An Efficient Parallel Determinisation Algorithm for Finite-state Automata

Thomas Hanneforth and Bruce W. Watson

[1] Universität Potsdam, Germany
[2] Stellenbosch University, South Africa
Thomas.Hanneforth@uni-potsdam.de Bruce@fastar.org

Abstract. Determinisation of non-deterministic finite automata (NFA) is an important operation not only for optimisation purposes, but also the prerequisite for the complementation operation, which in turn is necessary for creating robust pattern matchers, for example in string replacement and robust parsing. In the paper, we present an efficient parallel determinisation algorithm based on a message-passing graph approach. In a number of experiments on a multicore machine we show that the parallel algorithm behaves very well for acyclic and cyclic NFAs of different sizes, especially in the worst case, where determinisation leads to an exponential blow-up of states.

Keywords: finite-state automata, determinisation, parallel algorithms, message passing, flow graphs, Kahn process networks, replacement rules

1 Introduction

Given a nondeterministic finite automaton (an NFA), *determinisation* is the construction of an equivalent deterministic finite automaton (DFA), where 'equivalence' means that the NFA and DFA accept the same language. Many real-life applications involve the relatively straightforward construction and manipulation initially of NFA's, for example when compiling regular expressions, regular grammars, or other descriptive formalisms (such as replacement rules in computational linguistics) to finite automata. While NFAs are often very compact and easily manipulated, several situations motivate the subsequent construction of a DFA:

- The standard approach for considering *equivalence of two automata* [4] is to minimize their respective equivalent DFA's and compare those (thanks to the uniqueness-modulo-isomorphism of minimal DFA's per language).
- Effective *complementation* of regular languages requires the construction of a DFA (cf. [4]).
- Complementation is also the key for *robust* natural language processing applications based on finite-state automata, e.g. shallow parsing systems etc. Many of these systems are build upon regular conditional [6] and unconditional replacement rules [7] which heavily rely on complementation to ensure robust application.
- The end-goal of constructing automata is often to apply it to a string, e.g. for pattern matching, network security applications, and computational linguistics. *Determinism* of the DFA means that only a single 'current state' needs to be tracked while processing input. By contrast, in the worst case, all of an NFA's states may become active while processing input – an enormous computational overhead as each symbol is processed, and usually impractical. In all of those applications, a DFA is essential [1].

Thomas Hanneforth, Bruce W. Watson: An Efficient Parallel Determinisation Algorithm for Finite-state Automata, pp. 42–52.
Proceedings of PSC 2012, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05095-8 © Czech Technical University in Prague, Czech Republic

The classical 'subset construction' algorithm[1] follows directly from Rabin & Scott's proof of NFA/DFA equivalence, and also shows that an equivalent DFA can be exponentially larger than the NFA in the worst-case [4]. Most real-life implementations combine reachability with the subset construction, which can subsequently be tuned quite effectively. In addition to tuning for memory and speed performance, various toolkits also implement *incremental* determinisation in which the DFA is constructed on-the-fly while processing an input string. Recent work by van Glabbeek & Ploeger [3] presents five determinisation algorithms, classifying them in lattice (based on the resulting DFA size), and giving benchmarking results.

Despite these algorithmic advances, there has been little work on parallel determinisation. Clock-speeds of modern processors and memory have plateaued and Moore's Law advances in silicon chip production are now devoted to more processor cores, enabling cheap multi-threading, with the caveat that parallel algorithms are less well-known and much more difficult to get correct. This paper presents one of the first such parallel algorithms.

Before we turn to the algorithm, we define the relevant technical notions in the next section. Then, Section 3 restates the standard reachability-based serial determinisation algorithm, before it develops an efficient parallel one. In Section 4, we give a short C++ code fragment which implements the parallel algorithm. Finally, in Section 5, we report on several experiments we conducted to compare serial and parallel determinisation.

2 Preliminaries

An alphabet Σ is a finite set of symbols. A string $x = a_1 \cdot a_2 \cdots a_n$ over Σ is a finite concatenation of symbols a_i taken from Σ (the concatenation operator $\cdot$ is normally omitted). The length of a string $x = a_1 \cdots a_n$ – symbolically $|x|$ – is n. The empty string is denoted by ε and has length zero. Let Σ^* denote the set of all finite-length strings (including ε) over Σ.

A *non-deterministic finite-state automaton* (NFA) A is a 5-tuple $\langle Q, \Sigma, q_0, \delta_{nd}, F\rangle$ with Q being a finite set of states; Σ, an *alphabet*, $q_0 \in Q$, the start state; $\delta_{nd} : Q \times \Sigma \mapsto 2^Q$, the transition function; and $F \subseteq Q$, the set of final states.

