Non-blocking Parallel Subset Construction on Shared-memory Multicore Architectures

Hyewon Choi **Bernd Burgstaller**

Department of Computer Science
Yonsei University
Seoul, Korea
xizsmin@yonsei.ac.kr, bburg@cs.yonsei.ac.kr

Abstract

We discuss ways to effectively parallelize the subset construction algorithm, which is used to convert non-deterministic finite automata (NFAs) to deterministic finite automata (DFAs). This conversion is at the heart of string pattern matching based on regular expressions and thus has many applications in text processing, compilers, scripting languages and web browsers, security and more recently also with DNA sequence analysis. We discuss sources of parallelism in the sequential algorithm and their profitability on shared-memory multicore architectures. Our NFA and DFA data-structures are designed to improve scalability and keep communication and synchronization overhead to a minimum. We present three different ways for synchronization; the performance of our non-blocking synchronization based on a compare-and-swap (CAS) primitive compares favorably to a lock-based approach. We consider structural NFA properties and their relationship to scalability on highly-parallel multicore architectures. We demonstrate the efficiency of our parallel subset construction algorithm through several benchmarks run on a 4-CPU (40 cores) node of the Intel Manycore Testing Lab. Achieved speedups are up to a factor of 32x with 40 cores.

Keywords: Subset construction, shared-memory multicore architectures, non-blocking synchronization, concurrent data-structures

1 Introduction

The subset construction algorithm converts an NFA to the corresponding DFA. Subset construction is frequently applied with string pattern matching based on regular expressions. A standard technique to match a regular expression on an input text is to convert the regular expression to an NFA using Thompson's construction, perform subset construction to derive a DFA, and minimize the DFA. The DFA is then run on the input text. If the DFA accepts its input, the input is known to be matched by the regular expression. This method has been described by Aho et al. (1986).

With multicores becoming the predominant computer architecture, it is desirable to parallelize the subset construction algorithm. Although algorithms for DFA state minimization and DFA matching (discussed in Section 7) exist, to the best of our knowledge this is the first attempt to parallelize subset construction.

We parallelize subset construction for shared-memory multicore architectures. We analyze the different potential sources of parallelism contained in the sequential subset construction algorithm and compare their profitabilities. We devise an efficient parallel version of the subset construction algorithm, which guarantees load-balance and provides good scalability. We state the data-structures devised for the parallelization, which help improve the efficiency of the operations performed on shared-memory multicore architectures. To ensure correctness of the algorithm while not compromising on parallelism, we developed efficient synchronization methods. The performance of our non-blocking synchronization based on a CAS primitive compares favorably to a lock-based approach. We consider structural NFA properties and their relationship to scalability on highly-parallel multicore architectures. We demonstrate the efficiency of our parallel subset construction algorithm through several benchmarks run on a 4-CPU (40 cores) node of the Intel Manycore Testing Lab (accessed Aug. 2012).

This paper is organized as follows. In Section 2, we present sequential subset construction and related background material. In Section 3, we identify potential sources of parallelism and their profitability in the sequential subset construction algorithm. In Section 4 we introduce the algorithm's data-structures for supporting scalability on shared-memory multicore architectures. Section 5 discusses synchronization and the algorithm's termination condition. Section 6 provides our experimental results. We discuss the related work in Section 7 and draw our conclusions in Section 8.

2 Background

Let Σ denote a finite alphabet of characters and Σ^* denote the set of all strings over Σ. Cardinality $|\Sigma|$ denotes the number of characters in Σ. A language over Σ is any subset of Σ^*. The symbol $\emptyset$ denotes

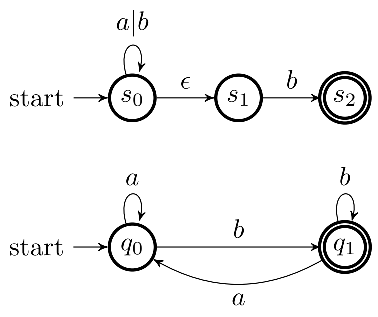


Figure 1: An example NFA (above), and its DFA converted through subset construction (below). DFA state q_0 represents NFA states s_0 and s_1, while DFA state q_1 represents NFA states s_0, s_1 and s_2.

the empty language and the symbol ϵ denotes the null string. A finite automaton A is specified by a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $q_0 \in Q$ is the start state and $F \subseteq Q$ is a set of final states. We define A to be a DFA if δ is a transition function of $Q \times \Sigma \to Q$ and $\delta(q, a)$ is a singleton set for any $q \in Q$ and $a \in \Sigma$. Otherwise, it is classified as an NFA. Let $|Q|$ be the number of states in Q. By the *density* of an automaton, we denote the ratio of the number of transitions in a given NFA to the number of transitions of a fully connected DFA of the same number of states and symbols (Leslie 1995). For a comprehensive background on automata theory we refer to (Hopcroft & Ullman 1979, Wood 1987).

