A Process-Oriented Implementation of Brzozowski's DFA Construction Algorithm

Tinus Strauss[1], Derrick G. Kourie[2], Bruce W. Watson[2], and Loek Cleophas[1]

[1] FASTAR Research group
University of Pretoria
South Africa
{tinus,loek}@fastar.org
[2] FASTAR Research group
University of Stellenbosch
South Africa
{derrick,bruce}@fastar.org

Abstract. A process-algebraic description of Brzozowski's deterministic finite automaton construction algorithm, slightly adapted from a previous version, shows how the algorithm can be structured as a set of communicating processes. This description was used to guide a process-oriented implementation of the algorithm.
The performance of the process-oriented algorithm is then compared against the sequential version for a statistically significant number of randomly generated regular expressions. It is shown that the concurrent version of the algorithm outperforms the sequential version both on a multi-processor machine as well as on a single-processor multi-core machine. This is despite the fact that processor allocation and process scheduling cannot be user-optimised but are, instead, determined by the operating system.

Keywords: automaton construction, concurrency, CSP, regular expressions

1 Introduction

Although contemporary computers commonly have multicores, the processor allocation and scheduling is not in the hands of the application software developer but, instead, determined by the operating system. This fact raises numerous questions. What are the implications of parallelising algorithms that have traditionally been expressed sequentially? The strengths and weaknesses of the sequential algorithms are generally well-known, and often a lot of effort has gone into sequential optimisations. Furthermore, for most software developers parallel thinking is unfamiliar, difficult and error-prone compared to sequential algorithmic thinking. Is parallel thinking inherently difficult for software developers or is the relative scarcity of parallel versions of sequential software simply a matter of inertia? Is it worth the effort to convert traditional sequential algorithms into parallel format when the fate of the software—the processor allocation and process scheduling—is largely out of the developer's control? Perhaps questions such as these explain, at least in part, why there has not been a mushrooming of parallel software algorithm development, notwithstanding more than a decade of hype about the future of computing being in parallelism. These observations apply not only to algorithmic software development in general, but also to the specific case of algorithmic software related to stringology.

This paper makes a start in assessing the practical implications of developing and implementing parallel algorithmic versions of well-known stringological sequential algorithms in contexts where we have no direct control over processor allocation and

Tinus Strauss, Derrick G. Kourie, Bruce W. Watson, Loek Cleophas: A Process-Oriented Implementation of Brzozowski's DFA Construction Algorithm, pp. 17–29.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

scheduling. A process algebraic description of a Brzozowski's deterministic finite automaton construction algorithm, slightly adapted from a previous version [15], shows how the algorithm can be structured as a set of communicating processes in Hoare's CSP [8,7]. This description was used to guide a process-oriented implementation of the algorithm in Go [17], as Go's concurrency features (inspired by CSP) allowed us to easily map from CSP to Go. A scheme is described to randomly generate regular expressions within certain constraints. The performance of the process-oriented algorithm is then compared against the sequential version for a statistically significant number of randomly generated regular expressions. It is shown that the concurrent version of the algorithm outperforms the sequential version on a multi-processor machine, despite the fact that processor allocation and process scheduling cannot be user-optimised but is, instead, determined by the operating system.

Of course, [15] is one of several efforts at developing parallel algorithms for stringological problems. Some previous efforts include [4,18] for finite automaton construction, [12,3,9] for membership testing, and [16,10,14] for minimization. In [6] Watson and Hanneforth consider the parallelisation of finite automaton determinisation and in [15], a high-level CSP-based specification of Brzozowski's DFA construction algorithm was proposed.

The next section discusses Brzozowski's classical sequential DFA construction algorithm. Section 3 then presents a process-oriented implementation of the algorithm, suitable for concurrent execution, and is followed by a performance comparison between the two approaches in Section 4. Section 5 presents some concluding remarks and ideas for future work.

2 Sequential algorithm

Brzozowski's DFA construction algorithm [2] employs the notion of derivatives of regular expressions to construct a DFA. The algorithm takes a regular expression E as input and constructs an automaton that accepts the language represented by E.

The automaton is represented using the normal five-tuple notation $(D, \Sigma, \delta, S, F)$ where D is the set of states; Σ the alphabet; δ the transition relation mapping a state and an alphabet symbol to a state; and $S, F \subseteq D$ are the start and final states, respectively. $\mathcal{L}$ is an overloaded function giving the language of a finite automaton or a regular expression.

Brzozowski's algorithm identifies each DFA state with a regular expression. Apart from the start state, this regular expression is the derivative of a parent state's associated regular expression[1]. Elements of D may therefore interchangeably be referred to either as regular expressions or as states, depending on the context of the discussion.