Define $\delta^*_{nd} : Q \times \Sigma^* \mapsto 2^Q$ as the reflexive and transitive closure of δ_{nd}:

- $\forall q \in Q, \delta^*_{nd}(q, \varepsilon) = \{q\}$ and
- $\forall q \in Q, a \in \Sigma, w \in \Sigma^* : \delta^*_{nd}(q, aw) = \bigcup_{p \in \delta_{nd}(q,a)} \delta^*_{nd}(p, a)$.

δ_{nd} may be a partial function. In case $\delta_{nd}(S, a)$ is undefined for some state set $S \subseteq Q$ and $a \in \Sigma$, we take $\delta_{nd}(S, a)$ be equal to $\emptyset$. The language of a NFA $A = \langle Q, \Sigma, q_0, \delta_{nd}, F\rangle$, symbolically $L(A)$, is defined as $L(A) = \{w \in \Sigma^* \mid \delta^*_{nd}(q_0, w) \cap F \neq \emptyset\}$.

A *deterministic finite-state automaton* (DFA) A is defined as a 5-tuple $\langle Q, \Sigma, q_0, \delta_d, F\rangle$ where A, Q, q_0, and F are the same as in the NFA case and δ_d is a (partial) function mapping $Q \times \Sigma$ to Q. The notions of δ^*_d and $L(A)$ are defined analogously to the ones in NFAs.

A state q is *reachable* if there exists a word $w \in \Sigma^*$ such that $\delta^*_d(q_0, w) = q$.

For every NFA, an equivalent DFA (with respect to the recognized language) can be constructed. The key idea is the *subset construction*:

[1] Sometimes known as the 'powerset construction', see the next section.

Definition 1 (Subset construction). *Let $A = \langle Q, \Sigma, q_0, F, \delta_{nd} \rangle$ be an NFA. Define A', the equivalent DFA with $L(A') = L(A)$ as $A' = \langle 2^Q, \Sigma, \{q_0\}, F', \delta_d \rangle$ with:*

- $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$
- $\delta_d(S, a) = \bigcup_{q \in S} \delta_{nd}(q, a), \ \forall a \in \Sigma, \forall S \subseteq Q$

The next section describes serial and parallel determinisation algorithms based on the subset construction.

3 Determinisation algorithms

This section recapitulates the standard serial determinisation algorithm and introduces our parallel version of it.

3.1 Serial determinisation

A naïve NFA determinisation algorithm implementing Definition 1 directly would lead to worst-case behaviour in every case. In practice, it turns out that most of the states in the powerset of Q are not reachable from the start state of the DFA. Thus, their creation can be completely avoided by incorporating a reachability constraint into the algorithm. This leads directly to the queue-based version shown in Algorithm 1.

Algorithm 1: Serial NFA determinisation algorithm

```
   Input: NFA $A = \langle Q, \Sigma, q_{0_{nd}}, \delta_{nd}, F \rangle$
   Output: DFA $A' = \langle Q', \Sigma, q_{0_d}, \delta_d, F' \rangle$
 1 $R(\{q_{0_{nd}}\}) \leftarrow c \leftarrow q_{0_d} \leftarrow 0$
 2 $L \leftarrow \emptyset$
 3 $Q' \leftarrow F' \leftarrow \emptyset$
 4 $Enqueue(L, \langle \{q_{0_{nd}}\}, q_{0_d} \rangle)$
 5 while $L \neq \emptyset$ do
 6     $\langle S, q \rangle \leftarrow Dequeue(L)$
 7     $Q' \leftarrow Q' \cup \{q\}$
 8     if $S \cap F \neq \emptyset$ then
 9         $F' \leftarrow F' \cup \{q\}$
10     $C = \{\langle a, \bigcup_{p \in S} \delta_{nd}(p, a) \rangle \in \Sigma \times 2^Q \mid \exists r \in S : \delta_{nd}(r, a) \neq \emptyset\}$
11     for each $\langle a, S' \rangle \in C$ do
12         $p \leftarrow R(S')$
13         if $p = \top$ then
14             $c \leftarrow c + 1$
15             $R(S') \leftarrow p \leftarrow c$
16             $Enqueue(L, \langle S', p \rangle)$
17         $\delta_d(q, a) \leftarrow p$
```

