secondly, we have the metacharacters which represent operations we can perform on the sets of strings represented by the regular expressions. The set of metacharacters are not absolutely fixed because it is possible to introduce new symbols representing new operations we can perform. However the following symbols are those relevant to the present discussion: $(,)$, $*$, \cdot , $+$, $\&$, and \neg .

Kleene star $(*) : R^* = \{r_1 \cdot r_2 \cdot \dots \cdot r_k \mid k \geq 0 \text{ and each } r_i \in R\}$

Concatenation $(\cdot) : R \cdot S = \{r \cdot s \mid r \in R, s \in S\}$

Alternation $(+) : R + S = \{r \mid r \in R \vee r \in S\}$

Intersection $(\&) : R \& S = \{r \mid r \in R \wedge r \in S\}$

Complement $(\neg) : \neg R = \{s \mid s \in \Sigma^* \wedge s \notin R\}$

A regular expression (with respect to alphabet Σ) is well formed if it adheres to the following recursive definition:

1. \emptyset, ε and $a \in \Sigma$ are regular expressions.
2. If R and S are regular expressions then: R^* , $(R \cdot S)$, $(R + S)$, $(R \& S)$ and $\neg R$ are regular expressions.

Anything that is not the result of a finite application of the above definitions is not a well-formed regular expression.

In the interest of conciseness we will often represent concatenation implicitly (i.e., $R \cdot S$ can be written RS) and consider the precedence of the operators from highest to lowest as: \neg , $*$, \cdot , $\&$, $+$. This means parentheses can be omitted when the meaning of a regular expression follows these priorities. For example, $((a \cdot b)^* \cdot (c)^*)$ could be expressed as $(ab)^*c^*$.

For an alphabet defined as $\Sigma = \{a, b, c\}$, we offer some simple examples of regular expressions and the sets they represent below:

$$\begin{aligned} a &= \{a\} \\ ab + c &= \{ab, c\} \\ ac(ab)^* &= \{ac, acab, acabab, acababab, \dots\} \end{aligned}$$

2.2 A note on “extended” regular expressions

For Kleene [6], a regular expression made use of just the three “regular” operations: \cdot , $+$ and $*$. The reader will have noticed that we also allow $\&$ and \neg . In some contexts, “extended” regular expressions are taken to mean those for which the operation of intersection [1] or both intersection and complementation operations [8] are defined. In another sense, “extended” regular expressions can refer to POSIX Extended Regular expressions. These are a number of syntactic extensions that offer a degree of conciseness, for example using $\{n\}$ to indicate n repetitions of a regular expression, or the $?$ operator to signal zero or one instances. These latter kind of regular expressions offer extra conciseness, however, we can still capture their semantics with our working definition, e.g., $R?$ can be captured by $\varepsilon + R$ or $R\{n\}$ can be expanded out to R concatenated to itself $n - 1$ times.

Note that both intersection and language complement remain aligned with the more strictly theoretical notion of regular expressions first detailed by Kleene. But it is worth noting, that while complement and intersection do not enlarge the class of languages that regular expressions can represent, they can make the representations themselves rather more concise – it can be much easier to describe the one way something is *not* the case instead of the multitude of ways that it is. Being able to handle both intersection and complement is one of the principal advantages of Brzozowski’s method, which we consider shortly.

There is yet a third sense of “extended” applied to what goes by the name of regular expressions that can be found in many modern programming languages, such as Perl and Python. Some of the features offered by this third kind of extensions cannot, in fact, be captured by the definition of regular expression we are utilising herein. This is because some features such as backreferences actually mean that this kind of regular expression can denote languages which are not regular and in some cases not even context-free. The necessity of precluding such extensions will be explained in the next section when we discuss an equivalence between regular expressions and certain kinds of automata.

2.3 Deterministic finite automata

A deterministic finite automaton (DFA) is a finite-state machine which can be used to recognise a language. It does so by either accepting or rejecting any given string which is provided to it. A DFA is described by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states,
- Σ is a finite alphabet,
- q_0 is the initial state,
- $F \subseteq Q$ is the set of final, or accepting states, and
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function.

Intuitively, we can think of a DFA recognising whether a given string belongs to the language in terms of a process of stepping through the input character-by-character. Starting from the designated initial state, we follow the transition function for the pair of this state and the first character in the input giving us the next current state. We repeat this process for each character until the entire input has been consumed – in other words, until we encounter the empty string, ε . At this point, if the current state belongs to the set of accepting states then the string belongs to the language recognised by this DFA; otherwise, it does not.

A state diagram is one way of representing a DFA that can make this process more concrete for small examples. States are depicted by circles, with accepting states represented as a doubled circle. The transitions between these states are expressed by arrows labelled with the symbol. While the transition function of a DFA is a total function, we can simplify the diagram by considering any transitions not explicitly depicted as being implicit transitions to an error state which will consume any remaining input and ultimately reject. As an example, Figure 1 offers a state diagram for a DFA recognising the language that the regular expression aab^* recognises, i.e., any string consisting of two a s followed by any number of b s.

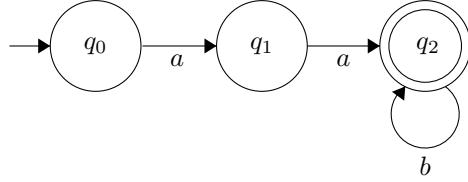


Figure 1: A state diagram for the DFA corresponding to aab^*

Tracing the transitions in this DFA we can see that strings such as aa and $aabb$ would be accepted, whereas ϵ , a and $aabba$ would be correctly rejected.

There is an equivalence between regular expressions and DFAs, in that we can construct a DFA capable of recognising precisely the set of strings constituting the language expressed by any regular expression. Thus, having a DFA which can recognise a language is another criterion for that language's regularity. As our present concern is a DFA construction algorithm, we constrain our focus to the kinds of properly regular expressions we describe above – i.e., those which capture only regular languages – as these are precisely the expressions for which we can construct equivalent DFAs.

3 Derivatives of regular expressions

We now turn to the concept of a derivative of a regular expression introduced by Brzozowski [2]. The derivative of a set of strings, L , with respect to a symbol a is the set of suffixes of strings in L which start with a . In formal terms, the derivative with respect to a , where $a \in \Sigma$ and $s \subseteq \Sigma^*$, is defined as

$$D_a L = \{s \mid as \in L\}$$

We can use this notion of derivatives with respect to a character to determine if a given string belongs to a language. We can match a string with a language via a series of derivatives, one for each character in the string. The language matches the string if ϵ is in the set resulting from these derivatives. We say that a set which contains the empty string is nullable, and we use the notion of nullability to define an auxiliary function \mathcal{N} .

$$\mathcal{N}(R) = \begin{cases} \epsilon, & \text{if } \text{nullable}(R) \\ \emptyset, & \text{otherwise} \end{cases}$$

As a regular expression is a method of describing a regular language, we can directly express the concept of a derivative of regular expression as follows:

$$\begin{aligned} D_a \emptyset &= \emptyset \\ D_a \epsilon &= \emptyset \\ D_a b &= \begin{cases} \epsilon, & \text{if } a = b \\ \emptyset, & \text{otherwise} \end{cases} \\ D_a(R^*) &= D_a R \cdot R^* \\ D_a(R \cdot S) &= D_a R \cdot S + \mathcal{N}(R) \cdot D_a S \\ D_a(R + S) &= D_a R + D_a S \\ D_a(R \& S) &= D_a R \& D_a S \\ D_a(\neg R) &= \neg D_a R \end{aligned}$$

We also note that regular languages are closed under derivation, meaning the derivative of a regular expression is another regular expression.