The well-known sequential version of the algorithm is given in Dijkstra's guarded command language [5] in Figure 1. The notation assumes that the set operations ensure 'uniqueness' of the elements at the level of *similarity* [2, Def 5.2], i.e. $a \in A$ implies that there is no $b \in A$ such that a and b are similar regular expressions.
The algorithm maintains two sets of regular expressions (or states): a set T ('to do') containing the regular expressions for which derivatives need to be calculated; and another set D ('done') containing the regular expressions for which derivatives have

[1] In fact, it can be shown that the language of each state's associated regular expression is also the right language of that state.

```
func Brz(E, Σ) →
    δ, S, F := ∅, {E}, ∅;
    D, T := ∅, S;
    do (T ≠ ∅) →
        let q be some state such that q ∈ T;
        D, T := D ∪ {q}, T \ {q};
        { build out-transitions from q on all alphabet symbols }
        for (a : Σ) →
            { find derivative of q with respect to a }
            d := a⁻¹q;
            if d ∉ (D ∪ T) → T := T ∪ {d}
            ▯ d ∈ (D ∪ T) → skip
            fi;
            { make a transition from q to d on a }
            δ(q, a) := d
        rof;
        if ε ∈ L(q) → F := F ∪ {q}
        ▯ ε ∉ L(q) → skip
        fi
    od;
    return (D, Σ, δ, S, F)
cnuf
```

Figure 1. Brzozowski's DFA construction algorithm

been found already. When the algorithm terminates, T is empty and D contains the states of the automaton which recognises $\mathcal{L}(E)$.

The algorithm iterates through all the elements $q \in T$, finding derivatives with respect to all the alphabet symbols and depositing these new states (regular expressions) into T in those cases where no similar regular expression has already been deposited into $T \cup D$.

Each q, once processed in this fashion, is then removed from T and added into D.

In each iteration of the inner **for** loop (i.e. for each alphabet symbol), the δ relation is updated to contain the mapping from state q to its derivative with respect to the relevant alphabet symbol.

Finally if state q represents a regular expression whose language contains the empty string[2], then that state is included in the set of final states F.

In the forthcoming section we present a process-oriented implementation of the algorithm in which we attempt to structure the algorithm around a number of communicating sequential processes which may benefit from concurrent execution.

3 Concurrent description

We present here an approach to decompose the algorithm into communicating processes. These processes may then be executed concurrently which may result in im-

[2] Such a regular expression is called "nullable".

proved runtimes on multi-processor platforms. Of the many process algebras that have been developed to concisely and accurately model concurrent systems, we have selected CSP [8,7] as a fairly simple and easy to use notation. It is arguably better known and more widely used than most other process algebras. Below, we provide a brief introduction to the CSP operators that are used in the subsequent process definitions.

3.1 Introductory Remarks

CSP is concerned with specifying a system of communicating sequential processes (hence the CSP acronym) in terms of sequences of events, called traces. Various operators are available to describe the sequence in which events may occur, as well as to connect processes. Table 1 briefly outlines the main operators used in this article.

$a \rightarrow P$	event a then process P
$a \rightarrow P \vert b \rightarrow Q$	a then P choice b then Q
$x : A \rightarrow P(x)$	choice of x from set A then $P(x)$
$P \parallel Q$	P in parallel with Q
	Synchronize on common events in the alphabet of P and Q
$b!e$	on channel b output event e
$b?x$	from channel b input to variable x
$P \not< C \not> Q$	if C then process P else process Q
$P; Q$	process P followed by process Q
$P \,\Box\, Q$	process P choice process Q

Table 1. Selected CSP notation

Full details of the operator semantics and laws for their manipulation are available in [8,7]. Note that *SKIP* designates a special process that engages in no further event, but that simply terminates successfully. Parallel synchronisation of processes means that if $A \cap B \neq \emptyset$, then process $(x : A \rightarrow P(x)) \parallel (y : B \rightarrow Q(y))$ engages in some nondeterministically chosen event $z \in A \cap B$ and then behaves as the process $P(z) \parallel Q(z)$. However, if $A \cap B = \emptyset$ then deadlock results. A special case of such parallel synchronisation is the process $(b!e \rightarrow P) \parallel (b?x \rightarrow Q(x))$. This should be viewed a process that engages in the event $b.e$ and thereafter behaves as the process $P \parallel Q(e)$. This can also be interpreted as processes communicating over a channel b. The one process writes e onto channel b and the other process reads from channel b into variable x.

3.2 Process descriptions

The concurrent version of the algorithm can be modeled as a process *BRZ* which is composed of four concurrent processes which may themselves be composed of more processes. The first of these processes is named *OUTER* and corresponds to the outer loop of the algorithm in Figure 1. It is responsible for maintaining the sets D, T, and F. The second process computes the derivatives of regular expressions and is called *DERIV*. A *FANOUT* process is responsible for distributing a regular expression to the components of *DERIV*. The final process *UPDATE* modifies the transition relation δ. The process definition for *BRZ(T,D,F,δ)* is thus:

$$BRZ(T, D, F, \delta) = OUTER(T, D, F) \parallel FANOUT \parallel DERIV \parallel UPDATE(\delta)$$

Process $OUTER$ is modelled as a process which initialises its local state and then repeatedly performs actions to modify T, D, and F as well as its local state. This repetition is modelled as process $LOOP$.

