A Parallel Algorithm for Minimization of Finite Automata

B. Ravikumar X. Xiong
Department of Computer Science
University of Rhode Island
Kingston, RI 02881
E-mail: {ravi,xiong}@cs.uri.edu

Abstract

In this paper, we present a parallel algorithm for the minimization of deterministic finite state automata (DFA's) and discuss its implementation on a connection machine CM-5 using data parallel and message passing models. We show that its time complexity on a p processor EREW PRAM ($p \leq n$) for inputs of size n is $O(\frac{n \log^2 n}{p} + \log n \log p)$ **uniformly on almost all instances.** *The work done by our algorithm is thus within a factor of* $O(\log n)$ *of the best known sequential algorithm. The space used by our algorithm is linear in the input size. The actual resource requirements of our implementations are consistent with these estimates. Although parallel algorithms have been proposed for this problem in the past, they are not practical. We discuss the implementation details and the experimental results.*

1. Introduction

Finite state machines are of fundamental importance in theory as well as applications. Their applications range from classical ones such as compilers and sequential circuits to recent applications including neural networks, hidden Markov models, hyper-text meta languages and protocol verification. A central problem in automata theory is to minimize a given Deterministic Finite Automaton (DFA). This problem has a long history going back to the origins of automata theory. Huffman [7] and Moore [11] presented an algorithm for DFA minimization in 1950's. Their algorithm runs in time $O(n^2)$ on DFA's with n state and was adequate for most of the classical applications. Many variations of this algorithm have appeared over the years; see [14] for a comprehensive summary. Hopcroft [5] developed a significantly faster algorithm of time complexity $O(n \log n)$.

Although Hopcroft's algorithm is very efficient in practice, it can't handle DFA's with millions of states. But there are many applications involving such large DFA's. A parallel algorithm can handle such instances since the combined memory size of the processors is adequate to store and process such large DFA's. This work is part of a collection of parallel programs we are developing for many computation-intensive testing problems involving DFA's.

we will briefly review the previous work on the design of parallel algorithms for DFA minimization problem. The problem has been studied extensively, [10, 12, 9, 2] etc. JáJá and Kosaraju [9] studied the special case in which the input alphabet size is 1 on a mesh-connected parallel computer and obtained an efficient algorithm. Cho and Huynh [2] showed that the problem is in NC^2 and that it is NLOGSPACE-complete. The algorithm of [12] applies only to the case in which the input alphabet is of size 1. In this case, the most efficient algorithm is due to JáJá and Ryu [10] which is a CRCW-PRAM algorithm of time complexity $O(\log n)$ and total work $O(n \log \log n)$. When the input alphabet is of size greater than 1, no efficient parallel algorithm (i.e., one of time complexity $O(log^k n)$ for some fixed k) is known with a 'reasonable' total work (say, $O(n^2)$ or below). In fact, the only published NC algorithm for this problem is due to [2] which has a time complexity $O(log^2 n)$ *but requires* $\Omega(n^6)$ *processors.*

This paper differs from the work mentioned above in two ways: (i) Our goal is to *implement parallel algorithms* for DFA minimization on actual parallel machines. (ii) Our figure of merit is the *average case* performance, not the worst-case performance. Here we report on the implementation of programs for minimizing DFA's (with unrestricted input alphabet size) on a CM-5 connection machine with 512 processors. We implemented two classes of algorithms - one based

1063-7133/96 $5.00 © 1996 IEEE
Proceedings of IPPS '96

on data parallel programming and the other based on message passing. Our programs perform well in practice – They can minimize 'typical' DFA's with millions of states in a few minutes and their scalability is good. The primary reason for their success is that they have very good performance on almost all instances. We formally establish this using a probabilistic model of Traktenbrot and Barzdin. We also discuss the implementation details and the experiments we conducted with our programs. We conclude with some new questions that arise from our study.

In this abridged version, we omit the proofs of all the claims. They can be found in the full version.

