A Parallel Algorithm for Minimization of Finite Automata

B. Ravikumar X. Xiong
Department of Computer Science
University of Rhode Island
Kingston, RI 02881
{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 [14] and Moore [19] 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 [27] for a comprehensive summary. Hopcroft [12] developed a significantly faster algorithm of time complexity $O(n \log n)$. This algorithm can be coded to minimize DFA's of size up to several thousands in a few seconds. (Here the size of a DFA is taken as the product of the number of states and the size of the alphabet.)

Despite the existence of an efficient sequential algorithm like Hopcroft's, design and implementation of a *parallel* algorithm for DFA minimization is important for several reasons. The first reason has do with space complexity. All the algorithms for minimization (including Hopcroft's) take linear space or more. Thus when the size of the input DFA is very large, say in millions, most of the single processor machines do not have enough (main) memory to run the algorithm. The problem now takes on a different character as the external memory operations become dominant. Parallel machines can handle such instances. Such large inputs are common in some recent applications involving DFA's. Another reason for this study is that our minimization algorithm will be used as a preprocessing step for more computation-intensive problems which require parallelization - such as the problem of finding a *homing sequence* or a *checking sequence* in a large finite state machine [20, 18]. We also wanted to understand the *average case* behavior of various sequential algorithms for this problem and determine which of them is (are) likely to parallelize well in practice. Finally, we wanted to understand the effectiveness of different parallel programming paradigms (data parallelism vs. message passing) using DFA minimization as a test case. Sorting and some basic graph theoretic problems have been studied in the past as test cases for parallel program design. But our work appears to be the first one to report an actual *implementation* of parallel algorithms for DFA minimization.

First, we will briefly review the previous work on the design of parallel algorithms for DFA minimization problem. The problem has been studied extensively, [16, 22, 17, 5] etc. JáJá and Kosaraju [17] 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 [5] showed that the problem is in NC^2 and that it is NLOGSPACE-complete. The algorithm of [22] 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 [16] 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 [5] which has a time complexity $O(log^2 n)$ *but requires* $\Omega(n^6)$ *processors.*

In this work, 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 are currently in the process of testing our code on other parallel machine platforms as well). We have implemented two classes of algorithms - one based on data parallel programming and the other based on message passing. Although our parallel algorithms are based on a simple sequential algorithm of worst-case time complexity $O(n^2)$, they perform very well in practice. A main 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. 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 $x = x_1x_2...x_m$, the DFA will visit a sequence of states $q_0q_1...q_m$ 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.

The problem of DFA minimization is to find a DFA with the fewest possible states equivalent to a given DFA. A central result of the theory of finite automata is that the minimal DFA is unique up to isomorphism and the number of states in the minimal DFA is given by the number of equivalence classes in the partition on the set of all strings Σ^* defined (with respect to the underlying regular language L) as follows: Two strings x and y are equivalent if and only if for all strings z, $xz \in L$ if and only if $yz \in L$. We say that two strings x and y that belong to different equivalence classes in the above partition *distinguishable* and a z such that exactly one of xz, yz is in L their *distinguishing string.* A typical minimization algorithm repeatedly searches for such distinguishable strings and refines the current partition until no further split is possible. It is common to present the minimization as an equivalent problem called the *coarsest set partition* problem 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 *ParallelMin 1*
input: DFA with 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**

(1) Get the *block number* B of state q_j;

(2) Get *the block number* B' of the state $\delta(q_j, x_i)$, label the state q_j with the pair (B, B');

(3) Re-arrange (e.g. by parallel sorting) the states into blocks so that the states with the same labels are contiguous;

      end_parallel_for;

(4) Assign to all states in the same block a new (unique) block number in parallel;

end_for;
(5)**until** no new block produced;
(6)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 begin by defining formally a probability distribution over DFA's. The standard model for defining the average case complexity of any problem is to assign a uniform (or any other) probability distribution over all instances of a certain fixed 'size' (say the number of states). But our probabilistic model is slightly different and actually yields much stronger results. There are two reasons for not using the standard model. The first one is that the inputs in the standard model with a fixed number of states include DFA's in which some of the states are not reachable from the starting state. Such inputs clearly violate our assumptions about the input to our algorithm. Another reason is that we want to obtain as strong as a result as possible in the following sense. We want to allow as many 'components' of the input as possible to be chosen adversarially so that the model will be close to the worst case model.