2.1 Sequential Subset Construction

Algorithm 1 depicts the sequential subset construction algorithm from Hopcroft & Ullman (1979). Figure 1 depicts a sample NFA and the DFA computed by the subset construction algorithm. To begin with, we get the start state derived from s_0 of the NFA. i.e., we take $S_0 = \epsilon\text{-}closure(s_0)$ first. Then we compute $\epsilon\text{-}closure(Move(s_0, \sigma))$ for each $\sigma \in \Sigma$. If we get several states at once as a result of the computation, we make a set with them and treat it as a single DFA state. For a single DFA state T, we find all the states which can be reached by each $\sigma \in \Sigma$ from all the elements of T. Then we compute ϵ-*closure*s for the results, and this creates a new DFA state. If this DFA state has not been appeared before, we add it to the DFA table. This process is iterated until no more DFA states are added.

To determine the time complexity for this algorithm, we consider the complexity of computing $\epsilon\text{-}closure(s_0)$ first. Because s_0 may have $(|Q| - 1)$ ϵ-*transitions*, we conclude that computing $\epsilon\text{-}closure(s_0)$ takes $\mathcal{O}(|Q| - 1)$. Then the process without set equality test is computed with an

$$\mathcal{O}(|\Sigma| \times |Q| \times (|Q| - 1) \times (2^{|Q|} - 1))$$

time complexity. Now what we have to do is finding the factors which can be parallelized to reduce the complexity. We discuss the details in the following section.

Algorithm 1: Sequential Subset Construction

```
   Input  : NFA
   Output: DFA states Dstates,
           DFA transition function $\delta$
1  $Dstates \leftarrow \{\}$;
2  Add $\epsilon$-closure($s_0$) as an unmarked DFA
   state to Dstates;
3  while there is an unmarked state T in
   Dstates do
4      mark T;
5      for each $\sigma \in \Sigma$ do
6          $U \leftarrow \epsilon$-closure(Move(T, $\sigma$));
7          if $U \notin Dstates$ then
8              add T as an unmarked DFA state
               to Dstates;
9          $\delta[T, \sigma] \leftarrow U$;
```

3 Potential Sources of Parallelism With Sequential Subset Construction

In this section we identify sources of parallelism and their profitability with the sequential subset construction algorithm.

Source 1: The first opportunity for parallelization is on symbols $\sigma \in \Sigma$ (line 5 in Algorithm 1). Hence, this method is a task parallelization. After checking if there exists an unmarked state T in *Dstates*, what we have to do is taking its $\epsilon\text{-}closure(Move(T, \sigma))$ for each $\sigma \in \Sigma$, which is described in line 6 to line 9 of Algorithm 1. With the sequential version, this part must be computed $|\Sigma|$ times, i.e., for each $\sigma \in \Sigma$. However, because there is no dependency upon symbols, we may partition the symbols in Σ among the available processors to be processed in parallel. Thus, the time complexity becomes

$$\mathcal{O}(\left\lceil \frac{|\Sigma|}{p} \right\rceil \times |Q| \times (|Q| - 1) \times (2^{|Q|} - 1)).$$

Source 2: The second opportunity is parallelizing the outer while-loop (line 3) of the sequential algorithm. Until the algorithm terminates, *Dstates* has at least one DFA state at the beginning of every iteration. Thus, for every iteration we partition the states in *Dstates* among processors to parallelize the algorithm. This requires each processor to deal with the steps from line 4 to 9. A step for counting the number of unmarked states in *Dstates* must be added right after entering the while loop. The result is used for determining the number of to-be-processed DFA states for each processor. For marking that follows after distribution, we should allow all processors to access *Dstates*. Time complexity for this algorithm becomes

$$\mathcal{O}(|\Sigma| \times |Q| \times (|Q|-1) \times \left\lceil \frac{2^{|Q|}-1}{p} \right\rceil).$$