2. Preliminaries

We assume familiarity with basic concepts about finite automata. To fix the notation, however, we will briefly introduce some of the central notions. For details, we refer the reader to [6]. A DFA M consists of a finite set Q of states, a finite input alphabet Σ, a start state $q_0 \in Q$ and a set $F \subseteq Q$ of accepting states, and the transition function $\delta : Q \times \Sigma \rightarrow Q$. An input string x is a sequence of symbols over Σ. On an input string $x = x_1 x_2 ... x_m$, the DFA will visit a sequence of states $q_0 q_1 ... q_m$ starting with the start state by successively applying the transition function δ. Thus $q_1 = \delta(q_0, x_1)$, $q_2 = \delta(q_1, x_2)$ etc. The language $L(M)$ accepted by a DFA is defined as the set of strings x that take the DFA to an accepting state. I.e., x will be in $L(M)$ if and only if $q_m \in F$. Such languages are called regular languages, a well-understood and important class. Two DFA's M_1 and M_2 are said to be equivalent if they accept the same set of strings.

It is common to present DFA minimization as an equivalent problem called the *coarsest set partition* problem [1] which is stated as follows: Given a set Q, an initial partition of Q into disjoint blocks $\{B_0, ..., B_{m-1}\}$, and a collection of functions $f_i : Q \rightarrow Q$, find the *coarsest* (having fewest blocks) partition of Q say $\{E_1, E_2, ..., E_q\}$ such that: (1) the final partition is *consistent* with the initial partition, i.e., each E_i is a subset of some B_j and (2) a and b in E_i implies for all j, $f_j(a)$ and $f_j(b)$ are in the same block E_k. The connection to minimization of DFA is the following. Q is the set of states, f_i is the map induced by the symbol a_i (this means, $\delta(q, a_i) = f_i(q)$), the initial partition is the one that partitions Q into two sets, F the set of accepting states and $Q \backslash F$ the nonaccepting ones. It should be clear that the size of the minimal DFA is the number of equivalence classes in the coarsest partition. In the following, we will assume that the input is a DFA with n states defined over a k-letter alphabet *in which every state is reachable from the start state.* This assumption simplifies the presentation of our algorithm; otherwise a preprocessing step should be added to delete the unreachable state.

3. Parallel Algorithm and Its Analysis

A high level description of our parallel algorithm is given below:

```
Algorithm ParallelMin1
input: DFA with n states, k inputs, 2 blocks B_0, B_1.
output: minimized DFA.
begin
  do
    for i = 0 to k - 1 do
      for  j = 0 to n - 1 do in parallel
        Get the block number B of state q_j;
        Get the block number B' of the state δ(q_j, x_i),
          label the state q_j with the pair (B, B');
        Re-arrange (e.g. by parallel sorting) the states
          into blocks so that the states with the same
          labels are contiguous;
      end_parallel_for;
      Assign to all states in the same block a new
          (unique) block number in parallel;
    end_for;
  until no new block produced;
  Deduce the minimal DFA from the final partition;
end.
```

The outer loop of the above algorithm appears inherently sequential and it seems very hard to parallelize it significantly. But this is not of great concern since experiments show that the total number of iterations of the outer loop is very small (usually less than 10) for all the DFA's we tested of sizes ranging from 4 thousands to more than 1 million. This relative invariance of the number of iterations can be explained theoretically; see the discussion below. Thus, in practice, there is no real gain in attempting to parallelize the outer loop.

In the remainder of this section, we will mainly discuss the high-level details of implementing the above parallel algorithm on EREW-PRAM model and present an analysis that holds for *almost all instances in a precise sense.*

We start with a brief description of the model due to Traktenbrot and Barzdin [13]. Our notation will be similar to the one used by [3] who adapted Traktenbrot-Barzdin model in their study of passive learning algorithms. Our model involves a slight generalization of theirs. Basically, the probability space on which we base our average case is narrowed by fixing a DFA G arbitrarily *except for its set of accepting states.* Thus an adversary can choose G (called the automaton graph) subject to the only condition that all states are reachable from the start state. The probability distribution is defined over all DFA's M_G which can be derived from

this automaton graph by randomly selecting the set F of accepting states. In our model, we will allow each state to be included in F independently with probability p (p can be a function of n).

Let $P_{n,\varepsilon,p}$ be any predicate on an n-state automaton which depends on n, a confidence parameter ε (where $0 \leq \varepsilon \leq 1$) and p (defined above). We say that **uniformly almost all automata** have property $P_{n,\varepsilon,p}$ if the following holds: for all $\varepsilon > 0$, for all $n > 0$ and for any n-state underlying automaton graph G, if we randomly choose the set of accepting states where each state is included independently with probability p, then with probability at least $1 - \varepsilon$, the predicate $P_{n,\varepsilon,p}$ holds.