So far we have seen that regular expression derivatives can be used to implement regular expression matching. This, however, only partially suggests the usefulness of this concept. Brzozowski offers a natural and elegant, derivative-based algorithm for directly constructing a DFA recognising the language of a given regular expression. The algorithm proceeds as follows: the original regular expression corresponds to the initial state of our DFA, we iteratively take derivatives for each symbol in the alphabet, with the resulting regular expression being the state in the DFA we transition to on that symbol. The algorithm concludes when no new states are derived, and the final DFA is constructed. Pseudocode is given in Figure 2.

```

function BUILDDFA( $q_0, \Sigma$ )
     $TODO \leftarrow \{q_0\}$ 
     $Q \leftarrow \{q_0\}$ 
     $\delta \leftarrow \emptyset$ 
     $F \leftarrow \emptyset$ 
    if  $\mathcal{N}(q_0) = \varepsilon$  then
         $F \leftarrow \{q_0\}$ 
    while  $TODO \neq \emptyset$  do
        let  $q \in TODO$ 
         $TODO \leftarrow TODO \setminus \{q\}$ 
         $Q \leftarrow Q \cup \{q\}$ 
        for  $a \in \Sigma$  do
             $q' \leftarrow D_a q$ 
             $\delta \leftarrow \delta \cup \{(q, a), q'\}$ 
            if  $q' \notin Q \cup TODO$  then
                 $TODO \leftarrow TODO \cup \{q'\}$ 
            if  $\mathcal{N}(q') = \varepsilon$  then
                 $F \leftarrow F \cup \{q'\}$ 
    return  $(Q, \Sigma, \delta, q_0, F)$ 

```

Figure 2: Brzozowski’s DFA construction algorithm

3.1 Similarity of regular expressions

The set operations in our BUILDDFA function are presumed to work with a notion of regular expression equivalence. The “strength” of our notion of equivalence will have repercussions on the overall performance of our construction algorithm. Each unseen regular expression is added to the $TODO$ set, and adds a new state to our DFA. Therefore, not catching semantically equivalent, yet structurally distinct, regular expressions will result in a larger DFA. We shall say two regular expressions are equivalent when they denote the same set of strings. For example, it is easy to see that the regular expressions $a + a^*$ and a^* describe the same set of strings i.e., they are equivalent, yet they are not structurally identical. We want our algorithm to operate with a notion of equivalence which will minimise the number of unnecessary states added to our DFA, but we must consider the strictness of our equivalence as a parameter to our algorithm. This is because determining equivalence between arbitrary regular expressions is a non-trivial problem [11]. A standard way to check equivalence would be to construct and minimise a DFA for each expression and check for isomorphism. But that defeats the point, the whole aim is to construct a DFA in the first place. There can, however, be a compromise between time for full equivalence checking and size of our resulting DFA. To this end, Brzozowski offers three identities for equivalence over the alternation operation, $+$: idempotence, associativity, and commutativity.

$$\begin{aligned}
R + R &\approx R \\
R + S &\approx S + R \\
(R + S) + T &\approx R + (S + T)
\end{aligned}$$

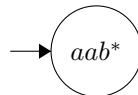
Brzozowski uses these identities to found a notion of similarity – if two regular expressions are similar they are equivalent, but two regular expressions may be equivalent without being similar. It is obviously much easier to check whether two regular expressions are similar in this weakened sense than whether they are strictly equivalent. We can rewrite expressions featuring $+$ to a canonical form, and then check for syntactic equality, without requiring an arbitrarily complicated method of checking regular expression equality. Brzozowski also proves that this notion of similarity is enough to ensure that the construction algorithm will always terminate – i.e., not spuriously generate an infinite number of equivalent, yet syntactically distinct, regular expression states – by showing that any given regular expression can produce only a finite number of different types of derivatives (modulo similarity) [2]. Owens, Reppy and Turon [12] augment Brzozowski's concept of similarity with a number of likewise easily checked identities:

$$\begin{array}{lll}
R \& R \approx R & \\
R \& S \approx S & \\
(R \& S) \& T \approx R \& (S \& T) & \\
\emptyset \& R \approx \emptyset & \emptyset + R \approx R \\
\neg\emptyset \& R \approx R & \neg\emptyset + R \approx \neg\emptyset \\
\\
(R \cdot S) \cdot T \approx R \cdot (S \cdot T) & (R^*)^* \approx R^* \\
\emptyset \cdot R \approx \emptyset & \varepsilon^* \approx \varepsilon \\
R \cdot \emptyset \approx \emptyset & \emptyset^* \approx \varepsilon \\
\varepsilon \cdot R \approx R & \neg(\neg R) \approx R \\
R \cdot \varepsilon \approx R &
\end{array}$$

Owens et al. state that operating with just Brzozowski's identities results in much larger automata than when the expanded concept of similarity is utilised. Furthermore, in their implementation this concept of regular expression similarity was enough to construct minimal automata in all but two test cases. Thus, we can see the level of sophistication of similarity checking we operate with as a parameter to our algorithm, affecting its overall performance. The impact that different notions of similarity can have on parallelism is a research question we address in Section 6.

3.2 An example

As a concrete example consider the process of the algorithm for the input regular expression aab^* and alphabet $\Sigma = \{a, b, c\}$. Initially, we add aab^* to $TODO$ and enter the while loop. Our initial state is set to our input regular expression.



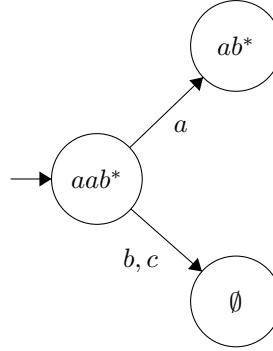
Looping over the alphabet we take derivatives with respect to aab^* for each symbol in our alphabet. For a , we get

$$\begin{aligned} D_a(aab^*) &= D_a a \cdot (a \cdot b^*) + \mathcal{N}(a) \cdot D_a a \cdot b^* \\ &= \varepsilon \cdot (a \cdot b^*) + \emptyset \cdot D_a a \cdot b^* \\ &= a \cdot b^* \end{aligned}$$

This adds a new state to our *TODO* set, and thus our state machine. Derivatives with respect to both b and c result in the regular expression \emptyset .

$$\begin{aligned} D_b(aab^*) &= D_b a \cdot (a \cdot b^*) + \mathcal{N}(a) \cdot D_b a \cdot b^* \\ &= \emptyset \cdot (a \cdot b^*) + \emptyset \cdot D_b a \cdot b^* \\ &= \emptyset \end{aligned}$$

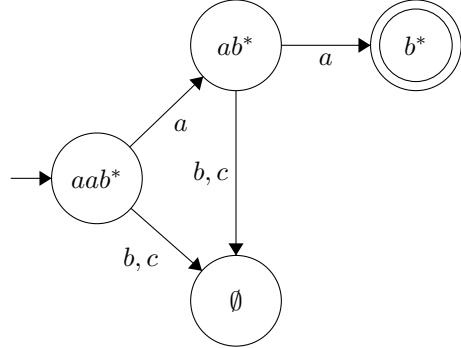
Neither of our newly derived states are nullable, thus the accepting set remains empty. At this point our state diagram is reflected as:



Our *TODO* set currently contains \emptyset and ab^* . The derivative of \emptyset results in \emptyset regardless of symbol, so all transitions from this state are reflexive. Derivatives of ab^* with respect to b and c both result in \emptyset . Taking the derivative with respect to a we obtain:

$$\begin{aligned} D_a a \cdot b^* &= D_a a \cdot b^* + \mathcal{N}(a) \cdot D_a b^* \\ &= \varepsilon \cdot b^* + \emptyset \cdot \emptyset \\ &= b^* \end{aligned}$$

b^* is a nullable regular expression, so it is marked as an accepting state. Our state machine reflecting this iteration now looks like:



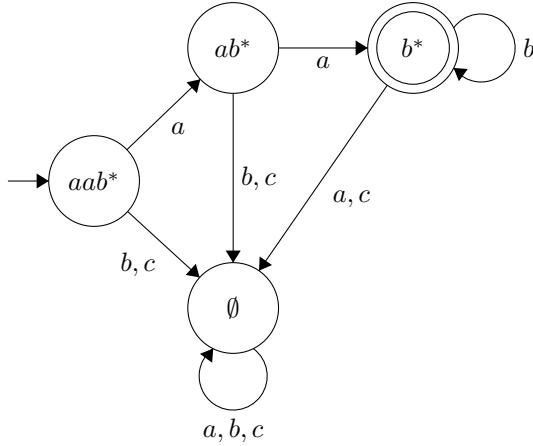


Figure 3: the result of $\text{BUILDDFA}(aab^*, \{a, b, c\})$

The only new state is b^* . Taking the derivative with respect to both a and c gives us \emptyset .

$$\begin{aligned} D_a b^* &= D_a b \cdot b^* \\ &= \emptyset \cdot b^* \\ &= \emptyset \end{aligned}$$

With respect to b we obtain:

$$\begin{aligned} D_b b^* &= D_b b \cdot b^* \\ &= \varepsilon \cdot b^* \\ &= b^* \end{aligned}$$

At this point, our *TODO* set is empty thus the algorithm returns our constructed DFA. Figure 3 depicts the final result. It is worthwhile to note that this process does construct a total transition function with an explicit error state – i.e., the state marked \emptyset . If we look at the states besides this error state (and considering any transitions to this state as implicit error transitions) we get a DFA that is equivalent to the minimal DFA shown above in Figure 1. With the explanation of the algorithm in place, we turn our attention towards parallelising it.