$$OUTER(T, D, F) = init \rightarrow LOOP(T, D, F)$$

$LOOP$ first modifies the local state and has a choice between the following behaviours. It may write $q \in T$ to channel $outNode$ and repeat, it may receive a new regular expression d from channel $inNode$ and repeat, or it may terminate if the local state is such that no more states need to be processed. When $LOOP$ sends q out, q is removed from T and added to D and if $\varepsilon \in \mathcal{L}(q)$ then q is also added into F. In the case when a new state is received it is added into T if no similar state is in $T \cup D$.

$$\begin{aligned} LOOP(T, D, F) = {} & modifyLocalState \rightarrow \\ & ((q : T \rightarrow outNode!q \rightarrow \\ & \quad LOOP(T \setminus q, D \cup q, F \cup q) \not< \varepsilon \in \mathcal{L}(q) \not> LOOP(T \setminus q, D \cup q, F)) \\ & \Box \\ & (inNode?d \rightarrow LOOP(T \cup d, D, F) \not< d \notin T \cup D \not> LOOP(T, D, F)) \\ & \Box \\ & SKIP) \end{aligned}$$

$DERIV$ is responsible for concurrently calculating the derivatives of a regular expression with respect to each alphabet symbol. This corresponds to the inner `for` loop in Figure 1. $DERIV$ is thus modelled as the concurrent composition of $|\Sigma|$ processes, each responsible for calculating the derivative with respect to a given $i \in \Sigma$.

$$DERIV = \|_{i:\Sigma}\ DERIV_i$$

Each $DERIV_i$ repeatedly reads a regular expression re from its input channel $dOut_i$, calculates the derivative and then sends the result as a triple $\langle re, i, i^{-1}re \rangle$ out on a shared channel $derivChan$.

$$DERIV_i = dOut_i?re \rightarrow computeDeriv.re \rightarrow derivChan!\langle re, i, i^{-1}re \rangle \rightarrow DERIV_i$$

The $FANOUT$ process connects $OUTER$ and $DERIV$ and is responsible for communicating the regular expressions from $OUTER$ to each $DERIV_i$. It repeatedly reads a regular expression from its input channel $outNode$ and concurrently replicates it to the $|\Sigma|$ output channels $dOut_i$.

$$FANOUT = (outNode?re \rightarrow \|_{i:\Sigma}\ (dOut_i!re \rightarrow SKIP)); FANOUT$$

In order to complete the DFA, we need to record all the state transitions in δ. This is the responsibility of $UPDATE$. It is modelled as a repeating process which reads a triple $\langle re, i, d \rangle$ from its input channel $derivChan$ and records $\delta(re, i) = d$. It also sends one element of the triple, d, on to $OUTER$ via channel $inNode$. This d is potentially a new state from which transitions should be calculated and hence it should be added into T if a similar node has not been processed before.

$$UPDATE(\delta) = derivChan?\langle re, i, d \rangle \rightarrow inNode!d \rightarrow UPDATE(\delta \cup \langle re, i, d \rangle)$$

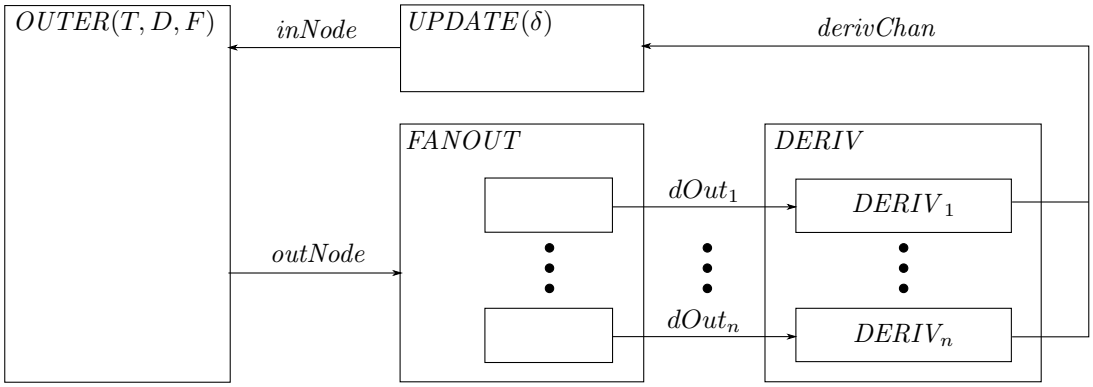

Figure 2. The communications network of the *BRZ* process.

Figure 2 shows the communicating processes along with their associated input and output channels.

Termination is not addressed completely in the above process models. Notably, processes which repeatedly read from an input channel live until their input channels are closed. Consequently they can be modelled as a choice between reading from the channel and *SKIP*. These choices were omitted above to simplify the presentation.

In the following section we compare the performance of the concurrent implementation against the sequential algorithm.