Algorithm 1 uses several auxiliary data structures: First of all, $R : 2^Q \mapsto \mathbb{N} \cup \{\top\}$ is a *state register* mapping subsets of Q to natural numbers. If some set S is not in the register, $R(S)$ returns $\top$. Initially, the set containing q_{0_d} is mapped to zero. Furthermore, the algorithm maintains a queue L holding pairs $\langle S, q \rangle \in 2^Q \times \mathbb{N}$ and a global state counter c initially set to 0. In line 4, the initial pair $\langle \{q_{0_{nd}}\}, 0 \rangle$ is added to the queue which is subsequently processed in the **while**-loop between lines 5 and 17. In line 6, a pair $\langle S, q \rangle$ is removed from L. If S contains a final state, q is added to the final states of the DFA. In line 10, a set C of *candidate states* is constructed. For this purpose, a set $\Sigma' \subseteq \Sigma$ is created such that a is in Σ' if $\delta_{nd}(r, a)$ is defined (that is, $\delta_{nd}(r, a) \neq \emptyset$) for some $r \in S$. Then, for each $a \in \Sigma'$, a new state set S' is assembled holding all the destination states $\delta_{nd}(p, a)$ for all $p \in S$. In the following, we will refer to this step as the *symbol indexing step*. The **for**-loop in lines 11–17 processes all pairs $\langle a, S' \rangle$. Line 12 looks up state set S' in the register R. If S' is not found in R, it is added to R by assigning it a new state number p by incrementing the global state counter c (line 14–15). Furthermore, the pair $\langle S', p \rangle$ is added to the queue L (line 16). In both cases, a new transition from q to p with a is added to δ_d (line 17).

By maintaining a queue L, the algorithm ensures that each state q added to Q' in line 7 is reachable from that start state q_{0_d}. Nevertheless, in the worst case, all subsets of Q are added to the queue resulting in a running time in $\mathcal{O}(|2^Q||\Sigma|)$.

Given an alphabet $\Sigma = \{a, b\}$, the worst case is exihibited by NFAs resulting from regular expressions $r(k)$ of the form $\Sigma^* a(a+b)^k$ which leads to DFA with 2^{k+1} states. Figure 1 shows an NFA constructed from $r(2)$, while Figure 2 shows the equivalent DFA. Note that DFA constructed from regular expressions $r(k)$ are also complete, that is, δ_d is a total function.

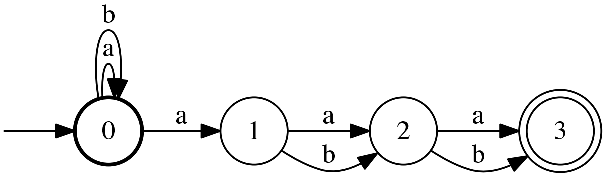

Figure 1. NFA created from regular expression $\Sigma^* a(a+b)^2$

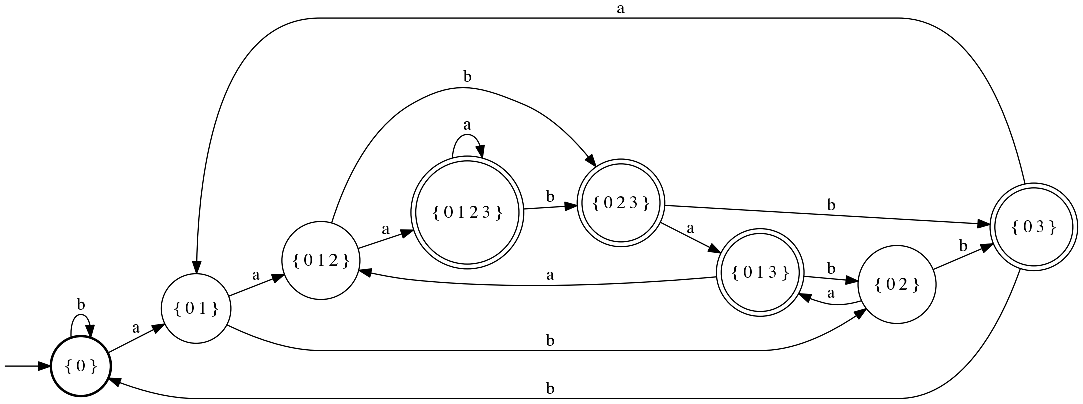

Figure 2. Equivalent DFA to the NFA of Figure 1

This worst case of exponential blow-up may not be so uncommon in practice as one might expect. Consider a *pattern matching* problem [1] where some finite set P of patterns is to be efficiently found in some given input text. In automata-theoretic terms, this amounts to constructing an NFA for $\Sigma^* \cdot P$, the infinite regular language consisting of all strings having some $p \in P$ as a suffix. If P has the form $a(a+b)^k$ or something similar, then determinisation is exponential.