4 Towards parallelism

4.1 Parallelism versus concurrency

In many contexts descriptions of two or more events as happening “in parallel” or “concurrently” are considered largely synonymous. This is not necessarily the case when we are considering the sense of parallelism and concurrency used in the context of specifying and implementing algorithms. Before proceeding, it is important to briefly discuss the differences between the terms *parallel* and *concurrent* programming.

The ultimate aim of parallel programming is to achieve faster execution times for a computation. This is achieved by executing multiple actions *simultaneously*, for example by distributing workload to multiple cores or processors. In contrast, concurrent programming refers to a methodology of

structuring programs into modular processes that are possibly unrelated to one another. This structuring may be useful for achieving parallelism, but this is not its only aim. An operating system is an example of a system that should be specified in terms of concurrency. This is because it receives input from and performs output to multiple devices and a user would not want one process to appear totally non-responsive while waiting for another process to complete (e.g., we want to be able to move the mouse and read from disk in the same period of time). A concurrent specification of such a system can describe these processes independently from one another, but whether or not the processes are actually executed simultaneously depends on the underlying hardware. A multicore machine may indeed achieve simultaneous execution of different processes by distributing them to different cores, however the concurrent specification still allows for the system to function on a single processor by interleaving the execution of distinct processes so as to appear simultaneous. Thus concurrency can be used to achieve parallelism, but is a somewhat more general concept.

Bearing this distinction in mind, we consider the prior work of Strauss et al. [13, 14, 15] towards a concurrency-based specification and implementation of Brzozowski’s algorithm.

4.2 The concurrent, process-oriented specifications of Strauss et al.

Via a number of iterative refinements, the authors develop a series of concurrent specifications of Brzozowski’s DFA construction algorithm, in the style of Hoare’s Communicating Sequential Processes, or CSP [3, 4]. In establishing their concurrent specifications, they identify three potential sources of parallelism inherent in the sequential specification. These correspond to three steps performed within the inner for-loop.

- Checking nullability of the regular expression for inclusion in the set of accepting states
- Taking a derivative with respect to each symbol in the alphabet
- Updating the set of states in the DFA and the *TODO* set, based on regular expression similarity

These three steps can be executed independently of one another, and are thus promising sources of potential parallelism. Given the basis of CSP as the notation, the authors define these three steps as three independent sequential processes – named as variations of *OUTER*, *DERIVERS* and *UPDATER* in their specification – which communicate via synchronous channels. The *DERIVERS* process is itself comprised of parallel subprocesses, individually named *DERIV*_a for each $a \in \Sigma$.

In [14], the authors detail the results of implementing their concurrent specification in the language Go. This is a natural choice given the influence CSP has had on Go’s language design, resulting in language primitives of *goroutines* and *channels* corresponding, respectively, to processes and channels in Hoare’s CSP.¹

Their benchmarking procedure involved running their implementation against a test suite of randomly generated regular expressions of varying complexity for two different alphabet sizes. Performance was evaluated in terms of overall elapsed time for the sequential and parallel implementations, as well as speed-up and efficiency when run on parallel hardware. Speed-up, *S*, can be determined by running the same program on different numbers of processors, and is quantified as the elapsed time, *t_s*, when run on a single processor (i.e., sequentially) divided by the elapsed time, *t_p*, when run on *p* processors. Efficiency, *E*, refers to the ratio of speed-up to the number of processors, i.e., how effectively each of the processors is being used [5].

$$S = \frac{t_s}{t_p} \quad E = \frac{S}{p} = \frac{t_s}{p \cdot t_p}$$

¹CSP as originally described in [3] had processes communicating directly between one another i.e., process A addresses process B by name (this is the model of communication adopted by Erlang). However, [4] updates the language to include communication over channels à la Go, rather than directly between processes.

They ran two separate trials, one on a machine with two dual-core processors and one on a machine with a single four-core processor.² For the first of the trials, a median speed-up of 1.72 for the small alphabet and 1.09 for the large alphabet case is reported. In the second trial, they attained a median speed-up of 1.94 on the small alphabet, and 1.53 for the larger alphabet (12.8% and 40.4% increases respectively).

In their discussion of these results they suggest that the lower gains in speed-up with respect to the larger alphabet makes sense given the extra overhead involved: their specification creates a process for each symbol in the alphabet – and it is to be expected that there will be a consequent increase in scheduling overhead as the alphabet size is increased.

For Strauss et al. the choice of implementation language was motivated by the fact that Go provides language primitives for concurrent programming directly influenced by Hoare’s CSP. The need for concurrency was a major factor in the design of the Go language from the very beginning, and the designers of the language have taken cues from Hoare’s CSP in implementing this design. As such, Concurrency is the suggested way by which parallelism is to be achieved in Go: “*Do not communicate by sharing memory; instead, share memory by communicating*” is a motto of the language. While the authors have some motivations beyond simply implementing a more performant algorithm (e.g., exploring the effects of CSP as a design methodology on algorithm development and implementation), performance remains a key criterion and they present results in terms of speed-up and efficiency of their concurrently specified implementation. Thus they are exploring concurrency as a means to achieve parallelism. We, too, are interested in achieving parallelism. However, as we shall shortly see, the problem does not need to be expressed using the facilities of concurrency in Haskell.

5 Shifting to a functional context

In contrast to Strauss et al.’s exploration of the advantages offered by a concurrent specification of the problem, one of the main aims of the present research was to consider the resources that the functional paradigm offers in this situation.

The target language for our implementation is Haskell. Haskell is a strongly typed, lazily evaluated and purely functional language. Of these characteristics, it shares only strong types with Go. Thus it is reasonable to expect there may be significant differences in attempting to parallelise the algorithm. While Haskell does provide facilities for concurrent programming (and these can indeed be used to achieve parallelism), there is a stricter delineation between concurrency and parallelism. This can be considered in terms of the difference between deterministic, pure code and non-deterministic, impure (i.e., side-effecting) code.

- Concurrency is a method for structuring programs and relates naturally to situations wherein multiple threads of control are executing at the same time (for example responding to I/O from multiple devices). This necessarily introduces non-determinism, i.e., the program may produce a different result depending on the order in which these threads of control are interleaved and executed.
- Parallelism, on the other hand, means speeding up the execution of a (possibly pure and deterministic) program by running it on multiple processors.

As a pure language, Haskell functions produce no side-effects unless explicitly indicated (by the IO monad). A language with side-effects can use the structuring techniques of concurrency to achieve synchronisation, ensuring that a side-effect produced by the computation in one part of the program does not interfere with the correctness of a computation somewhere else. In this sense concurrency is used to ward-off unwanted non-determinism in the case where the result of a parallel computation should be deterministic – we don’t want different results depending on which parts of a parallel

²a MacPro with two Dual-Core Intel Xeon 2.66 GHz and 5 GB RAM and a MacBook Pro with a four-core Intel i7 2.3 GHz processor with 16 GB of RAM.

computation are initiated or executed in which order. However, if the language is free from side-effects then it cannot matter in which order components of an overall computation are evaluated. Parallel evaluation of pure code is guaranteed to be deterministic. Pure parallelism also means no race conditions and no deadlocks.

If the problem can be expressed in a pure way (i.e., no I/O) then Haskell offers a number of alternative methods for realising parallelism, besides concurrency.³ Evaluation Strategies and the dataflow parallelism of the `Par` monad are the two considered in the present research.

5.1 Evaluation strategies

Haskell provides two primitives for achieving deterministic parallelism in the `Control.Parallel` library⁴

```
par :: a -> b -> b
pseq :: a -> b -> b
```

Both these functions have the same type, and applications of `par` and `pseq` are both equivalent to the value of their second argument. The difference comes in what they say about the order of evaluation of these arguments. We can think of `par` as an annotation suggesting that its first argument *may* be evaluated in parallel. The use of this annotation to suggest an opportunity for parallelism is described as *sparking*. A ‘spark’ is a reference to a pure computation that may be evaluated in parallel, and they are stored in a work queue called the spark pool. It is up to the runtime system to manage the scheduling, execution or discarding (by garbage collection) of these opportunities for parallelism [9, 17]. `pseq`, on the other hand, expresses a sequence of evaluation. First, evaluate the first argument and then evaluate the second. The first argument is evaluated to what is termed *Weak Head Normal Form* (WHNF). An expression is in WHNF if it is either a lambda abstraction (e.g., `\x -> 1+2`), or if its outermost data constructor has been evaluated (e.g., `True`, `Just (1+2)`, `'a' : ("b" ++ "c")`). Subexpressions in a WHNF expression may or may not be evaluated. An expression is in *Normal Form* if it has been fully evaluated, meaning there are no unevaluated subexpressions. `True`, `Just 3` and `"abc"` are all in Normal Form.