Let $M = < Q, \Sigma, q_0, \delta, F >$ be a DFA. The output associated with a string $x = x_1...x_m$ from state q defined as the sequence $y_0...y_m$ where $y_i = 1$ if the state reached (starting at q) after applying the prefix $x_1...x_i$ is an accepting state, 0 otherwise. (Note that y_0 is 1 or 0 according as q is accepting or not.) We denote this output by $q < x >$. A string $x \in \Sigma^*$ is a distinguishing string for q_i and q_j if $q_i < x > \neq q_j < x >$. Finally define $d(q_i, q_j)$ as the length of the shortest distinguishing string for the states q_i and q_j. We are now ready to present our results about the expected performance of *ParallelMin1*. In the following, lg will denote logarithm to base 2.

Lemma 1. *For uniformly almost all automata with n states, every pair of states has a distinguishing string of length at most*

$$\frac{2\lg(\frac{n^2}{\varepsilon})}{\lg(\frac{1}{1-2p(1-p)})}.$$

Our next result presents the average time complexity (in the above sense) of an EREW (exclusive-read, exclusive-write) PRAM implementation of the parallel algorithm presented above. We will assume that the readers are familiar with the PRAM model. See [8] for a systematic treatment of the PRAM algorithms.

Theorem 2. *For uniformly almost all automata with n states, the algorithm ParallelMin1 can be implemented on an EREW PRAM with $q \leq n$ processors with time complexity $O(k\frac{n}{q}\frac{\lg n \lg(n^2/\varepsilon)}{\lg(1/(1-2p(1-p)))} + \lg q \lg n)$. The space complexity (= amount of shared memory used) is $O(n)$. Thus the total work done by the algorithm is $O(n \log^2 n)$ if p and k are constants.*

The next result explains why Hopcroft's algorithm does not parallelize well.

Theorem 3. *Let Q' be the set of states of a DFA after minimization, and i the iteration number of the Hopcroft's algorithm, then*

$$i \geq |\Sigma| \cdot (|Q'| - 1).$$

4. Implementation as Data Parallel Program

Data-parallelism is the parallelism achievable through the simultaneous execution of the same operation across a set of data [4]. Since each operation is applied simultaneously to the data, a data-parallel program has only one program counter and is characterized by the absence of multi-threaded execution which makes the design, implementation and testing of programs very difficult. Although data parallel programming was originally developed for SIMD (or vector) hardware, it has become a successful paradigm in all types of parallel processing systems such as shared-memory multiprocessors, distributed memory machines, and networks of workstations.

Since our parallel algorithm *ParallelMin1* has a simple control structure and uses an uniform data structure (array), it is easy to implement it as a data parallel program. We present our data parallel version of the algorithm in the pseudo code similar to C*, a data parallel extension of C programming language.

We assume in our implementation that the computation begins with the input DFA already stored as parallel variables.

The data structures used are as follows.

- *delta* is an array of $|\Sigma|$ parallel variables, each is a parallel one of size n. The i-th element of the *delta*[j], [i]*delta*[j], stores the transition function of state i under input symbol j (i.e., the next state of state i under input symbol j, $\delta(i,j)$), $0 \leq i \leq n-1$, $0 \leq j \leq |\Sigma| - 1$.

- *block* is a parallel variable to store partition numbers for all states. [i]*block* is the block number of state i.