3.2 Parallel determinisation

When looking at Algorithm 1 for parts which could be run in parallel and which can not, the following observations could be made:

- The **while**-loop between lines 5 and 17 is a good candidate for parallel processing, since several pairs $\langle S, q\rangle$ could be removed from the queue (line 6) and further processed in parallel.
- This is in particular the case for the symbol indexing step in line 10, since the creation of follower candidate states for each state set S is completely independent for all state sets S.
- The **for**-loop (lines 11–17) could in principle be parallelised, but the main actions in its body – querying/adding to the state register and adding new transitions to the DFA – must certainly be serialised.
- The same is true for adding final states to the DFA (line 8–9). Assuming a suitable data structure for the DFA, adding final states (line 9) and adding transitions (line 17) can certainly be done in parallel.
- Incrementing the state counter (line 14) must again be serialised.

A straightforward way to link parallel and serial components of the algorithm are the concepts of *Kahn Process Networks*, (cf. [5]) and *Labeled Transition Systems* ([2]).

Definition 2 (Labelled Transition System (LTS), cf. [2]). *Let a channel c be an unbounded FIFO-queue (first-in-first-out queue) with elements taken from some alphabet Σ_c. Let $Chan$ denote the set of all channels.*
An LTS is a tuple $\langle S, s_0, I, O, Act, \rightarrow\rangle$ consisting of a set S of states, an initial state $s_0 \in S$, a set $I \subseteq Chan$ of input channels, a set $O \subseteq Chan$ (distinct from I) of output channels, a set Act of actions consisting of input actions $\{c?a \mid c \in I, a \in \Sigma_c\} \subseteq Act$, output actions $\{c!a \mid c \in O, a \in \Sigma_c\} \subseteq Act$ and a labelled transition relation $\rightarrow \;\subseteq S \times Act \times S$.

Definition 3 (Kahn Process Network (KPN), cf. [2]). *A Kahn process network is a tuple $\langle P, C, I, O, Act, \{LTS_p \mid p \in P\}\rangle$ with the components as follows:*

- *P is a finite set of processes.*
- *C, I and O ($\subseteq Chan$) are finite and pairwise disjunct sets of internal channels, input channels and output channels, respectively.*
- *$Act = \{c?a, c!a \mid c \in C \cup I \cup O, a \in \Sigma_c\}$*
- *Every process $p \in P$ is defined by a* sequential *labelled transition system*[2] *$LTS_p = \langle S_p, s_{p_0}, I_p, O_p, Act, \xrightarrow{p}\rangle$, with $I_p \subseteq I \cup C$ and $O_p \subseteq O \cup C$.*
- *For every channel $c \in C \cup I$, there is exactly one process $p \in P$ that reads from it ($c \in I_p$) and for every channel $c \in C \cup O$, there is exactly one process $p \in P$ that writes to it ($c \in O_p$).*

Since KPNs are essentially graph structures, they admit an intuitive graphical representation. Figure 3 shows a KPN for the parallel version of the determinisation algorithm.

The start state labelled s_0 in Figure 3 starts the network by passing the initial pair $\langle\{q_{0_{nd}}\}, q_{0_d}\rangle$ to the state labelled *process state set*. This corresponds to enqueuing the initial pair in line 4 of Algorithm 1. State *process state set* – which basically implements the body of the **while**-loop in Algorithm 1 – is connected with 3 other nodes:

[2] Basically, a *sequential* LTS accepts at most one input/output operation at a given point in time.

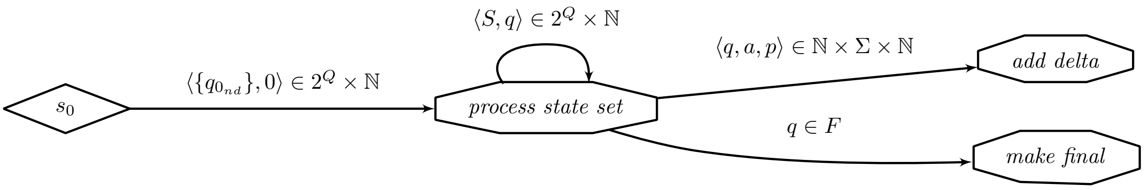
$\langle S, q\rangle \in 2^Q \times \mathbb{N}$
$\langle \{q0_{nd}\}, 0\rangle \in 2^Q \times \mathbb{N}$
$\langle q, a, p\rangle \in \mathbb{N} \times \Sigma \times \mathbb{N}$
s_0
process state set
add delta
$q \in F$
make final

Figure 3. KPN for the parallel determinisation algorithm. Nodes with octogon shape are parallel nodes

1. with itself. This corresponds to line 15 of Algorithm 1: a state set S may lead to the creation of further state sets if these are not already present in the state register.
2. with state *add delta* which reflects line 17 of the serial algorithm and
3. with state *make final* which corresponds to line 9.