4 Performance comparison

In the preceding section the Brzozowski DFA construction algorithm was decomposed into a network of communicating processes. The next step is then to implement the CSP descriptions as an executable program and compare the performance of the sequential and concurrent algorithms.

It was decided to use the Go programming language [17] for the implementation. Go is a compiled language with concurrency features inspired by Hoare's CSP. Particularly relevant to the present context is the fact that Go has channels as first class members of the language. The CSP processes in the process network from Section 3 map to so-called go-routines and the communication channels map to Go channels. An alternative language that implements CSP-like channels of which we are aware is occam-π [1]. Go was chosen over occam-π since Go allows us to implement data structures more easily and documentation is also more readily available.

4.1 Experimental setup

The aim of the present experiment is simply to test the hypothesis that the concurrent implementation can construct DFAs faster than the sequential version. No attempt was made to investigate completely the performance characteristics of the process-oriented implementation.

In order to compare the performance the following approach was followed. A regular expression was generated and both the sequential and concurrent algorithms were executed with this regular expression. The respective execution times were recorded. In order to reduce the effects of transient operating system events, each construction

was executed 30 times and the minimum duration was used as the data point for that regular expression. Various regular expressions were used as input to observe the performance of the algorithms over a range of input.

Regular expressions were randomly generated via a simple recursive procedure $gen(\Sigma, d)$. The procedure takes as input two parameters: an alphabet Σ and an integer d. If $d = 0$ the procedure returns a random symbol from Σ. If $d > 0$ then $gen(\Sigma, d)$ randomly chooses a regular expression operator and then recursively generates the required operands for the operator by calling $gen(\Sigma, d-1)$. The size of the regular expression is thus controlled by d since d defines the depth of the expression tree for the regular expression. The upper bound for the number of operators in the tree is $2^d - 1$ and for the number of symbols (leaves) it is 2^d. Many generated regular expressions will be smaller since some regular expression operators are unary operators which result in a tree that is smaller than a complete binary tree.

We decided to consider the performance of the algorithms with regular expressions over both a small alphabet and a larger alphabet. The small alphabet contained 4 symbols and the larger alphabet 85 symbols.

Regular expression were generated with depths $d = 5, 6, \ldots, 10$. For each of the 12 elements in $\{4, 85\} \times \{5, 6, \ldots, 10\}$ we generated 50 regular expressions.

The implementations were compiled in Go version 1.2.2 and initially executed in Mac OS X 10.7.5 on a MacPro1,1 with two Dual-Core Intel Xeon 2.66 GHz processors and 5 GB RAM. The runtimes of the programs on this platform are analysed in the next section.

4.2 Observations

Let us now consider the results of the experiment. The results will be communicated mainly through graphical plots and a few statistical calculations. These plots and statistical calculations were produced using the statistical system R version 3.1.0 [13]. In most cases the two alphabet cases were considered separately.

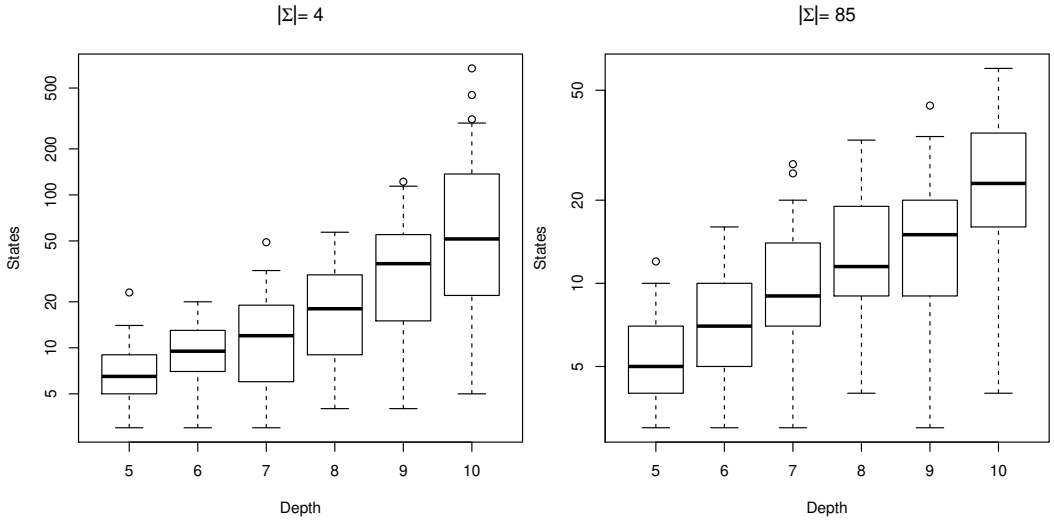

Figure 3. Sizes of automata generated.

First we consider the sizes of the automata constructed by the algorithms. The plots in Figure 3 show the number of states (on a logarithmic scale) in the resultant

automata. The plots in the figure provide a visualisation of the distribution of the 50 values for each of the depths. As expected, the regular expressions with larger depths generally yielded larger automata. It should also be noted that for the small alphabet case a few very large automata were constructed. The data confirm that the input regular expressions did indeed vary significantly and hence the algorithms were executed with a variety of input. In future work a more sophisticated approach to obtain input should be considered so that one may better control the nature of the input regular expressions.