In *ParallelMin2* below, statements like [*parallel_index*]*parallel_var1* = *parallel_var2* are *send* operations which send values of parallel variable (*parallel_var2*) elements to other parallel variable (*parallel_var1*) elements according to index mapping *parallel_index*. statements like *parallel_var1* = [*parallel_index*]*parallel_var2* are *get* operations which get values of parallel variable (*parallel_var1*) elements from other parallel variable (*parallel_var2*) elements.

```
Algorithm ParallelMin2
input: delta, array of parallel variables to store transition
    function; block, initial block numbers of the DFA;
output: block, (updated) block numbers denote state
    equivalence classes of the minimized DFA;
begin
  q = 2;
  do
    check = 0;
    for k = 0 to |Σ| − 1 do
      Get block numbers:
        stateIndex = pcoord(0); {stateIndex is used to
          keep track of movement of state indices}
        output[0] = block;
        output[1] = [delta[k]]block;
      Sort according to outputs:
        sorted = rank(output[0]);
        [sorted]stateIndex = stateIndex;
        [sorted]output[0] = output[0];
        [sorted]output[1] = output[1];
        sorted=rank(output[1]);
        [sorted]stateIndex = stateIndex;
        [sorted]output[0] = output[0];
        [sorted]output[1]=output[1];
      Set new block names:
        link = 0;
        where(output[0]≠[-1]output[0]or
              output[1]≠[-1]output[1])
          link = 1;
        newBlock = prefixSum(link);
      Check if new blocks produced:
        if ( q equal to [n-1]newBlock )
          check++; continue; {consistent for input k}
        else [stateIndex]block=newBlock;
        q = [n-1]newBlock;
    end_of_for;
  until (check equal to |Σ|);{consistent for all inputs}
end.
```

In our implementation, sorting is done by finding ranks (a system library function) of elements and three *get* operations to re-distribute them. The parallel prefix computation is performed by a library function as well. The performance characteristics of ParallelMin2 are discussed in section 6.

5. Implementation as Message Passing Program

Now we present the message passing version of the basic parallel algorithm. It is a little more complicated than the data parallel version because we have to control communication between processors explicitly.

Our approach for storing the input DFA is to divide the transition function and the final state information equally among p different processors. Storage of the intermediate results will also be distributed among the processors. This is a standard approach used in distributed memory parallel algorithms.

For simplicity, we assume that p divides n, the number of states in the input DFA. Processors will be identified by integers from 0 to $p-1$. A detailed description of the various steps of this algorithm can be found in the full paper.

```
Algorithm ParallelMin3 for the processor myAdd
input: delta[0..n/p − 1] array to store part of the transition
    function. block[0..n/p − 1], initial block numbers of states
    stored in this processor.
output: (updated) block[0..n/p − 1]
begin
  done = false;
  Initialization of indexedBuffer:
    indexedBuffer[0, 0..n/p − 1]= myAdd · n/p . .
    (myAdd + 1)· n/p − 1. So first row of
    indexedBuffer contains the state numbers associated
    with this processor.
    indexedBuffer[1, 0..n/p − 1] = block[0..n/p − 1], put
    current block numbers of the states in this processor;
  Construct Message Passing Tables according to
    delta. These tables contain for each state q in the
    processor the address of the processor that contains
    δ(q, x_j) and of the processors which contain the
    state(s) δ^{-1}(q, x_j) for each j.
  while (not done)
    done = true;
    for k =0 to |Σ| − 1 do
      Exchange outputs: All-to-all communication takes
        place as follows: Suppose a state s_t belongs to a
        processor t and suppose δ(s_t, x_j) belongs to
        myAdd. The processor will send a message to t
        containing the block number of δ(s_t, x_j).
        The received messages will be put the third row,
        namely indexedBuffer[2, 0..n/p + 1].
      Parallel-sort: Sort indexedBuffer using the second
        and third row together as key. Note that After
        sorting, indexedBuffer's first row (state indices)
        may not be ordered and may not belong to
        this node. That is, the sorting will re-distribute
        data items.
      Rename blocks: If two states have the same 'key',
        they are put in the same block in the refined parti-
        tion. Put new block numbers in indexedBuffer's
        second row. This is a global scanning operation on
        indexedBuffer of all processors.
      Parallel-sort: Sort indexedBuffer using the first row
        as the key. This will bring the new block number
        of each state back to the processor to which it
        belongs.
      if (a new block produced) done = false;
    end_of_for;
  end_of_while;
  block[0..n/p − 1] = indexedBuffer[1, 0..n/p − 1];
end.
```

6. Experimental Results on CM-5

Next, we present our experimental results on CM-5. We ran the programs described above for many DFA's of different sizes on a CM-5 with 32 nodes. Each node has 4 vector units. The instances of DFA's used in this experiment were constructed using four different methods. All of them have two input symbols.