Evaluation Strategies are an abstraction built upon these primitives for parallelism. A `Strategy` is a function expressing how subexpressions in its argument (for example, a data structure like a list or tree) might be evaluated in parallel.

```
type Strategy a = a -> Eval a
```

The function traverses the data structure given to it, and evaluates subexpressions either in parallel or sequentially – determined by the `par` and `pseq` annotations. Thus we can see that strategies depend on Haskell’s laziness: the substructure of the argument must contain some unevaluated components, otherwise there is no parallel computation to be done. Strategies are used to describe the *dynamic behaviour* of a parallel computation, i.e., how the runtime should organise the computation in the parallel context [16]. We can think of dynamic behaviour as comprised of two aspects: parallelism control (which unevaluated computations need to be sparked and in which order) and evaluation degree (the degree to which each sparked computation need be evaluated).

The intention is that the strategy function is an identity function, i.e., it returns the same value as was passed to it. This identity property holds for the strategies provided by the library, but it cannot be enforced for arbitrary, programmer-defined strategies. We can see from the type that a `Strategy` returns a value in wrapped in the `Eval` type constructor. `Eval` is a monadic context used to preserve evaluation order. The value can be extracted from this context using the `runEval` function

```
runEval :: Eval a -> a
```

³Indeed, the HaskellWiki offers the following rule of thumb: “use Pure Parallelism if you can, Concurrency otherwise.” See: <https://wiki.haskell.org/Parallelism>

⁴See: <http://hackage.haskell.org/package/parallel-3.2.2.0/docs/Control-Parallel.html>

However, it is typical to make use of the `using` function (often in its infix form ‘`using`’) which is defined in terms of `runEval`:

```
using :: a -> Strategy a -> a
x `using` s = runEval (s x)
```

To put this in somewhat more concrete terms, we can consider an application of strategies towards parallel evaluation of elements in a list. Given a function `f :: a -> b` and list `xs :: [a]` we obtain list of type `b` by applying function `f` to each element of `xs` with `map :: (a -> b) -> [a] -> [b]`

```
map f xs
```

Using functions provided by the `Control.Parallel.Strategies` library⁵

```
parList :: Strategy a -> Strategy [a]
rdeepseq :: NFData a => Strategy a
```

We can express that each element of the mapped list should be evaluated in parallel as follows:

```
map f xs `using` parList rdeepseq
```

`parList` is a higher-order strategy designed to apply a given strategy to each element of a list in parallel. `rdeepseq` is a strategy saying its argument should be fully evaluated to Normal Form.⁶ Thus, in terms of the dynamic behaviour of the overall computation, we can see `parList` corresponds to parallelism control and `rdeepseq` corresponds to evaluation degree.

This example demonstrates the *composability* and *modularity* of strategies. We can create new strategies with other strategies as the building blocks, i.e., we can *compose* them. And they allow for a *modular* design by decoupling the algorithm from the techniques used to parallelise it: we could substitute `rdeepseq` for another strategy with very minimal alterations to the code. And, as mentioned, the code with and without ‘`using`’ `parList rdeepseq` is semantically equivalent, by the identity property. Adding parallelism does not introduce non-determinism.

Where the strength of the strategies approach lies in modularity and composability, they can be weak in terms of fine-grained control. As has been suggested, strategies rely upon laziness in order to operate effectively – given the somewhat implicit, declarative style of using strategies it can be difficult to reason about evaluation order and degree (i.e., introducing too much strictness, or not enough). Also it can be hard to operate with the right level of granularity. To get reasonable performance increases due to parallelism, the work to be done needs to be divided up enough to keep the parallel hardware busy. But dividing the work at too fine a granularity introduces excessive overhead in terms of scheduling parcels of work and recombining them into the overall scheme of the computation. The `Control.Parallel.Strategies` library does offer some techniques useful for addressing the issue of granularity, for example, `parListChunk :: Int -> Strategy a -> Strategy [a]` which operates similarly to `parList`, but first divides the given list into n chunks, with the intention that the size of these chunks will constitute an appropriate amount of work.

5.2 Dataflow parallelism and the Par monad

The `Par` monad was originally posed as an alternative programming model to strategies, designed to address some of the potential shortcomings of that approach [10]. Compared to Strategies, the `Par` monad offers a more explicit programming model with direct granularity control. This model is influenced by Haskell’s programming model for concurrency, however, the `Par` monad retains the deterministic quality desirable for pure parallelism. In abstract, the `Control.Monad.Par` library⁷ offers the following basic primitives for specifying parallel computations:

⁵See: <http://hackage.haskell.org/package/parallel-3.2.2.0/docs/Control-Parallel-Strategies.html>

⁶the type class constraint `NFData` placed on `rdeepseq`’s argument says that there must be a function `rnf` defined for its argument’s type which specifies how this evaluation to Normal Form should be undertaken.

⁷See: <http://hackage.haskell.org/package/monad-par-0.3.4.8/docs/Control-Monad-Par.html>

```

data IVar a

new :: Par (IVar a)
put :: NFData a => IVar a -> a -> Par ()
get :: IVar a -> Par a

```

We can think of an `IVar` as a write-once variable. `new` calls an `IVar` into existence. `put` assigns a value to an `IVar` and can be called only once, as each `IVar` is write-once (subsequent puts result in an error).⁸ Finally, `get` reads the value assigned to an `IVar`, waiting for it to be evaluated, if necessary.

Parallelism in the `Par` monad is expressed as a *forking* of computations, expressed by the function

```
fork :: Par () -> Par ()
```

The argument passed to `fork` is the computation to be run in parallel with the current computation. `IVars` and the operations defined for them constitute the infrastructure by which a computation run in parallel can communicate its results to different computations. Finally, `runPar :: Par a -> a` is the means by which we run a parallel computation specified within the `Par` monad, and return its (deterministic) result.

The idea behind the `Par` monad is that the programmer can specify the program in terms of a *dataflow* model. We can see the use of the above primitives to specify such a dataflow model of the small example below:

```

runPar $ do
    i <- new
    j <- new
    fork (put i (f x))
    fork (put j (g x))
    a <- get i
    b <- get j
    return (a,b)

```

We create two `IVars`, `i` and `j`. We `fork` the computation of `f x` and `g x` to be evaluated in parallel in two separate threads, assigning the values of these computations to `i` and `j`, respectively. The calls to `get` will wait for the values to be assigned, then give them the names `a` and `b` (this way the values have been communicated from the forked threads back to the parent). Finally, we return the pair `(a,b)`, and this is extracted from the `Par` monad by the call to `runPar`.

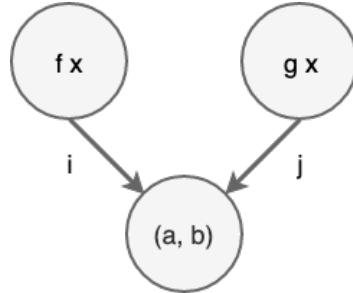


Figure 4: Dataflow model

Figure 4 provides a diagram representing this code sample, and demonstrates that our dataflow model is a directed graph. Nodes represent a call to `fork` creating a new computational context, and edges represent communication via `IVars`. But, unlike the style of parallelism annotations we saw with strategies, the programmer does not need to say that they can be run in parallel – potentials

⁸`put` is defined to be strict (which can be seen in the `NFData` constraint).

for parallelism can be determined from the programmer-specified dataflow model. For example, we can tell by the structure of the graph that because there are no dependencies between the node `f x` and node `g x`, they can be run in parallel. The graph can be automatically scheduled onto available processors by the scheduler implemented within the `Control.Monad.Par` library and we can make use of the parallelism inherent in the graph.⁹

It is important to note that dataflow models can be dynamically constructed and are not constrained to the type of static network illustrated in the small example above. Dynamic networks can be achieved by building abstractions upon the primitives, or by utilising those provided by the library, such as `parMap :: NFData b => (a -> b) -> [a] -> Par [b]`.

We obtain a more explicit programming model than strategies, offering direct control over granularity and dependencies. It has a somewhat more imperative flavour, but still remains deterministic. The fact that `IVars` are strictly write-once ensures determinism, as there can be only one value at the end of the graph. Namely, the relative ordering of `puts` and `gets` cannot affect the overall computation.