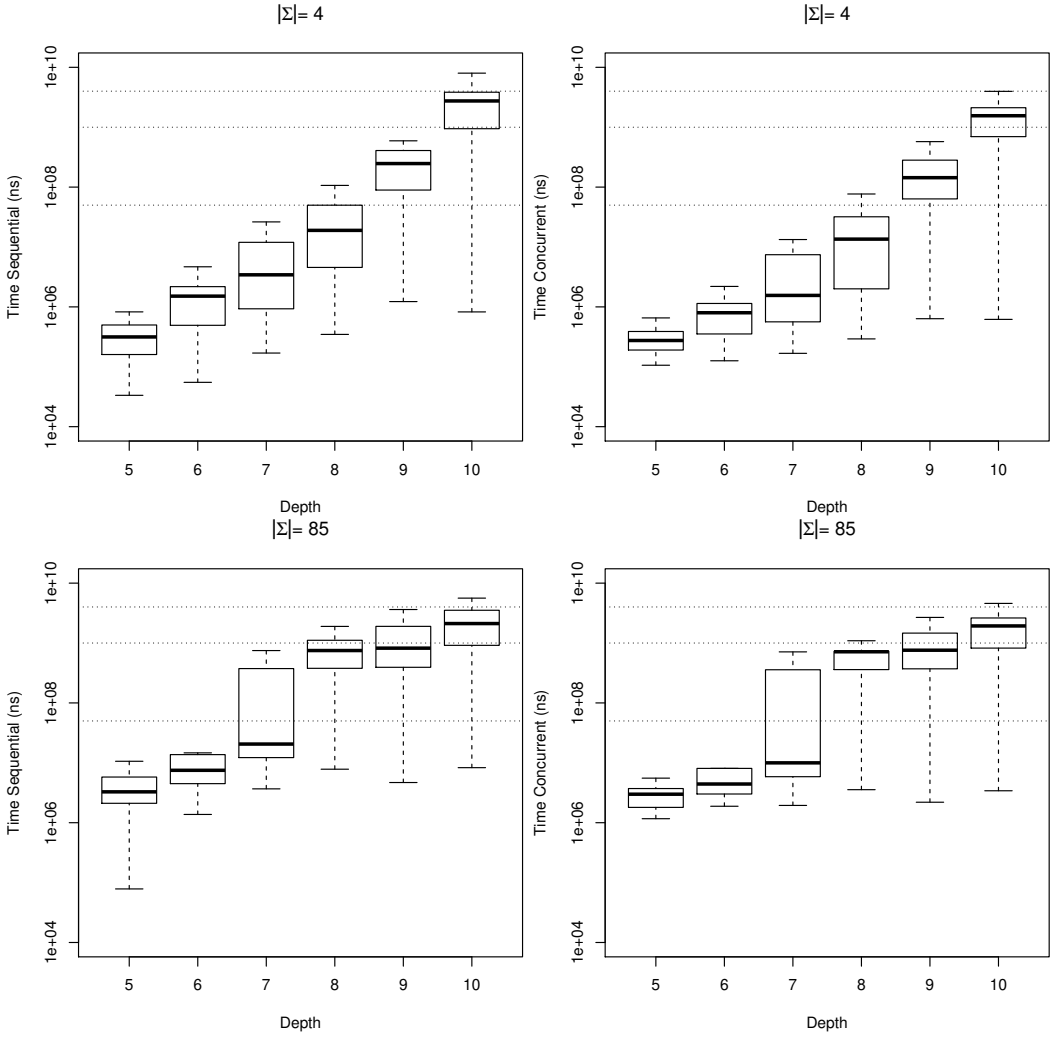

Figure 4. Construction times against problem size.

Let us now consider the construction times. Figure 4 shows the construction times for the sequential algorithm and the concurrent algorithm in both of the alphabet cases for the various regular expression sizes. Note that in each plot the y-axis is logarithmic. Outlying observations were omitted from the plots to make them clearer.

From the plots it is clear that in all cases the time increased as the regular expression grew. It can also be seen – although less clearly due to the logarithmic scale – that the construction time for the concurrent algorithm tends to be lower than that of the sequential algorithm. The construction time difference is less pronounced in the large alphabet case. This could be due to the larger overhead involved in this

case. For example, when one considers the process *FANOUT* from Figure 2 it will be seen that it creates a process for each alphabet symbol. As the alphabet grows, this creation and scheduling overhead will also increase.

The data from the plots suggest that the concurrent algorithm may indeed be faster. To confirm this we performed the Wilcoxon signed rank test for paired observations. The test tests the null hypothesis that the median difference between the pairs of construction times is zero against the alternative that the median difference is greater than zero:

$$H_0 : \text{median difference between runtimes is } 0$$
$$H_a : \text{median difference between runtimes is greater than } 0$$

The results of the tests are as follows.