All states except s_0 work in principle in parallel, but they share three common resources: the state register, the state counter and the emerging DFA. Access to these resources must be synchronised by using appropriate locking mechanisms.

4 Implementation

The algorithm of Section 3.2 is implemented on the basis of the *flow graph* construct in Intel's *ThreadBuildingBlocks* C++ library (TBB, [8]). TBB defines a number of different graph nodes classes like `broadcast_node`, `function_node` and `multifunction_node`, which can be connected to each other by data flow edges.

Unlike instances of `function_node`, which are required to always compute a result, instances of `multifunction_node` are connected to other flow graph nodes by a tuple of channels, to which output actions are send. This is exactly what is required by state *process state set* in Figure 3, since a NFA state set currently processed may not lead to further new state sets.

The serial and parallel version of the determinisation algorithm are basically based on the same data structures. State sets of NFAs were implemented as sorted vectors. To allow an efficient test for equality of state sets and to speed up look-up in the state register, each state set also stores a permanent hash value. The δ-function of the class representing serial DFAs is based on a STL `hash_map`, while the one of the concurrent DFA uses TBB's `concurrent_hash_map`, a map data structure where the keys can be individually locked. The state registers for the serial and parallel algorithms are implemented in a similar fashion. Figure 4 states some of the relevant definitions of the parallel algorithm in C++. `ParallelDeterminizerBody`, `AddDeltaBody`, and `MakeFinalBody` are classes which implement the actions executed by the three parallel nodes of Figure 3.

5 Experiments

For the experiments, we choose two different types of NFAs:

1. Acyclic NFAs compiled from word lists and
2. Cyclic NFAs exhibiting the worst case along the lines of Figure 1 with various number of states and alphabet sizes.

```
typedef int                                                  STATE;
typedef StateSet<STATE>                                      NFAStateSet;
typedef std::pair<NFAStateSet,STATE>                         NFAStateSetToState;
typedef tbb::concurrent_hash_map<NFAStateSet,STATE>          StateRegister;
typedef std::tuple<NFAStateSetToDFAState,AddDelta,MakeFinal> NFAStateSetToDFAStateTuple;
typedef tbb::flow::multifunction_node<NFAStateSetToState,NFAStateSetToStateTuple> ParDetWorkerNode;
typedef tbb::flow::function_node<AddDelta>                   AddDeltaNode;
typedef tbb::flow::function_node<MakeFinal>                  MakeFinalNode;

NFA nfa;
ConcurrentDFA dfa;
StateRegister state_register;
tbb::atomic<STATE> state_counter;
unsigned num_workers = tbb::flow::unlimited;

tbb::flow::graph g;
tbb::flow::broadcast_node<NFAStateSetToState> pardet_root_node(g);
ParallelDeterminizerBody pardet_worker_body(nfa,state_register,state_counter);
ParDetWorkerNode pardet_node(g, num_workers, pardet_worker_body);
AddDeltaNode add_transition_node(g, num_workers, AddDeltaBody(dfa));
MakeFinalNode make_final_node(g, num_workers, MakeFinalBody(dfa));
tbb::flow::make_edge(pardet_root_node, pardet_node);
tbb::flow::make_edge(tbb::flow::output_port<0>(pardet_node), pardet_node);
tbb::flow::make_edge(tbb::flow::output_port<1>(pardet_node), add_delta_node);
tbb::flow::make_edge(tbb::flow::output_port<2>(pardet_node), make_final_node);
```

Figure 4. C++ definitions of the parallel algorithm based on Intel's TBB framework

The acyclic NFAs derived from two different English words lists are maximally non-deterministic, that is, each word inserted into the NFA constitutes a separate chain from the start state to a distinct final state.

Table 1 summarises the different test automata.

NFA	$\lvert\mathbf{\Sigma}\rvert$	$\lvert\mathbf{Q_{nd}}\rvert$	$\lvert\delta_{\mathbf{nd}}\rvert$	$\lvert\mathbf{F_{nd}}\rvert$	$\lvert\mathbf{Q_{d}}\rvert$	$\lvert\delta_{\mathbf{d}}\rvert$	$\lvert\mathbf{F_{d}}\rvert$
NFA$_{dict1}$	56	681,719	681,718	49,999	271,194	271,193	49,999
NFA$_{dict2}$	45	994,676	994,675	128,972	270,411	270,410	128,972
NFA$_{r(k),2}$, $k \in [10 \dots 22]$	2	$k+1$	$2(k+1)+1$	1	2^{k+1}	$2 \cdot 2^{k+1}$	$\frac{2^{k+1}}{2}$
NFA$_{r(k),10}$, $k \in [10 \dots 19]$	10	$k+1$	$10(k+1)+1$	1	2^{k+1}	$10 \cdot 2^{k+1}$	$\frac{2^{k+1}}{2}$
NFA$_{r(k),100}$, $k \in [10 \dots 16]$	100	$k+1$	$100(k+1)+1$	1	2^{k+1}	$100 \cdot 2^{k+1}$	$\frac{2^{k+1}}{2}$