6 Performance evaluation

6.1 Experimental setup

In order to explore the possibilities for deterministic parallelism offered by Haskell in the context of Brzozowski’s DFA construction algorithm, we present a series of evaluations and compare them with Strauss et al.’s findings regarding concurrency-based parallelism. The basic approach to testing follows that employed by Strauss et al. in the hope of allowing for meaningful comparisons to be drawn between the results obtained. We randomly generated two test sets of regular expressions with a small ($|\Sigma| = 4$) and larger ($|\Sigma| = 94$) alphabet, each covering a range of different depth syntax trees. The expressions were generated via a recursive method analogous to that outlined in [14]. Given an input target depth, $n > 0$, and an alphabet, Σ , randomly select an operator from the set $\{\cdot, +, \&, \neg\}$ and recursively generate the argument(s) of the operator for $n - 1$. For $n = 0$, select a random symbol $a \in \Sigma$. We considered depths in the range 4 to 10, inclusive, and at each depth we generated 200 regular expressions.

We compare the performance of different implementations of Brzozowski’s DFA construction algorithm. The implementations evaluated consisted of a sequential algorithm as a baseline, and three versions based on deterministic parallelism. Two variations of an approach using strategies provided by the `Control.Parallel.Strategies` library were implemented, one with and one without chunking applied. In order to obtain a suitable chunk size parameter, n , the implementation was run (utilising all available cores) on both the test suites with n set to each of the values in $1..|\Sigma|$. The upper bound of $|\Sigma|$ is determined by the fact that the list to be parallelised cannot be chunked any smaller than individual elements). The value of n which produced the minimal runtime was then selected as the chunk size parameter for the comparison tests. Figure 5 shows the results of these chunking evaluations. The values for n obtained were 2 and 4 for the small and large alphabets, respectively. The final parallel approach was based on the dataflow parallelism of the `Control.Parallel.Monad` library.

Evaluation of performance involved recording the elapsed time for each of these implementations on the regular expression test sets, using the `Criterion` Haskell microbenchmarking library.¹⁰ Runtime measurements were taken against the entire test set and for each individual depth. From the elapsed time measurements it is possible to obtain values for speed-up and efficiency compared to number of cores utilised. These latter two measurements are considered particularly useful as metrics of overall performance. While it is perhaps not appropriate to directly compare elapsed time

⁹It is worthwhile mentioning that the scheduler is implemented as a Haskell library, rather than part of the runtime system as with Strategies. This allows the programmer to substitute different schedulers depending on the task at hand, or implementing their own.

¹⁰See: hackage.haskell.org/package/criterion

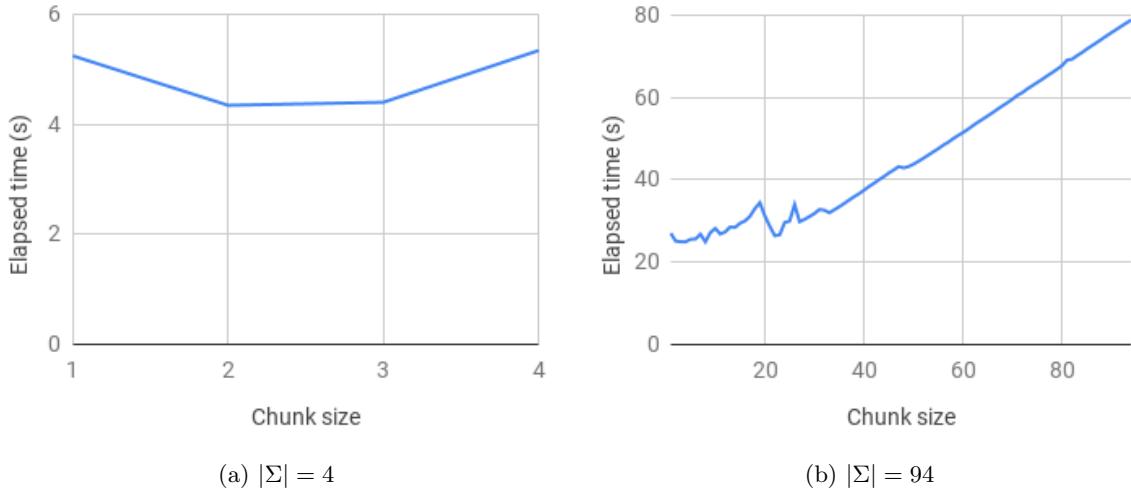


Figure 5: Peformance relative to chunking size parameter

between the results obtained and those presented by Strauss et al. due to the differences in testing environment, the relative metrics of speed-up and efficiency should still be able to give some insight into relative strengths and weaknesses between the different implementation methodologies.

The testing environment was a 2015 iMac with a 3.1GHz 4 core Intel i5 processor and 8GB of RAM. All code was compiled using GHC version 8.6.5, with the `-O2 -threaded -rtsopts -eventlog` compiler flags supplied.

6.2 Observations

Figures 6 - 9 detail the speed-up and efficiency compared to number of cores for each of the implementations when run against the entire test suite. We include these charts to give an overall view of the potential for increased performance realised by deterministic parallelism.

When considering the entire test sets, it can be seen that in every case introducing parallelism resulted in a decreased runtime, confirming the suitability of the parallel implementations for achieving increased performance. The overall largest increase was obtained by the `Par` monad implementation utilising four cores to yield a speed-up of 2.95. This is a fairly significant improvement on the best speed-up reported by Strauss et al. of 2.2 on four cores [14]. The most efficient use of parallelism was achieved by the same implementation utilising 2 cores, getting efficiency of 90% – meaning both cores were doing useful work for that proportion of time. Both of these measurements were obtained against the larger alphabet test set.

However, if we consider performance at each depth we can see that the bulk of the speed-up occurs on the largest, more complex, regular expressions. Furthermore, we introduce a slow-down over the smaller cases by introducing parallelism. There is an overhead associated with the parallel implementations, which can only be paid off if there is enough work to be parallelised.

The slow-down in the less complex test cases raises the question of overhead. If we consider the efficiency of parallel implementations when running on a single core (i.e., running sequentially) we can get an indication of the overhead these techniques introduce. Tables 1 and 2 show the efficiency of our implementation when using a single core, specified for each depth in the two test sets. For both the small and larger alphabet the `Par` monad is seen to carry significantly higher overhead in the single core case at smaller depths, but as depth increases the gap in efficiency between the `Par` monad and strategies implementations decreases. Indeed, in the large alphabet case at depth 8 and 9 it surpasses both strategies implementations.

These tables also show that the plain strategies approach carries very little overhead by itself.