	N	Test statistic	$p < 0.01$
$\lvert\Sigma\rvert = 4$	300	44482	Yes
$\lvert\Sigma\rvert = 85$	300	44141	Yes

In both the small alphabet and large alphabet cases the null hypothesis can be rejected at 99% level of confidence. Our data thus provides evidence in support of our hypothesis that the concurrent algorithm can outperform the sequential one.

To explore further the relationship between the sequential and concurrent construction times scatterplots were constructed. These can be found in Figure 5. Each point represents a pair of sequential and concurrent runtimes for a given regular expression. The x-coordinate of the point is the sequential runtime and the y-coordinate is the concurrent runtime. Each plot contains $50 \times 6 = 300$ points. From the plots

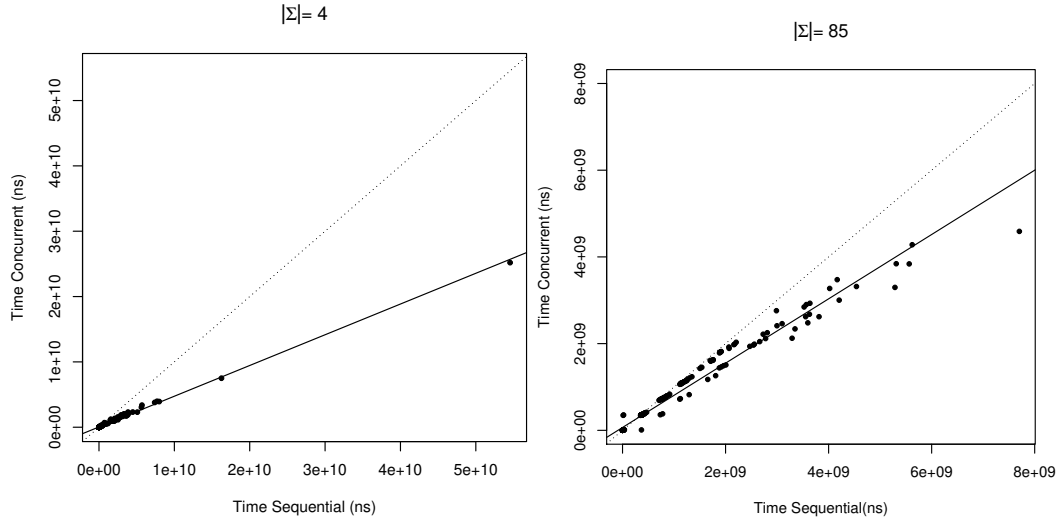

Figure 5. Scatter plots of sequential time against concurrent time.

it can be seen that there appears to be a linear relationship between the sequential and concurrent runtimes. Linear regression lines were fitted to data and also plotted on the graph as solid lines. The dotted line in each plot is simply a line through the origin with slope 1. From the graph it can be seen that the slope of the regression line for the small alphabet case is less steep than that of the regression line for the large alphabet case.

If we let T_c and T_s be the construction times for the concurrent and sequential algorithms respectively, then the regression lines that were fitted are as follows.

$$T_c = 39.6\,\text{ms} + 0.47 \cdot T_s \qquad \text{for } |\Sigma| = 4$$
$$T_c = 65.7\,\text{ms} + 0.74 \cdot T_s \qquad \text{for } |\Sigma| = 85$$

The fact that the slopes are less than one is consistent with the fact that the concurrent construction times are smaller than the sequential times. From the slope terms in the equations above it is clear that the performance increase for the small alphabet case was greater than for the large alphabet case. As mentioned earlier this can be explained by the greater amount of overhead present in the large alphabet case.

Speedup and efficiency are well-known metrics for characterising parallel algorithmic performance [11]. Speedup is defined as the execution time of the sequential program divided by the execution time of the parallel program. Efficiency is defined as the speedup divided by the number of processes. Table 2 contains the observed speedup and efficiency for our experiment. Each entry shows the median for the relevant subset of the data.

	Speedup		Efficiency	
Depth	$\|\Sigma\| = 4$	$\|\Sigma\| = 85$	$\|\Sigma\| = 4$	$\|\Sigma\| = 85$
All	1.72	1.09	0.43	0.27
5	1.15	1.21	0.29	0.30
6	1.84	1.45	0.46	0.36
7	1.82	1.43	0.46	0.36
8	1.80	1.06	0.45	0.27
9	1.71	1.09	0.43	0.27
10	1.83	1.21	0.46	0.30

Table 2. Speedup and efficiency overall and for different problem sizes.

Ideal speedup in a p processor environment is p and efficiency equal to 1 is very good. The median speedup for the small alphabet case is 1.72 and for the large alphabet case it is 1.09. Recall that the total number of cores in our experimental platform was four. We have clearly not achieved optimal speedup, but we have a median speedup greater than one. From this and from the relatively low efficiency numbers it is clear that the process-oriented approach is promising especially if we can reduce the amount of overhead.