Source 3: To exploit the final source of parallelism, we split a DFA state across its contained NFA states right after marking. The processors take the work described in line 5 through 9 in Algorithm 1. It should be noted that now processors work with the elements of a single DFA state (i.e., NFA states), not several DFA states. The granularity of concurrent computations is thus smaller than with Source 2. The partitioning of NFA states of an unmarked DFA state is conducted after we mark the unmarked DFA state. The work from line 5 to 8 is done in exactly the same way than the sequential version. Adding the result to a DFA state however, can be done in two different ways: we may let each worker thread add its NFA states to the DFA state, or have a dedicated master collect NFA states found by the workers and add them to the DFA state. Now it follows that the time complexity becomes

$$\mathcal{O}(|\Sigma| \times \frac{|Q|}{p} \times (|Q|-1) \times (2^{|Q|}-1)).$$

3.1 Profitability of Each Parallelism Source

Unfortunately, not all the suggested methods are equally profitable. Throughout our evaluated benchmarks (see also Section 6), we have found that the second and third opportunities are less effective than the first one (parallelizing on symbols). This phenomenon is caused by the following drawbacks of those two methods. To get a noticeable performance improvement by parallelizing the *Dstates*, it should be guaranteed that the number of DFA states at a certain moment is large enough. However, this number changes dynamically and cannot be known in advance. Even worse, larger automata sizes do not guarantee a larger number of *Dstates* at each point in time. Thus, even for large NFAs, a substantial performance gain cannot be expected: this observation implies that this algorithm would not be very useful in real situations.

In the case of parallelizing the NFA states of a DFA state, we need to collect the results from all the worker threads to construct a complete DFA state. For this work, the amount of synchronization is too high (e.g., from using barrier-synchronization) which eventually caused severe performance degradation with our evaluations.

On the contrary, parallelizing the algorithm on symbols is advantageous for load balance: because the number of symbols is constant and known a-priori, a static partitioning of work among worker threads can be computed.

3	1	4	5			...
2	0	3				...
4	0	1	2	4	8	...

Table 1: Thread-local store of DFA states. The union of all workers' DFA states constitutes the *Dstates* set (see line 1 of Algorithm 1).

4 Data Structures for Parallel Subset Construction

We attempted to minimize overhead due to communication and synchronization between worker threads. In general, we kept data thread-local as much as possible, except for the *Dstates* set, which must be updated by all worker threads to collect DFA states.

Each worker thread maintains a thread-local array of DFA states as depicted in Table 1. The union over all thread-local DFA states constitutes the elements of the *Dstates* set. Similar to the sequential, deterministic state ADT by Leslie (1995), we store DFA states as linear arrays of NFA state members. The first array element of a DFA state contains the number of NFA states contained in a DFA state. Subsequent array elements represent NFA states. This representation facilitates efficient comparison of two DFA states, which is required for the set membership test (line 7 of Algorithm 1). The comparison of two DFA states is depicted in Algorithm 2.

To create new DFA states, each worker thread maintains a one-element DFA state scratch-space. Potentially new DFA states are computed by the ϵ-$closure(Move(T, \sigma))$-term in line 6 of Algorithm 1. NFA states are stored in the order of their appearance during this step. Before performing the set membership test in line 7, we sort the potentially new DFA state's NFA states in ascending order and copy the DFA state to the worker's thread-local DFA state array.

The *Dstates* set is a shared data-structure represented as an array of pointers to thread-local DFA states. Adding a DFA state to the *Dstates* set reduces to a single pointer update, i.e., the first (lowest-index) empty entry of *Dstates* is set to point to the new DFA state. Pointers are padded up to the CPU's cache-line size to avoid false sharing of cache-lines (see, e.g., (Lin & Snyder 2008)) that would otherwise occur if worker threads update adjacent array elements.

Algorithm 2: DFA State Equality Test

```
    Input  : 1D-array $Dstate1$, $Dstate2$
    Output: True (if identical), False otherwise
1   for $i = 0$ to $Dstate1[0]$ do
2       if $Dstate1[i] != Dstate2[i]$ then
3           return False;
4   return True;
```

The *Dstates* set needs to maintain the following properties:

- The whole *Dstates* array is accessible by all worker threads.
- For the DFA states pointed by an entry of *Dstates*, duplication (entering a duplicate DFA state into *Dstates*) is not allowed.

After computing a potentially new DFA state in its thread-local store, a worker needs to determine whether the DFA state is allowed to be added to *Dstates*. As the first step of this decision, the thread performs the set membership test. Once it confirms that no identical DFA state has been added yet, it updates the next empty slot in *Dstates*.

Because *Dstates* allows access from any worker at any time, the set membership test followed by the subsequent pointer update is subject to potential race conditions. We will discuss three synchronization methods in the following section.