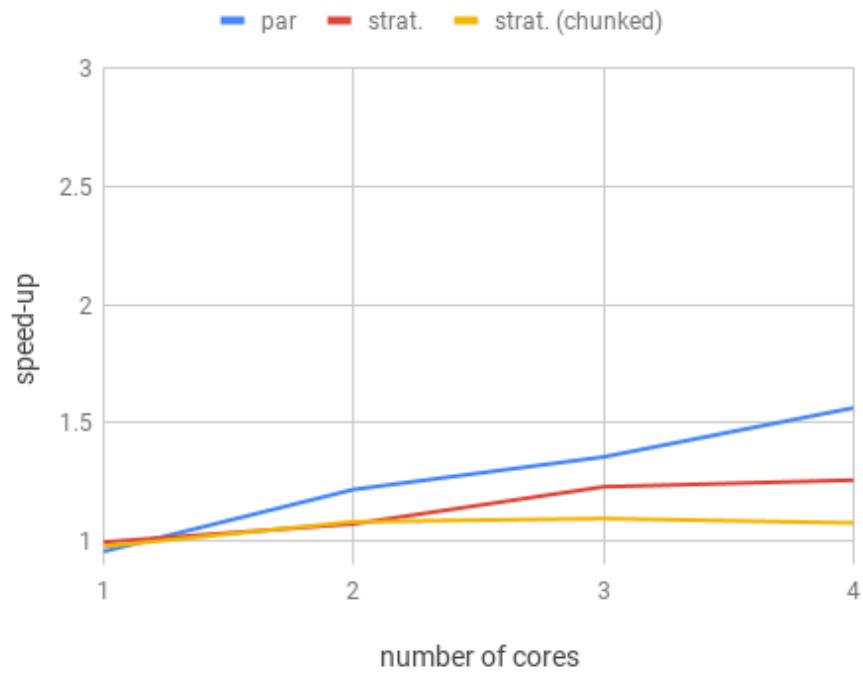


Figure 6: Speed-up compared to number of cores, $|\Sigma| = 4$

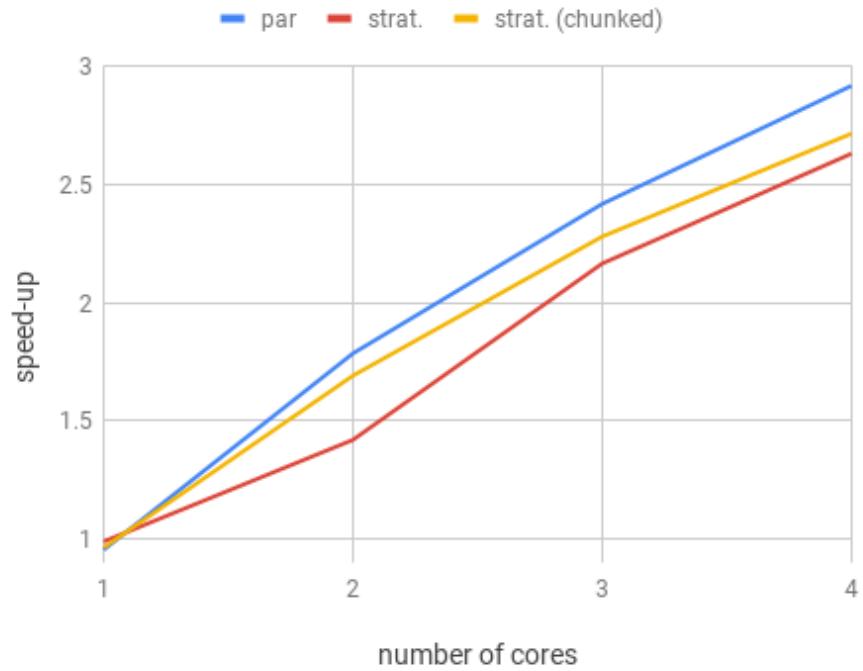


Figure 7: Speed-up compared to number of cores, $|\Sigma| = 94$

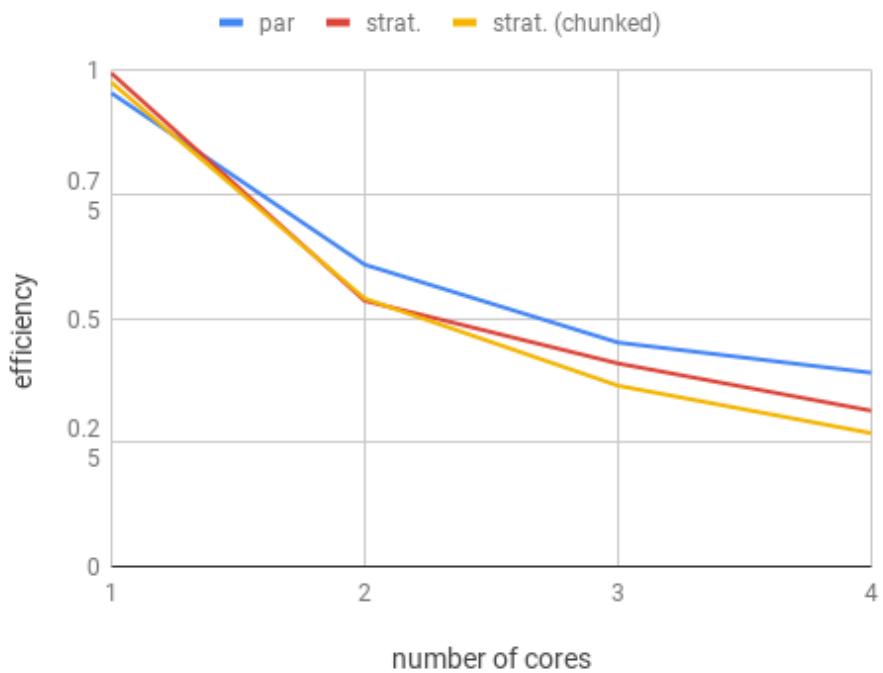


Figure 8: Efficiency compared to number of cores, $|\Sigma| = 4$

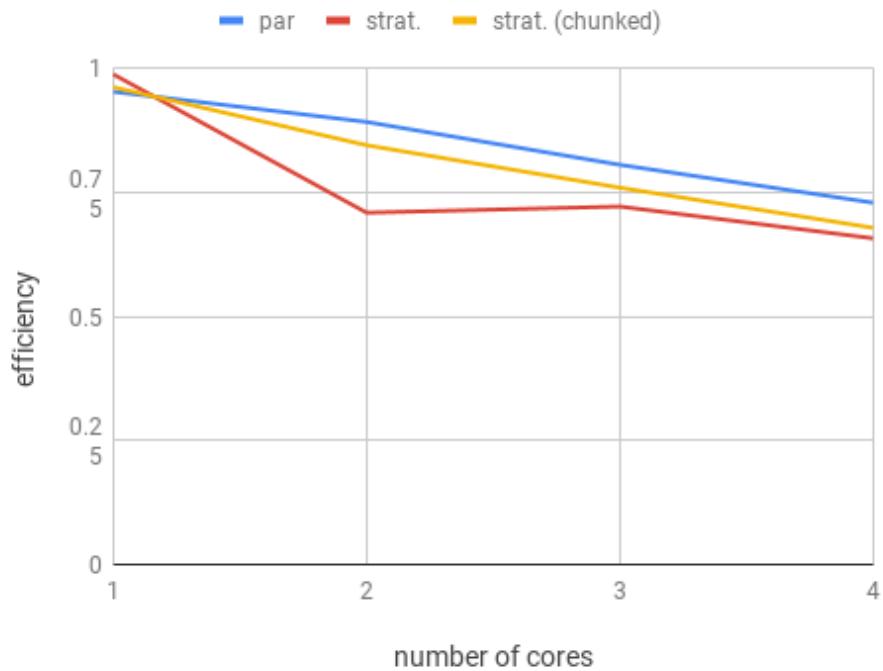


Figure 9: Efficiency compared to number of cores, $|\Sigma| = 94$

Chunking was similar, however the act of dividing up the preprocessed list and reassembling the computed result and obviously comes with a certain degree of overhead. This suggests why the implementation using chunking had significantly lower speed-up over the small alphabet test cases – chunking yielded improved performance only when the alphabet size was large enough to offset the overhead required by gains from the parallelism.

Depth	Sequential runtime (s)	Efficiency		
		Par monad	Strategies	Chunked
4	0.0316	76.14%	100.96%	99.06%
5	0.0581	82.06%	97.65%	97.98%
6	0.1028	86.68%	98.28%	96.89%
7	0.2659	88.52%	98.96%	97.76%
8	0.5991	96.33%	99.01%	97.97%
9	0.1427	96.39%	100.59%	98.80%
10	4.1063	94.58%	97.26%	99.81%

Table 1: Efficiency of the various implementations when run on a single core, $|\Sigma| = 4$

Depth	Sequential runtime (s)	Efficiency		
		Par monad	Strategies	Chunked
4	0.0559	59.72%	96.55%	92.24%
5	0.1358	78.57%	96.93%	91.39%
6	0.3665	87.19%	94.38%	95.32%
7	1.0618	92.72%	93.34%	88.74%
8	3.688	95.87%	93.20%	95.64%
9	10.629	98.50%	96.64%	96.92%
10	56.054	96.93%	99.90%	94.77%

Table 2: Efficiency of the various implementations when run on a single core, $|\Sigma| = 94$

Another observation to be drawn from the results is that generally the boost in speed-up to be gained and overall efficiency diminished with an increased number of cores. We demonstrate this increase in overhead with more cores by plotting at the performance of the **Par** monad implementation at each depth when utilising 1-4 cores in Figure 10. We focus on this implementation as it presents a more extreme case of a phenomenon present to a lesser degree in the other implementations. The charts are plotted on a logarithmic scale, so a downwards column represents a slow-down. For the large alphabet, the overhead was not counteracted by increased performance until depth 6, and even here the implementation still performed better when utilising only 3 cores. Introducing more processors into the mix will necessarily bring with it a certain amount of overhead, due to the fact that work must be scheduled to the processors and computations evaluated in parallel must be at some point recombined into one sequential context. This suggests that the right degree of parallelism to employ for any given problem should be determined on a case-by-case basis, informed by taking measurements on typical inputs. A similar story is told by the small alphabet cases. We can clearly see that until the break-even point of introducing parallelism is reached, more cores means decreased performance. And this break-even point is clearly related to regular expression complexity.