Finally, let us consider whether the problem size influences the speedup. Figure 6 shows the plots for speedup against the depths used to generate the regular expressions. The plots show that the speedup for smaller regular expressions is sometimes less than 1. This implies that the concurrent version is sometimes slower than the sequential version for these smaller regular expressions. This effect is more pronounced in the large alphabet case. In the small alphabet case, the effect is seen at depths 5 and 6, but in the large alphabet case the speedup less than 0 is also found at depth 7. In the smaller regular expressions the overhead removes entirely the performance gain of the concurrent processes. In larger expressions the nett gain is still positive.

4.3 A second experiment

In order to gain insight into whether or not the foregoing results are reasonably robust across different platforms, we repeated the experiment on a newer machine with a

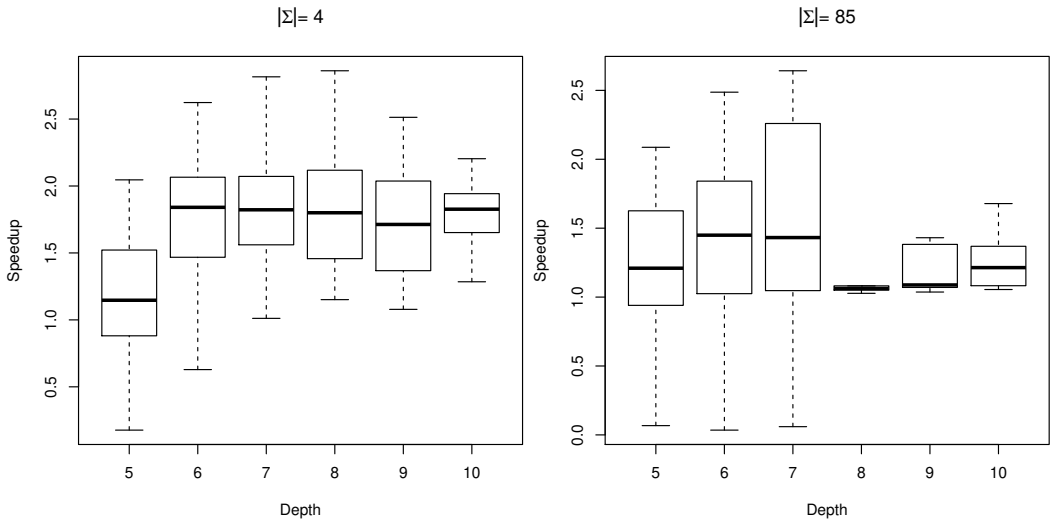

Figure 6. Speedup against problem size.

slightly different configuration. This machine is a MacBookPro11,3. It has a four-core *single* Intel i7 processor running at 2.3 GHz, whereas the earlier machine had *two* Dual-Core Intel Xeon processors, each running at 2.66 GHz. The newer machine ran Mac OS X 10.9.3 as opposed to the earlier machine which ran Mac OS X 10.7.5. Finally, the newer machine had considerably more RAM—16 GB compared to the earlier machine's 5 GB.

The data produced by this experiment results in figures that are broadly similar to those in Figures 3, 4 and 5, but on a somewhat different scale—i.e. there is a general increase in speedup and efficiency. The best speedup attained was 2.2, representing an efficiency of 0.55. This was for the large alphabet when the parameter d for the depth of regular expression trees was set to 5. Rather than providing all the raw values, Table 3 shows the increases in speedup and efficiency as a *percentage* of the corresponding data in Table 2.

Note that the improvement is largest for the large alphabet case where overall speedup and efficiency increases are attained of approximately 40%. The table also illustrates that these gains tend to diminish as the problem size increases. However, there is no obvious relationship between problem size and the extent to which the gains diminish. For the largest problem size, the speedup and efficiency increases on the large alphabet were around 20%. By way of comparison, there was a mere 2% to 3% increase in the case of the small alphabet.

It has already been pointed out that in the concurrent design that has been implemented, an increase in alphabet size results in an increase in process creation and scheduling. There results suggest that the overhead required to create and schedule processes over four cores on the same CPU is somewhat more efficient than achieving the same task across two dual-core CPUs. It has been left, however, to future research to carry out a more fine-grained analysis to determine the contribution of other factors to improved performance, such as the increase in RAM size, the small increase in clock speed and the more recent version of the operating system.

	Speedup Increase		Efficiency Increase	
Depth	$\|\Sigma\| = 4$	$\|\Sigma\| = 85$	$\|\Sigma\| = 4$	$\|\Sigma\| = 85$
All	12.8%	40.4%	14.0%	40.7%
5	22.6%	54.5%	20.7%	56.7%
6	2.7%	51.7%	2.2%	52.8%
7	15.9%	44.8%	15.2%	44.4%
8	15.6%	13.2%	15.6%	11.1%
9	18.1%	16.5%	16.3%	14.8%
10	3.3%	18.2%	2.2%	20.0%

Table 3. Speedup and efficiency increase on the four core machine.