5 Synchronization and Termination

We are facing two synchronization problems: how to avoid race conditions with workers entering DFA states in the *Dstates* array, and when to terminate workers.

5.1 Synchronizing *Dstates* Updates

The first synchronization issue is due to the adding of DFA states to *Dstates.* We suggest three ways to synchronize access to the *Dstates* array: using a coarse-grained lock, a fine-grained lock, and making the algorithm non-blocking by guarding access to the *Dstates* array through a CAS instruction. CAS compares a data item in memory with a previous value A and replaces it with a newly provided value B, only iff the data item has not been updated by another thread. I.e., the data item's value in memory is identical with A (Herlihy & Shavit 2008). We use mutexes from the Pthread library for all lock-based synchronization and GCC's intrinsics for CAS operations.

Using a coarse-grained lock is a naive way to protect *Dstates*. Once a worker finds a new DFA state, it tries to acquire the mutex which is protecting *Dstates*. If successful, it performs the set membership test to confirm that its DFA state is not a duplicate of an existing DFA state in *Dstates*. During the whole set membership test, no other worker is allowed to update *Dstates*. After the set membership test, no matter the addition has been allowed or rejected, the worker thread releases the mutex.

A fine-grained lock, compared to the coarse-grained one, helps reducing the waiting time for worker threads waiting for acquiring the *Dstates* lock. The major factor which distinguishes this method from the previous one is that this time, during the set membership test, the lock for *Dstates* needs not be held by a worker thread. Instead, right after the worker thread finishes the set membership test and determines that no identical state exists in *Dstates* so far, it acquires the *Dstates* lock, and begins the set membership test again, from the point where it has stopped the test before locking *Dstates* until the newest element in *Dstates.* This second set membership test is for compensating a potential race condition that workers face after finishing the set membership test and before locking *Dstates*. Within this short time period, another thread might add a DFA state to *Dstates*. If this happens, the empty entry found by the previous thread is taken and the set membership test must be continued.

The final synchronization method does not lock *Dstates*: instead, it makes *Dstates* a concurrent data structure (Shavit 2011) by employing a CAS instruction. First, this method goes through the set membership test without locking *Dstates* as done when using a fine-grained lock. This time, however, we do not protect *Dstates* even after the worker thread has finished the set membership test. Instead, after finding the first empty entry in the *Dstates* array, it attempts to add the DFA state by executing a CAS instruction.

Algorithm 3: Access to *Dstates* guarded by CAS instruction

```
1 while !CAS (Dstates entry[i], NULL, DFA state) do
2     if DFA State Equality Test (Dstates entry[i], DFA state) then
3         return False;
4     ++i;
5 // At this point:
6 // Dstates entry[i] = DFA state
7 return True;
```

Algorithm 3 shows the basic concept of the non-blocking implementation using CAS. This form of execution confirms that only if the *Dstates* entry is empty, then the DFA state will be added there. It should be noted that after a failed CAS instruction, the set membership test must be continued, similar to our fine-grained locking method. Because we never delete entries in the *Dstates* set, the ABA-problem (see, e.g., (Herlihy & Shavit 2008)) does not apply.

5.2 Terminating Condition

The second problem is to decide when to make worker threads terminate. As stated in Algorithm 1, subset construction is supposed to terminate when no more new DFA states are added to *Dstates*. This implies that we need to enable the worker threads to determine when it is guaranteed that no new DFA state will appear anymore. If a worker thread cannot find any new DFA state, it needs to observe other threads if any of them is still processing DFA states. If it turns out that all workers have processed all DFA states, then it is safe for a worker to terminate.

To maintain the status of each worker thread wrt. processing of DFA states in *Dstates*, we use one status array per worker. In a worker's status array L, a worker keeps track how far it progressed in processing the DFA states in *Dstates*. The nth entry of status array L represents a worker's status wrt. the nth DFA state in *Dstates*.

Entries in the status array can have three vales: *Not_accessed*, *Processing* and *Finished*. *Not_accessed* denotes the status that the worker thread has not begun to process the DFA state: Following the expression used in Algorithm 1, this DFA state has not been marked by this worker thread yet. Right after a worker begins processing within the while loop stated in line 3 of Algorithm 1 with the nth DFA state, it marks *L[n]* as *Processing*: if it finishes the work, it sets *L[n]* to *Finished*.