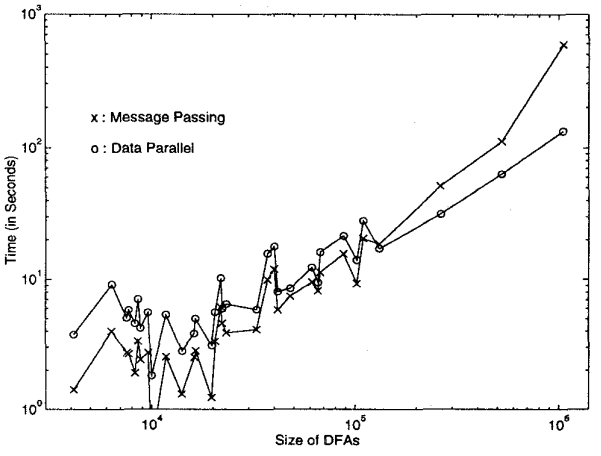

Figure 1. Running Time vs. Problem Size

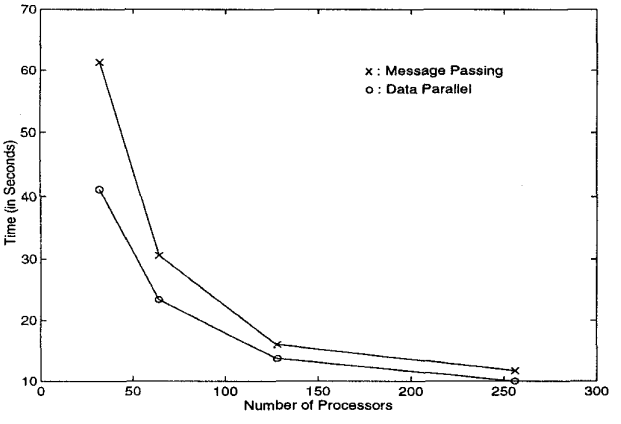

Figure 2. Scalability

In Figure 1, the number of processors is fixed at 32. When the input size is less than about 100000, the time complexity is almost linear so that it takes about twice as much time to solve an instance twice as large. But when the input size exceeds 100000, nonlinearity shows up, especially for the message passing program.

To study the scalability of our programs, we chose three DFA's (with 109078, 262144 and 524288 states) and minimized them by using both programs on a CM-5 configured with 32, 64, 128 and 256 nodes. Figure 2 shows the results of average time. Both programs seem to scale well; but the message passing program is the better one.

7. Acknowledgments

Our programs were implemented and tested on a connection machine CM-5 at the National Center for Supercomputer Applications at Urbana-Champaign. We thank A. Tridgell of Australian National University for his help with the sorting routine used in the message passing program.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Algorithms.* Addison-Wesley Publishing Co., Reading, MA, 1973.

[2] S. Cho and D. Huynh. The parallel complexity of coarsest set partition problems. *Information Processing Letters*, 42:89–94, 1992.

[3] Y. Freund et al. Efficient learning of typical finite automata from random walks. In *25th ACM Symposium on Theory of Computing*, pages 315–324, 1993.

[4] P. Hatcher and M. Quinn. *Data-parallel Programming on MIMD Computers.* MIT Press, Cambridge, Mass., 1991.

[5] J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.

[6] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Co., Reading, Mass., 1978.

[7] D. Huffman. The synthesis of sequential switching circuits. *Journal of Franklin Institute*, 257(3):161–190, 1954.

[8] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Co., Reading, Mass., 1992.

[9] J. JáJá and S. Kosaraju. Parallel algorithms for planar graph isomorphism and related problems. *IEEE Transactions on Circuits and Systems*, 35(3), March 1988.

[10] J. JáJá and K. Ryu. An efficient parallel algorithm for the single function coarsest partition problem. *ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[11] E. Moore. Gedanken-experiments on sequential circuits. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[12] Y. Srikant. A parallel algorithm for the minimization of finite state automata. *Intern. J. Computer Math.*, 32:1–11, 1990.

[13] B. Trakhtenbrot and Y. Barzdin'. *Finite Automata: Behavior and Synthesis.* North-Holland Publishing Company, 1973.

[14] B. Watson. A taxonomy of finite automaton minimization algorithms. Technical report, Faculty of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands, 1994.