5 Conclusion

We set out to test whether a process-oriented implementation of Brzozowski's DFA construction algorithm could outperform the sequential implementation in a multi-processor environment. The results of our experiment, carried out on two different (but similar) platforms, confirm that it is indeed possible. In neither case was the speedup close to ideal. Nevertheless, there were instances where double the speed of the sequential algorithm was reached. Even though this represents an efficiency of about 50%, the results are a big improvement over the sequential algorithm's runtime in light of typical under-utilization of multi-core capabilities of present-day CPUs.

Inefficiencies in the process-oriented implementation are, no doubt, part of the reason why efficiency measurements are not higher. The *FANOUT* process, in particular, could be enhanced to be more efficient by creating fewer processes. Future work would include improving on implementation efficiency and exploring further algorithms to implement in a process-oriented manner. It will also be interesting to verify results to date on a wider variety of platforms.

In the first experimental setup we used a machine with two CPUs, each with two cores. We conjecture that the operating system may schedule the threads of the executable to run only on one of the two processors—effectively utilising only two cores for the execution. If this turns out to be true, the observed speedup is, especially in the small alphabet case, rather closer to the ideal. This matter should be investigated further. As a simple first step, we repeated the experiment on a machine with a single processor with four cores and compared the results, obtaining somewhat improved efficiencies. More sophisticated profiling tools are, however, needed to examine the behaviour of the running processes in finer detail.

This uncertainty regarding the operating system's scheduling behaviour raises the theme of control over scheduling of tasks. A number of questions immediately come to mind. Would greater speedups be possible if such control was readily available? What (if any) are the disadvantages of granting greater control to software developers? Could such control mechanisms not be built into the operating systems, accompanied by appropriate escape measures in case the user abuses these mechanisms? These questions open up various avenues for further research.

References

1. F. Barnes and P. Welch: *occam-pi: blending the best of CSP and the pi-calculus.* `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.
2. J. A. Brzozowski: *Derivatives of regular expressions.* Journal of the ACM, 11(4) 1964, pp. 481–494.
3. B. Burgstaller, Y.-S. Han, M. Jung, and Y. Ko: *On the parallelization of DFA membership tests*, tech. rep., Technical Report. TR-0003, Department of Computer Science, Yonsei University, Seoul 120–749, Korea. http://elc. yonsei. ac. kr/PDFA. html, 2011.
4. H. Choi and B. Burgstaller: *Non-blocking parallel subset construction on shared-memory multicore architectures*, in Proceedings of the Eleventh Australasian Symposium on Parallel and Distributed Computing-Volume 140, Australian Computer Society, Inc., 2013, pp. 13–20.
5. E. W. Dijkstra: *A Discipline of Programming*, Prentice Hall, 1976.
6. T. Hanneforth and B. W. Watson: *An efficient parallel determinisation algorithm for finite-state automata*, in Stringology, J. Holub and J. Žďárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2012, pp. 42–52.
7. C. A. R. Hoare: *Communicating sequential processes.* Communications of the ACM, 26(1) 1983, pp. 100–106.
8. C. A. R. Hoare: *Communicating sequential processes (electronic version)*, 2004, `http://www.usingcsp.com/cspbook.pdf`.
9. J. Holub and S. Štekr: *On parallel implementations of deterministic finite automata*, in Implementation and Application of Automata, S. Maneth, ed., vol. 5642 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 54–64.
10. J. JáJá and K. W. Ryu: *An optimal randomized parallel algorithm for the single function coarsest partition problem.* Parallel Processing Letters, 6(2) 1996, pp. 187–193.
11. A. H. Karp and H. P. Flatt: *Measuring parallel processor performance.* Commun. ACM, 33(5) May 1990, pp. 539–543.
12. Y. Ko, M. Jung, Y.-S. Han, and B. Burgstaller: *A speculative parallel DFA membership test for multicore, simd and cloud computing environments.* International Journal of Parallel Programming, 2012, pp. 1–34.
13. R Core Team: *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2014.
14. B. Ravikumar and X. Xiong: *A parallel algorithm for minimization of finite automata*, in IPPS, IEEE Computer Society, 1996, pp. 187–191.
15. T. Strauss, D. G. Kourie, and B. W. Watson: *A concurrent specification of Brzozowski's DFA construction algorithm.* Int. J. Found. Comput. Sci., 19(1) 2008, pp. 125–135.
16. A. Tewari, U. Srivastava, and P. Gupta: *A parallel DFA minimization algorithm*, in HiPC, S. Sahni, V. K. Prasanna, and U. Shukla, eds., vol. 2552 of Lecture Notes in Computer Science, Springer, 2002, pp. 34–40.
17. The Go Authors: *The Go programming language.* `http://golang.org/`.
18. D. Ziadi and J.-M. Champarnaud: *An optimal parallel algorithm to convert a regular expression into its Glushkov automaton.* Theoretical Computer Science, 215(1-2) February 1999, pp. 69–87.