Now let us assume that a worker thread finds there is no new DFA state in *Dstates* after processing the nth DFA state, i.e., the *(n+1)*th entry in the *Dstates* array is empty. Then it observes the *L[n]* status values of the other workers: if any of the workers has not set its *L[n]* status value as *finished* yet, then the idle worker needs to wait for a new DFA state to appear in the *n+1* slot of *Dstates*. However, once all workers have set *L[n]* to *finished* and no DFA state appeared in slot *n+1* of *Dstates*, it is safe for all workers to terminate. It should be noted that workers write the *Finished* flag to the status array *after* they have entered a new DFA state to the *Dstates* array.

6 Experimental Results

We demonstrate that our implementation of the sequential subset construction shows reasonable performance, compared to related tools. We chose the Grail tool by Raymond & Wood (1995) as our yardstick. Grail is a formal language toolbox which already provides a sequential version of the subset construction algorithm.

To determine the scalability of our parallel subset construction algorithm, we have conducted experiments on a 4-CPU (40 cores) shared memory node of the Intel Manycore Testing Lab. The POSIX thread library (see, e.g., Butenhof (1997)) has been used for worker threads and mutexes. We compared the scalability of two groups of NFAs that differ in their density, to show that density affects scalability.

Finally, we conducted experiments for all three *Dstates* synchronization mechanisms: using a coarse-grained lock, a fine-grained lock, and the non-blocking method. We compare the efficiency of each method and we discuss the obtained results.

All execution times have been determined by reading the elapsed clock ticks from the x86 timestamp counter register (Paoloni 2010).

6.1 Performance Comparison with Grail

To confirm that our data-structures are efficient even for the sequential case, we compared our subset construction implementation to Grail. We used version 3.0 of the Grail code from the Grail+ Project Web Site (retrieved Aug. 2012), and we revised it to compile with g++ version 4.1.2. The performance comparison of a sequential version of our subset construction implementation with Grail is depicted in Figure 2.

The y axis represents relative time spent: we set the spent time for computing a 20-symbol automaton as 1. Thus, if the time spent for computing automata with x symbols takes five times longer than that of the 20-symbol automaton, the spent time for computing x is noted as 5. The number of states has been fixed to 20. This experiment has been conducted on an Intel Xeon E5405 CPU running Linux. As the NFA size increases, the performance gap increases remarkably: this implies that our NFA and DFA representations are efficient even with large NFAs.

6.2 Automata Density vs. Scalability

For this experiment, we have classified the NFA samples based on their *density*. In particular, we considered two groups of density 0.3 and 0.4, respectively. We observed the speedups gained with both groups. As depicted in Figure 3, there is a clear gap in scalability between those two groups. We conjectured that this is related to properties of the DFAs converted from NFAs through our algorithm. In particular, we investigated the number and sizes of DFA states, as shown in Figure 5.

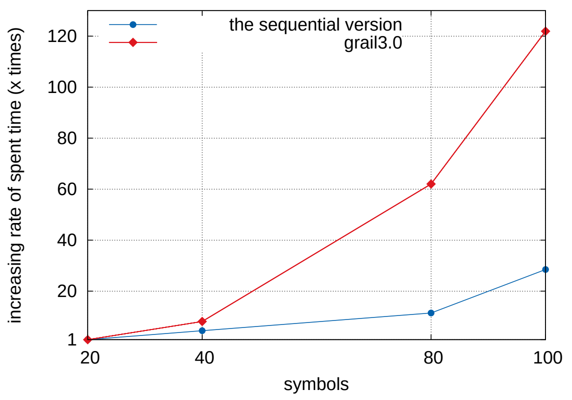
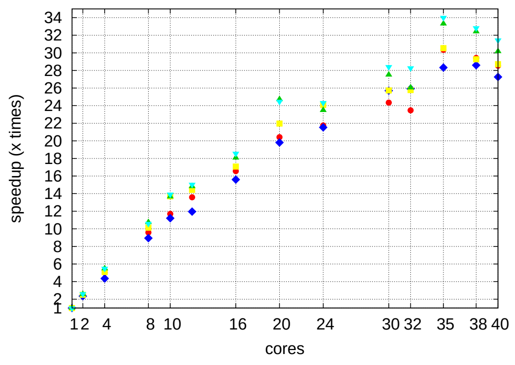
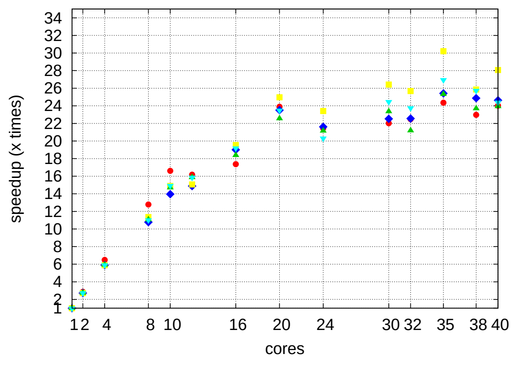


Figure 2: Performance comparison of proposed subset construction using custom NFA and DFA representations vs. the Grail tool.