Table 1. Input NFA for the determinisation algorithms

All subsequent experiments were run on a Linux machine with 2 Intel-XEON 64bit-2.93 GHz 4-core-CPUs. Hyperthreading was turned on. In hyperthreaded architectures, each physical core is supplemented by a virtual core which takes over control if the physical one is currently stalled because it is waiting for CPU cache data etc. Virtual cores duplicate only certain sections of their physical counterpart, mainly those holding the current thread's state, but not the main computing resources.

Let us now turn to the experiments. In Figure 5, we compare the serial and the parallel version of the determinisation algorithm for NFA$_{r(k),2}$ on a logarithmic time scale. Unsuprisingly, the processing time for both versions grows exponentially with the number k of disjunctions[3] in the NFA. Starting with $2^{11+1} \approx 4,000$ DFA states, the parallel algorithm outperforms the serial one, with $k = 13$, it is already more than twice as fast. With bigger ks, the processing time of the parallel version converges at approximately one-third of the serial one.

The advantage of the parallel algorithm becomes even better when the alphabet size is increased. Figure 6 compares the serial and parallel determinisation of NFA$_{r(k)}$ with $|\Sigma| = 10$ and $|\Sigma| = 100$, respecitively. For an alphabet size of 10, the parallel algorithm is, depending on k, approximately 2 to 3.5 times faster than the serial one. For $|\Sigma| = 100$, the ratio is between 3.7 to 1 for $k = 10$ and 4.7 to 1 for $k = 16$.

[3] Also known as *alternations*.

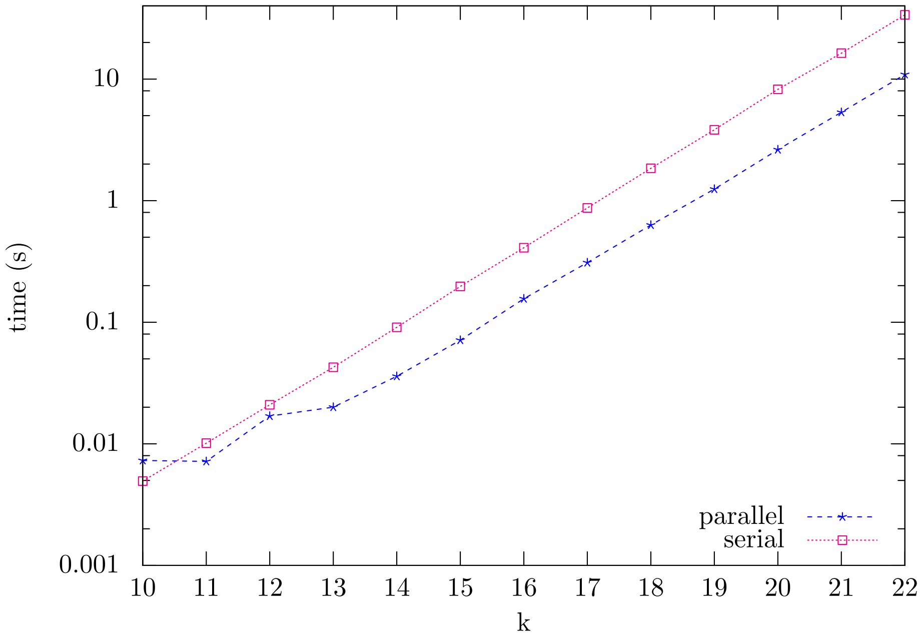

Figure 5. Serial vs. parallel determinisation of $\mathrm{NFA}_{r(k),2}$

An explanation for the speedup for bigger alphabet sizes could be, that the number of DFA states depends only on k and thus the number of pairs $\langle S, q\rangle$ forwarded on the looping channel in Figure 3 is independent of the alphabet size. Furthermore, the state register shared between the parallel determinisation workers – even when queried $|\Sigma|$ times for each state set – doesn't seem to slow down processing very much.

To assess the amount of the contribution of the other shared resource – the DFA under construction – we ran a further experiment where we turned off the channels to the states *add delta* and *make final* in Figure 3 and made a similar move in the serial version. The results for the $\mathrm{NFA}_{r(k),100}$ are shown in Figure 7.