Despite the additional overhead, using the **Par** monad to express the parallel computation outperformed the evaluation strategies approach for both the small and larger alphabet. A slight increase in source code complexity yields a slight boost in performance. Even though the overall speed-up on the smaller alphabet cases was far from optimal, it did clearly out perform both strategies approaches on this test set.

Looking at the charts comparing both alphabet cases it is obvious that more significant perfor-

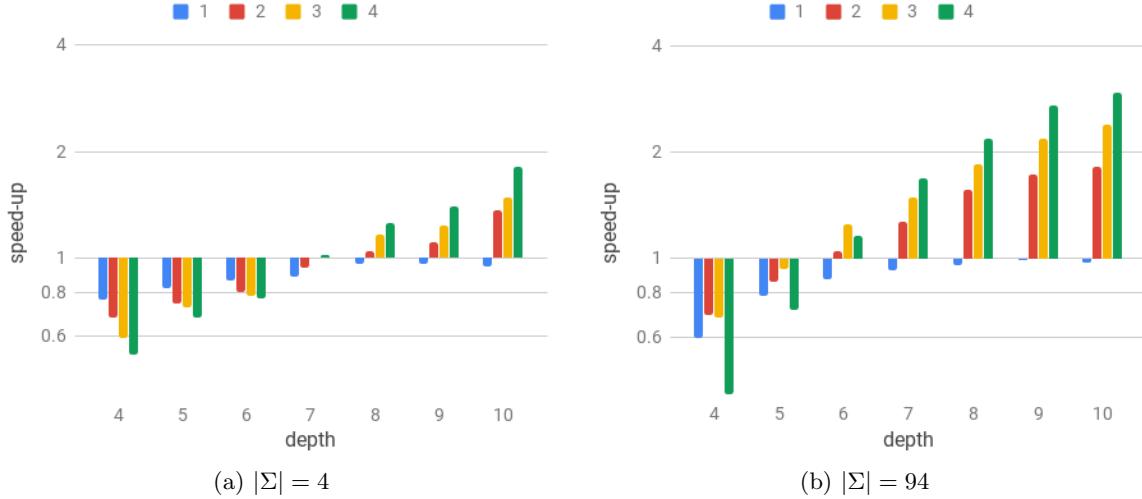


Figure 10: Peformance of the `Par` monad implementation at each depth. A column below 1.0 demonstrates an overall slow-down.

mance gains were achieved when the alphabet was larger. This makes intuitive sense, given that the tactic was to attempt parallelising the operations performed with respect to each symbol in the alphabet. Interestingly, this reverses the findings reported by Strauss et al. [14]. In the first of their two trials they achieved a median speedup of 1.72 for the small alphabet, yet only 1.09 for the large alphabet case. In the second experiment (utilising a four-core machine), the difference between the small and large alphabet cases was not as drastic, however the implementation still obtained better speed-up over the smaller alphabet (1.94 to 1.53 respectively). Thus, it appears that whereas the Go implementation handles the fine-grained parallelism of the smaller alphabet better, the deterministic parallelism of the Haskell implementations scale up better to larger alphabets.

Why this discrepancy between our results occurs is not immediately apparent, as their implementation is not publicly available. As noted above, Strauss et al. account for the decrease in effectiveness in terms of the increased overhead, related to scheduling and communication between processes that the larger alphabet necessitates. We have already noted that the `Par` monad performed considerably better on the small alphabet tests than either of the Strategy based approaches. One potential explanation is the fact that the `Par` monad programming model is – while explicitly deterministic – much closer to a concurrency based model of parallelism, such as that present in Go. The explicit granularity control that comes with the `Par` monad may be more suited to the reduced parallelism prospects that exist with the smaller alphabet.

Another, somewhat difficult to verify, possibility is that the discrepancy is an artefact of our respective test sets. Although the regular expressions were generated via an analogous process, the inherent randomness raises the probability for significant differences between the sets. During implementation and testing, it was observed that particular regular expressions produce significantly larger DFAs than others, even for the same depth regular expression. For example, DFAs for depth 10 regular expressions ranged from 2 to a maximum DFA size of 207. In contrast, Strauss et al. report a maximum DFA size for their larger alphabet of around 60, whereas the smaller alphabet tended to produce larger DFAs (the reported maximum was larger than 500 states). This makes sense given random generation and inclusion of the intersection operation: having more alphabet symbols to choose from makes it more likely to produce an expression of the form $a \& b$, where $a \neq b$. This obviously empty language would reduce the complexity of the overall DFA.

The size of the resulting DFA gives at least a rough insight into the runtime of the construction algorithm, in that each new state introduced into the DFA requires a derivative to be taken with

respect to each symbol in the alphabet. Thus, the different results may be explained by randomness between the two test sets. We discuss this limitation in Section 7.1 as an issue for further research.

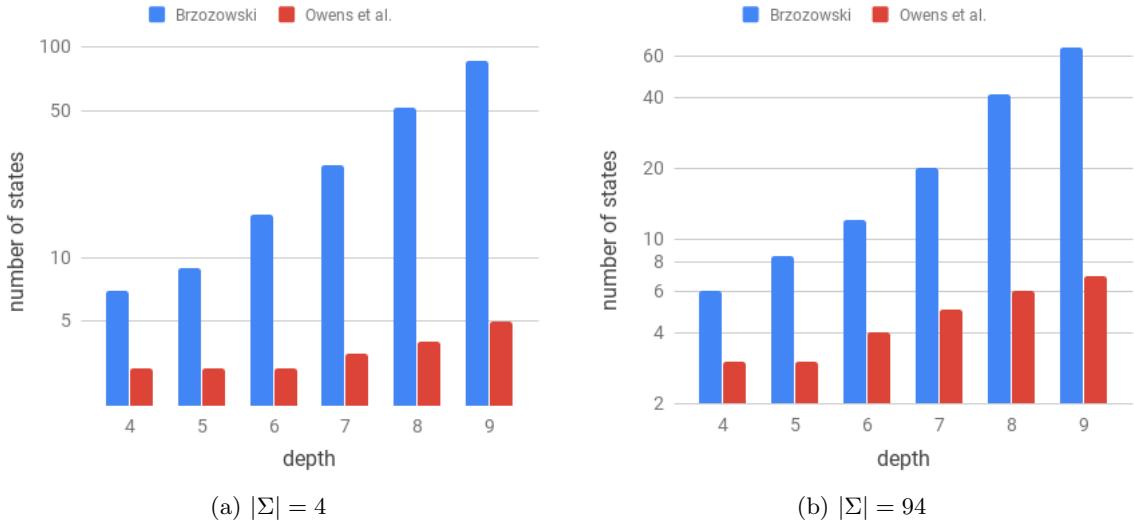


Figure 11: Median number of states produced utilising Brzozowski’s similarity notion and the extensions proposed by Owens, Reppy and Turon

We consider one final possibility which may stem from a different implementation of regular expression similarity. The notion of similarity they employ is not explicitly specified in their paper. The one that Brzozowski gives is related only to the associativity, commutativity and idempotence of the $+$ operator. To test the intuition, the simplification rules used in the various implementations were modified to perform according to Brzozowski’s original specification and benchmarks rerun to look at the impact on the different implementations. Figure 11 shows the median DFA size in terms of number of states obtained by both Brzozowski’s and Owens, Reppy and Turon’s notions of regular expression similarity. We can only show data for up to depth 9, because while theoretically guaranteed to terminate, the massive sizes of some of the constructed DFAs made the problem practically intractable for the largest test cases. As an example, while the maximum number of states at depth 9 for the Owens, Reppy and Turon rules was 102 the same regular expression for Brzozowski’s rules produced a massive DFA containing a 54,243 states. Note that the vertical scale is logarithmic, so we get an exponential increase in states from an increase in depth. This is to be expected given that the test expressions are generated by a method which can double the size at each recursive step. We see that utilising only Brzozowski’s rules produces significantly bigger DFAs, and the rate of growth relative to depth is also larger.

Considering the effects the notions of similarity have on parallelism, we show the speed-ups obtained on 4 cores by the `Par` monad utilising both sets of similarity rules, in Figure 12. Potential speed-up on the larger alphabet cases was reduced significantly, as we would expect if it was the case that Strauss et al. were using this restricted set of similarity rules. However, there was no significant boost in potential parallelism in the small alphabet case with both sets of rules attaining very similar levels of speed-up. It must be noted that these are relative speed-ups. The total time elapsed for the implementation using just Brzozowski’s rules took 298.46 seconds for the depth 9 expressions, compared to just 0.818 seconds for the extended set of rules. This massive difference in overall elapsed time precludes Brzozowski’s restricted rules from being useful in any practical implementation. The question of exactly what notion of similarity Strauss et al. used in their implementation remains open. However, we have shown that reducing the strength of the similarity checker does not, by itself, explain the difference in our respective results regarding performance with respect to alphabet size.