(a) NFAs with density 0.3



(b) NFAs with density 0.4

Figure 3: Scalability test with NFAs of density 0.3 and 0.4. NFAs from both groups have 20 states and 100 symbols.

To generate NFA samples, we have used an NFA random generator from Almeida et al. (2007). In this program, an automaton's density is used as a factor which determines the number of transitions. Let density be denoted by d, the number of states denoted by n, k denote the number of symbols and t denote the temporary number of transitions. The NFA random generator computes the temporary number of transitions as

$$t = dn^2k - (n-1).$$

The final number of transitions is determined by adding a small random factor to t. This formula suggests that if the number of states and symbols are not changed, as the density increases, the number of transitions also increases. As a result, each state gets more reachable states on average. From Algorithm 1, we can infer that if such an automaton undergoes subset construction, the average DFA state size would increase. Now we will show that this increment leads to a decrease in the number of DFA states. For an NFA, let Q denote the set of states. Then DFA states are subsets of Q. Let $n = |Q|$ and m the average DFA state size. Then we may approximate the number of possible DFA states by $\binom{n}{m}$.

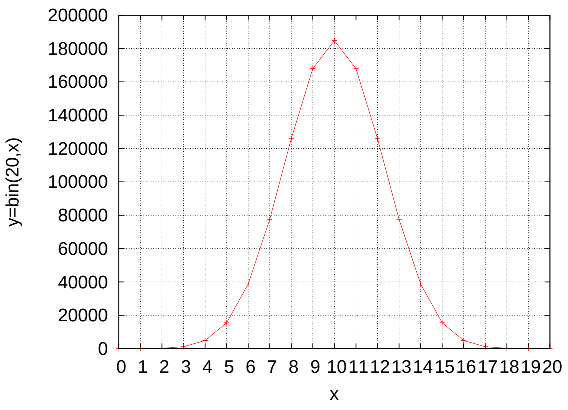


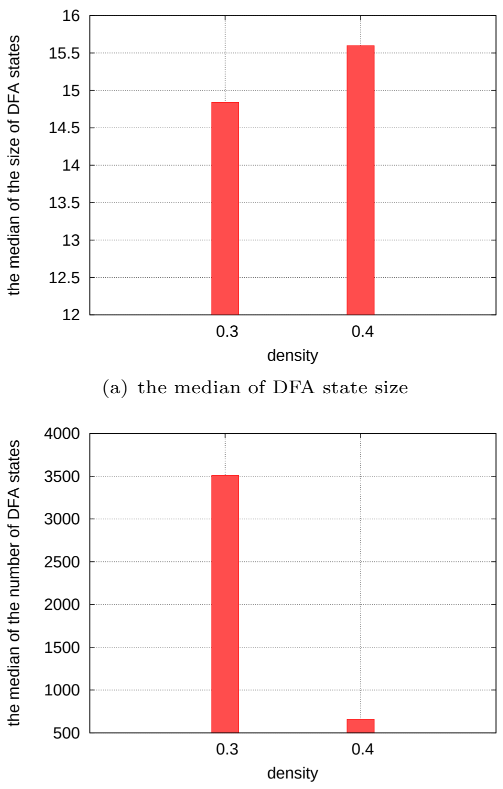
Figure 4: Number of possible DFA states $y = \binom{20}{x}$ for a 20-state NFA and $0 \leq x \leq 20$ NFA states per DFA state.

Figure 4 shows this tendency for n set to 20, and m ranging from 0 to n. As we may consider the x axis as the average size of the DFA states, we can infer that within a certain range of the DFA size, the number of DFA states would increase as the size increases, while the tendency could be reversed in some other range.

The DFAs from our experiment clearly follow this trend, as Figure 5 shows. To mitigate the effect of a few outliers, we have taken the median of the sample data instead of an arithmetic mean. The sizes and the number of DFA states collected from our NFA samples clearly match the approach suggested in Figure 4, which claims that between the average DFA state size of 14 to 16, the number of DFA states will decrease. This eventually reduces the amount of computation in the subset construction, which negatively affects the scalability as observed in Figure 3: compared to Figure 3(a), Figure 3(b) shows a clear performance degradation.