Figure 7 shows that for the serial case whether DFA construction is turned on or off makes almost no difference with respect to processing time. But the situation is different for the parallel algorithm where the DFA construction contributes with more than 40% to the overall processing time.

Since both serial and concurrent DFA implementations rely on efficient hash maps, the difference must be explained with locking issues and the administrative overhead for implementing micro locks.

In our second-last experiment, we compare serial and parallel determinisation applied to acyclic NFAs, namely, the NFAs derived from the two word lists. Table 2 summarises the results.

NFA	serial	parallel
$dict_1$	0.465 s	0.197 s
$dict_2$	0.550 s	0.230 s

Table 2. Serial and parallel determinisation of the dictionary NFAs

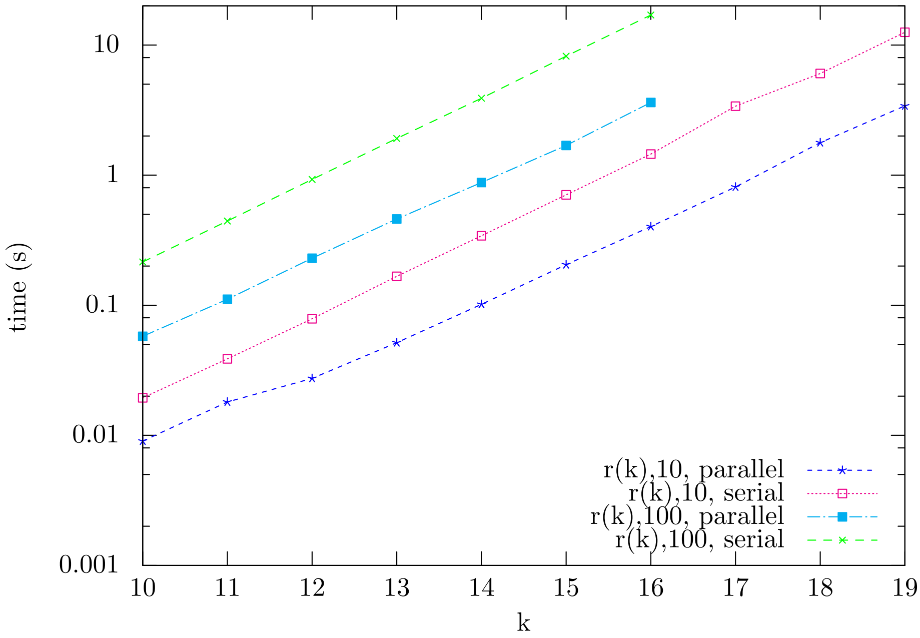

Figure 6. Serial vs. parallel determinisation of NFA$_{r(k),10}$ and NFA$_{r(k),100}$, resp.

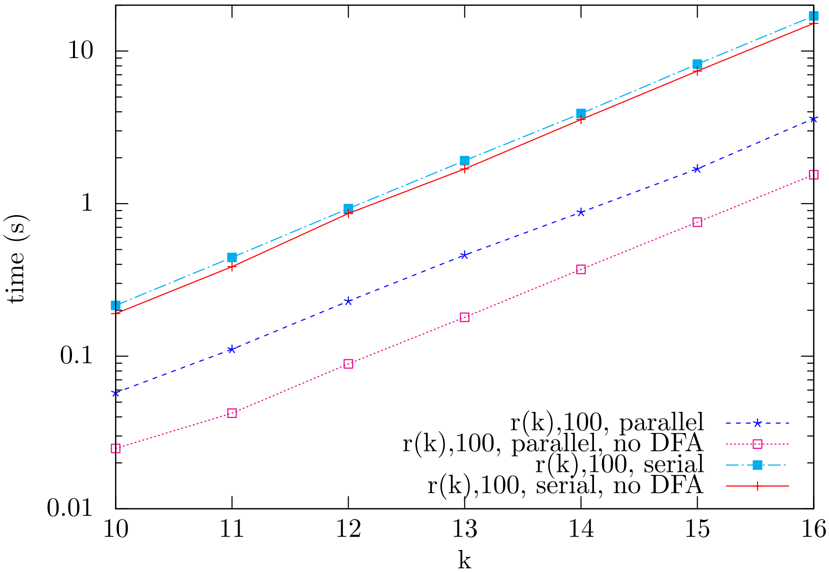

Figure 7. Serial and parallel determinisation of NFA$_{r(k),100}$, with DFA construction turned on and off

The parallel version exhibits a speed-up of a factor of approximately 2.4 compared to the serial algorithm. Even though the alphabet sizes are bigger, this is less than the speed-up in the cyclic case with an alphabet of size 2.