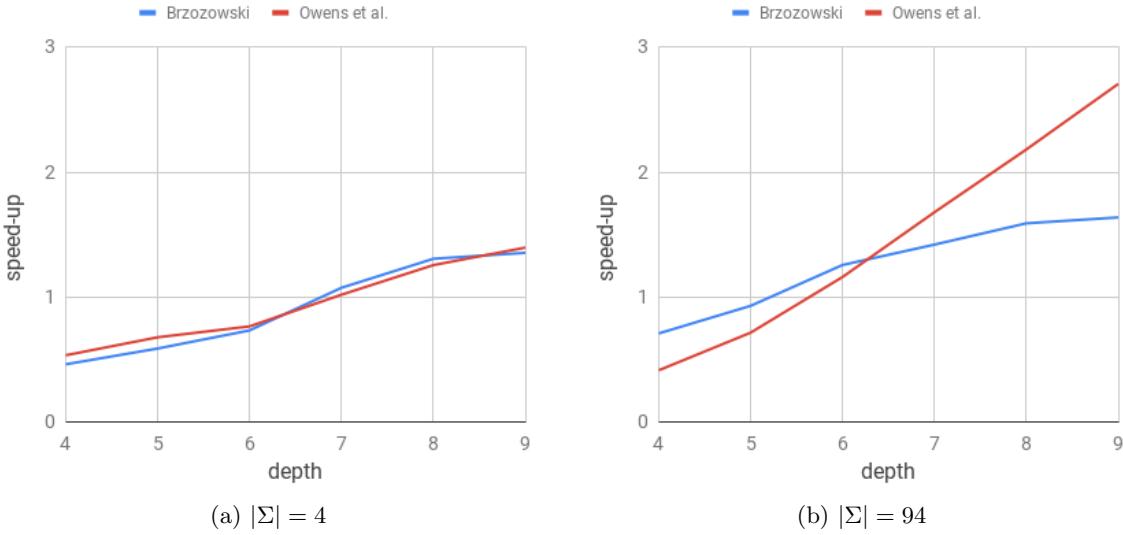


Figure 12: Speed-ups observed with the `Par` monad implementation using different notions of regular expression similarity

7 Conclusions

We have demonstrated that Haskell offers resources for parallelism capable of improving performance of Brzozowski’s DFA construction algorithm. The focus was on how a deterministic parallel specification compares to a concurrent approach to the same problem. Our findings suggest that both approaches are capable of attaining increased performance, but the relative strengths may suit different scenarios and use cases.

To further explore these differences we examined how differences in algorithm implementation (regular expression similarity) and input characteristics (alphabet size and expression complexity) affect performance. These investigations suggest that the Haskell implementation is most suited to situations with a larger alphabet and more complex expressions. Indeed, in the intersection of these conditions we were able to produce a maximum speedup of 2.95, improving on the best speed-up of 2.2 reported in a previous study.

The techniques that were used in our parallel implementations were not overly specific to Brzozowski’s algorithm. Instead, we were mostly able to use functions provided by the various parallelism-oriented libraries designed to work with standard data structures, such as lists. This lends our findings a degree of generality, suggesting we may hope to see similar improvements across a variety of domains.

7.1 Future work

The present study has suggested the suitability of deterministic parallelism as a means to achieve improved performance in an implementation of Brzozowski’s DFA construction algorithm. Having established a comparison with Strauss et al.’s findings [14] further work would be oriented towards taking this implementation in the direction of real-world practicality.

The test cases relied on randomly generated regular expressions. The decision to follow this path was to align the testing methodology with that employed by Strauss et al. The regular expressions generated by such a random process are obviously somewhat dissimilar to those employed in ‘real-world’ use cases. The applicability of the results attained in the present study to different types – i.e., not randomly generated – remains an open question.

The regular expression definition used and implemented for this research remains more closely aligned to the theoretical or formal notion than the kinds of regular expressions one may find implemented in various programming languages. Various standards for regular expressions such as POSIX and PCRE define additional operators such as ? or the dot ‘.’ wildcard operator. While these operators can be captured in a more verbose fashion by the set of operators we have defined, the effect the more concise representation would have on performance – particularly on parallel performance – is not answered by the present research. Extending the implementation to incorporate such syntactic extensions and benchmarking the results is left as a task for a future enquiry.

Another aspect of the present research that may not reflect real world applications is alphabet size. The sizes considered herein were chosen in order to provide a meaningful comparison with those selected in [14]. However, the largest alphabet size of 94 (a printable subset of ASCII) is obviously very different from the upward of 1.1 million Unicode code points. While the implementation is built upon the Haskell `Char` datatype which represents Unicode characters and would correctly handle such an alphabet, the time and space requirements render the problem intractable for the current specification. Owens, Reppy and Turon [12] consider an extension to Brzozowski’s original algorithm and the resulting DFA. Under this modification, transitions between states are expressed for classes of characters, rather than a single transition per symbol. This allows their implementation to work at Unicode scale in a sequential context. The effect that such a modification would have on prospects for parallelism is particularly interesting given the close relationship identified between alphabet size and the increases in throughput that could be attained. This too is identified as a potentially fruitful avenue for further study.

References

- [1] ANTIMIROV, V. M., AND MOSES, P. D. Rewriting extended regular expressions. *Theor. Comput. Sci.* 143, 1 (1995), 51–72.
- [2] BRZOZOWSKI, J. A. Derivatives of regular expressions. *J. ACM* 11, 4 (1964), 481–494.
- [3] HOARE, C. A. R. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677.
- [4] HOARE, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [5] KARP, A. H., AND FLATT, H. P. Measuring parallel processor performance. *Commun. ACM* 33, 5 (1990), 539–543.
- [6] KLEENE, S. C. Representation of events in nerve nets and finite automata. In *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton University Press, Princeton, NJ, 1956, pp. 3–41.
- [7] KONSTANTINIDIS, S., MOREIRA, N., REIS, R., AND SHALLIT, J., Eds. *The Role of Theory in Computer Science - Essays Dedicated to Janusz Brzozowski* (2017), World Scientific.
- [8] LEISSL, E. L. Is complementation evil? In Konstantinidis et al. [7], pp. 121–134.
- [9] MARLOW, S., MAIER, P., LOIDL, H., ASWAD, M., AND TRINDER, P. W. Seq no more: better strategies for parallel Haskell. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010* (2010), J. Gibbons, Ed., ACM, pp. 91–102.
- [10] MARLOW, S., NEWTON, R., AND PEYTON JONES, S. L. A monad for deterministic parallelism. In *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011* (2011), K. Claessen, Ed., ACM, pp. 71–82.
- [11] MEYER, A. R., AND STOCKMEYER, L. J. The equivalence problem for regular expressions with squaring requires exponential space. In *13th Annual Symposium on Switching and Automata*

Theory, College Park, Maryland, USA, October 25-27, 1972 (1972), IEEE Computer Society, pp. 125–129.

- [12] OWENS, S., REPPY, J. H., AND TURON, A. Regular-expression derivatives re-examined. *J. Funct. Program.* 19, 2 (2009), 173–190.
- [13] STRAUSS, T., KOURIE, D. G., AND WATSON, B. W. A concurrent specification of Brzozowski’s DFA construction algorithm. In *Proceedings of the Prague Stringology Conference, Prague, Czech Republic, August 28-30, 2006* (2006), J. Holub and J. Zdárek, Eds., Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, pp. 90–99.
- [14] STRAUSS, T., KOURIE, D. G., WATSON, B. W., AND CLEOPHAS, L. G. A process-oriented implementation of Brzozowski’s DFA construction algorithm. In *Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014* (2014), J. Holub and J. Zdárek, Eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, pp. 17–29.
- [15] STRAUSS, T., WATSON, B. W., KOURIE, D. G., AND CLEOPHAS, L. CSP for parallelising Brzozowski’s DFA construction algorithm. In Konstantinidis et al. [7], pp. 217–244.
- [16] TRINDER, P. W., HAMMOND, K., LOIDL, H., AND PEYTON JONES, S. L. Algorithms + strategy = parallelism. *J. Funct. Program.* 8, 1 (1998), 23–60.
- [17] TRINDER, P. W., LOIDL, H., AND POINTON, R. F. Parallel and distributed Haskells. *J. Funct. Program.* 12, 4&5 (2002), 469–510.