6.3 Comparing Scalability of the Three Synchronizing Methods

Figure 6 compares the scalability test results from three different versions of the parallelized algorithms: using a coarse-grained lock, a fine-grained



(a) the median of DFA state size

(b) the median of number of DFA states

Figure 5: Median over all DFA sizes and over the number of DFA states of the sample automata.

one, and the non-blocking mechanism. The graph clearly shows that using a coarse-grained lock causes severe serialization as the number of cores increases. With the coarse-grained lock, the set membership test is performed while holding the *Dstates* lock. Thus, while a worker thread is doing the set membership test for a newly found DFA state, other threads who want to add their DFA states cannot proceed, which serializes the algorithm.

On the contrary, the parallelized algorithms which used a fine-grained lock and the non-blocking mechanism show good scalability. Both algorithms show a similar tendency, because their strategies to protect *Dstates* are fundamentally similar to each other: they both let each worker thread perform the set membership test without locking *Dstates*, and once a worker finds an empty entry to add its DFA state, it begins the second test to confirm that it is safe to do the addition. What we need to focus here is that the most time-consuming part is the first set membership test, not the second one. Thus, this part affects on the whole processing time.

Both algorithms, i.e., using a fine-grained lock and the non-blocking mechanism, show an important phenomenon: Around 30 cores, the scalability becomes imperfect: this is due to the non-parallelized part of the algorithm, such as computing an epsilon-closure, the set equality test performed as a single step of the set membership test, etc. As the number of cores increases, time spent for the parallelized part decreases, but the non-parallelized part remains. In our experiment, this

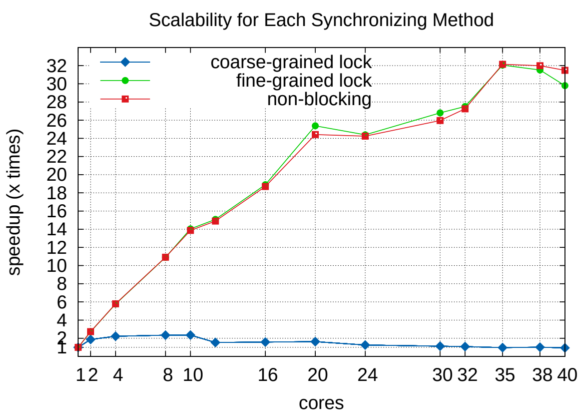


Figure 6: Scalability test result of the parallelized subset construction, with three synchronizing methods: using a coarse-grained lock, a fine grained lock and our non-blocking mechanism

limitation appears as the number of cores reaches around 30.

7 Related Work

Tewari et al. (2002) devised a parallel algorithm for DFA minimization. Approaches to parallelize DFA matching have been contributed by Luchaup et al. (2011), Wang et al. (2010), Holub & Štekr (2009), Jones et al. (2009), Luchaup et al. (2009), Scarpazza et al. (2007), Misra (2003), Hillis & Steele (1986), Ladner & Fischer (1980) and Ko et al. (2011). However, to the best of our knowledge this is the first attempt to parallelize subset construction.

Sequential subset construction has been described in the literature on automata theory by Hopcroft & Ullman (1979) and Aho et al. (1986). Leslie (1995) improves the efficiency of sequential subset construction through data structures for hashing, heaps and bitvectors. This implementation brings two major advantages: avoiding redundant checks for symbols, and taking the advantage of sorted transitions. Because their multiway merging operation takes around 30% $\sim$ 90% of the overall running time, techniques have been suggested to improve this operation. Representing the ADT for DFA states as a hash table is proposed. Leiss (1980) proposed a DFA construction method that reduces DFA size. Johnson & Wood (1996) discuss methods for instruction computation of DFAs. Liu et al. (2011) propose a method to construct a combined DFA for a set of regular expressions. They apply hierarchical merging of DFAs for individual regular expressions.

8 Conclusions

We have parallelized the subset construction algorithm, which is used to convert non-deterministic finite automata (NFAs) to deterministic finite automata (DFAs). We have discussed sources of parallelism in the sequential algorithm, and critically evaluated their profitability on shared-memory multicore architectures. Data-structures for NFAs and DFAs have been chosen to improve scalability and keep communication and synchronization overhead to a minimum. Three different ways of synchronization have been implemented. The performance of our non-blocking synchronization based on a compare-and-swap (CAS) primitive compares favorably to a lock-based approach. We have shown that the amount of work and hence the scalability of parallel subset construction depends on the number of DFA states, which is related to automata density. We demonstrate the efficiency of our parallel subset construction algorithm through several benchmarks run on a 4-CPU (40 cores) node of the Intel Manycore Testing Lab. Achieved speedups are up to a factor of 32x with 40 cores.