But, (serial) determinisation of acyclic NFA is very efficient anyway, since it is linear in the size of the NFA, so parallelising the algorithm is normally not worth the effort.

Intel's *ThreadBuildingBlocks* framework allows the control of the amount of parallelism by specifying the number of flow graph node copies concurrently active. A value of *unlimited* means that the framework chooses an optimal amount of concurrency.

Figure 8 shows the dependencies between the number of workers and the time consumed for two NFA ($r(17), 2$ and $dict_2$).

The relative plateau in both graphs for 8 to 16 workers could perhaps be explained with Intel's *hyperthreading* feature. Since the non-virtual cores are not idle when the parallel algorithm is executed, the virtual ones cannot take over control and thus do not contribute at all. Also apparent from the graphs is, that the parallel algorithm performs best if TBB's scheduler is allowed to control the amount of parallelism.

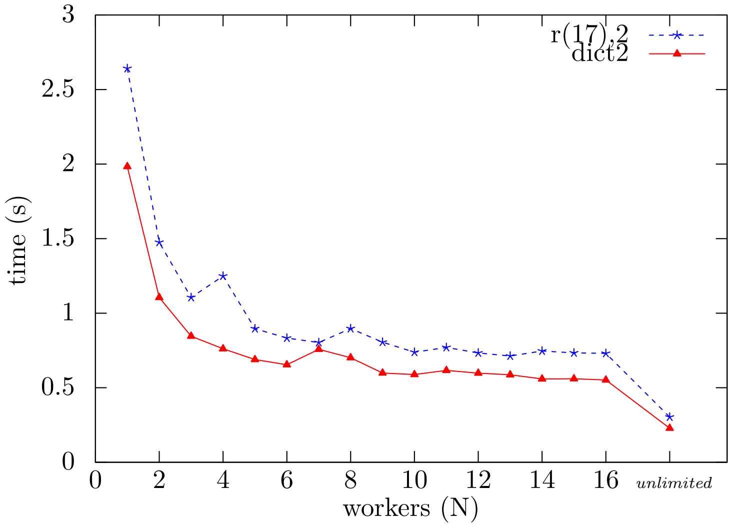

Figure 8. Dependency between the processing time and the number of workers for NFAs $r(17), 2$ and $dict_2$

6 Conclusion and further work

In the preceding sections, we developed an efficient parallel determinisation algorithm based on Kahn process networks. Experiments showed that the algorithm performed particularly well in cases of highly-cyclic result DFAs over realistically sized alphabets.

The worst-case pattern $\Sigma^* a(a+b)^k$ we choose for the cyclic test automata is not as artifical as it looks at first glance. For example, compiling replacement rules $\alpha \rightarrow \beta$ [7] relies on a *does-not-contain operator* $\overline{\Sigma^* \cdot \alpha \cdot \Sigma^*}$ to achieve robust behaviour by identity-mapping all strings to themselves which do not contain an instance of α. Since the standard complementation operation depends on a deterministic DFA for $\Sigma^* \cdot \alpha \cdot \Sigma^*$, choosing $a(a+b)^k$ for α creates the worst case.

In further work, we try to improve the algorithm in the following ways:

- Examine and profile the algorithm to reduce the number of locking situations,
- apply randomisation techniques to further reduce locking,

– make use of graph theoretic notions to divide the determinisation problem into largely independent subproblems which can be solved without making much use of shared resources, and
– study a redundant work approach where each parallel determinisation worker may process a limited number of state sets already processed by other workers to increase the relative independence of the workers from the shared resources.

References

1. A. V. Aho: *Algorithms for Finding Patterns in Strings*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. A, North-Holland, 1990, pp. 257–300.
2. M. Geilen and T. Basten: *Requirements on the Execution of Kahn Process Networks*, in Proc. of the 12th European Symposium on Programming, ESOP 2003, Springer Verlag, 2003, pp. 319–334.
3. R. Glabbeek and B. Ploeger: *Five Determinisation Algorithms*, in Proceedings of the 13th International Conference on Implementation and Applications of Automata, CIAA '08, Berlin, Heidelberg, 2008, Springer-Verlag, pp. 161–170.
4. J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.
5. G. Kahn: *The Semantics of a Simple Language for Parallel Programming.* Information Processing, 1974, pp. 471–475.
6. R. M. Kaplan and M. Kay: *Regular Models of Phonological Rule Systems.* Computational Linguistics, 20(3) 1994, pp. 331–378.
7. L. Karttunen: *The Replace Operator*, in ACL, 1995, pp. 16–23.
8. J. Reinders: *Intel Threading Building Blocks. Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly, 2007.