9 Acknowledgements

Our research has been partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0005234), and the OKAWA Foundation Research Grant (2009).

References

Aho, A. V., Sethi, R. & Ullman, J. D. (1986), *Compilers: principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Almeida, M., Moreira, N. & Reis, R. (2007), On the performance of automata minimization algorithms, Technical Report DCC-2007-03, Departamento de Ciência de Computadores & Laboratório de Inteligência Artificial e Ciência de Computadores.

Butenhof, D. R. (1997), *Programming with POSIX threads*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Grail+ Project Web Site (retrieved Aug. 2012), '`http://www.csd.uwo.ca/Research/grail`'.

Herlihy, M. & Shavit, N. (2008), *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Hillis, W. D. & Steele, Jr., G. L. (1986), 'Data parallel algorithms', *Commun. ACM* **29**(12), 1170–1183.

Holub, J. & Štekr, S. (2009), On parallel implementations of deterministic finite automata, *in* 'Proceedings of the 14th International Conference on Implementation and Application of Automata', pp. 54–64.

Hopcroft, J. & Ullman, J. (1979), *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA.

Intel Manycore Testing Lab (accessed Aug. 2012), '`http://software.intel.com/en-us/articles/intel-many-core-testing-lab`'.

Johnson, J. & Wood, D. (1996), Instruction computation in subset construction, Technical report, Institute for Information Technology, National Research Council and Hong Kong University of Science and Technology.

Jones, C. G., Liu, R., Meyerovich, L., Asanović, K. & Bodík, R. (2009), Parallelizing the web browser, *in* 'Proceedings of the First USENIX conference on Hot topics in parallelism', HotPar'09, USENIX Association, Berkeley, CA, USA, pp. 7–7.

Ko, Y., Jung, M., Han, Y.-S. & Burgstaller, B. (2011), A speculative parallel DFA membership test for multicore, SIMD and cloud computing environments, Technical Report arXiv:1210.5093, Dept. Computer Science, Yonsei University, Seoul 120-749, Korea, `http://http://arxiv.org/abs/1210.5093`.

Ladner, R. E. & Fischer, M. J. (1980), 'Parallel prefix computation', *J. ACM* **27**(4), 831–838.

Leiss, E. (1980), 'Constructing a finite automaton for a given regular expression', *SIGACT News* **12**(3), 81–87.

Leslie, T. (1995), Efficient approaches to subset construction, Master's thesis, University of Waterloo.

Lin, C. & Snyder, L. (2008), *Principles of Parallel Programming*, Addison Wesley.

Liu, Y., Guo, L., Guo, M. & Liu, P. (2011), Accelerating DFA construction by hierarchical merging, *in* 'Proceedings of the 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications', ISPA '11, IEEE Computer Society, Washington, DC, USA, pp. 1–6.

Luchaup, D., Smith, R., Estan, C. & Jha, S. (2009), Multi-byte regular expression matching with speculation, *in* 'Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection', RAID '09, Springer-Verlag, Berlin, Heidelberg, pp. 284–303.

Luchaup, D., Smith, R., Estan, C. & Jha, S. (2011), 'Speculative parallel pattern matching', *IEEE Transactions on Information Forensics and Security* **6**(2), 438–451.

Misra, J. (2003), 'Derivation of a parallel string matching algorithm', *Inf. Process. Lett.* **85**(5), 255–260.

Paoloni, G. (2010), How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures, Technical report, Intel Corporation.

Raymond, D. & Wood, D. (1995), 'Grail: A C++ library for automata and expressions', *Journal of Symbolic Computation* **17**, 17–341.

Scarpazza, D. P., Villa, O. & Petrini, F. (2007), Peak-performance DFA-based string matching on the Cell processor, *in* '21th International Parallel and Distributed Processing Symposium', pp. 1–8.

Shavit, N. (2011), 'Data structures in the multicore age', *Commun. ACM* **54**(3), 76–84.

Tewari, A., Srivastava, U. & Gupta, P. (2002), A parallel DFA minimization algorithm, *in* 'Proceedings of the 9th International Conference on High Performance Computing', HiPC '02, Springer-Verlag, London, UK, UK, pp. 34–40.

Wang, X., He, K. & Liu, B. (2010), Parallel architecture for high throughput DFA-based deep packet inspection, *in* '2010 IEEE International Conference on Communications', pp. 1–5.

Wood, D. (1987), *Theory of Computation*, John Wiley & Sons, Inc., New York, NY.