A model thoroughly investigated by Traktenbrot and Barzdin [25] suits both our requirements. We first present the central ideas of this model. Our notation will be similar to the one used by [8] 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 $0 \le \varepsilon \le 1$) 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.

An important point to note here is that our average-case results are of the form they hold with 'high probability'. Such claims are stronger than mere expected case claims. They also assert that most of the instances are concentrated about the expected value. It has been argued (see, e.g. [15]) that in parallel computation expected value claims are not strong enough; high probability arguments are more convincing.

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 a 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 [15] 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 a EREW PRAM with $q \le n$ 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.

We will now look at Hopcroft's algorithm and try to understand why it does not yield a good parallel algorithm. In the rest of the section, we will assume familiarity with Hopcroft's algorithm. We have noticed that it is also easy to parallelize each iteration of the loop, but as in our main algorithm, the difficulty is in parallelizing the successive iterations of the outer loop. However, there is an important difference. While uniformly on almost all instances, our algorithm performs only $O(\lg n)$ iterations, Hopcroft's algorithm seems to perform a very large number of them. An explanation for this is provided in the next result.

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 [11]. 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 [23].

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.

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** $|\Sigma| - 1$ **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 $|\Sigma|$); {consistent for all input, done}

end.

In the above code, 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.

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

In the message-passing approach to parallel programming, a collection of processes executes programs written in a standard sequential language augmented with calls to a library of functions for sending and receiving messages. Message-passing is probably the most widely used parallel programming model today. Message-passing programs create multiple tasks with each task encapsulating local data [7]. Our goals in implementing the algorithm on the message passing model are the following: (i) To compare this approach with data parallel model. (ii) To understand more precisely the communication cost of implementing the algorithm and make the performance of the algorithm more predictable. (iii) Make the program more portable as message passing is the most widely available model of parallel programming.

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 devides n, the number of states in the input DFA. Processors will be identified by integers from 0 to $p-1$.

Data structures in the processor with address $myAdd$:

- $delta[0..n/p-1, 0..k-1]$, partial transition function in this node. $delta[i][j]$ is the next state of state $myAdd \cdot \frac{n}{p} + i$ under input symbol j.
- $block[0..n/p-1]$, block numbers of states in the node. $block[i]$ is the block number of the state $myAdd \cdot \frac{n}{p} + i$.
- $indexedBuffer[0..2][0..n/p]$, the working space of the algorithm. $indexedBuffer[0]$ stores all state numbers in the node, initially ordered. $indexedBuffer[1]$ stores the current block numbers of states in the node. $indexedBuffer[2]$ stores the block number of 'next' states in the node under an input symbol.

ALGORITHM *ParallelMin3* for the processor $myAdd$

input: $delta[0..\frac{n}{p}-1]$ array to store part of the transition function for states from $myAdd \cdot \frac{n}{p}$ to $(myAdd+1) \cdot \frac{n}{p} - 1$;
$block[0..\frac{n}{p}-1]$, initial block numbers for states $myAdd \cdot \frac{n}{p}$ to $(myAdd+1) \cdot \frac{n}{p} - 1$;

output: $block[0..\frac{n}{p}-1]$ storing the block number, in the **final** partition, of states from $myAdd \cdot \frac{n}{p}$ to $(myAdd+1) \cdot \frac{n}{p} - 1$;

begin

(1) $done =$ **false**;

(2) Initialization of $indexedBuffer$: $indexedBuffer[0, 0..\frac{n}{p}-1] = myAdd \cdot \frac{n}{p}\ ..\ (myAdd+1) \cdot \frac{n}{p} - 1$. So first row of *indexedBuffer* contains the state numbers associated with this processor.
$indexedBuffer[1, 0..\frac{n}{p}-1] = block[0..\frac{n}{p}-1]$, put current block numbers of the states in this processor;

(3) Construct Message Passing Information Tables according to $delta$. The table contains for each state q in the processor the address of the processor that contains $\delta(q, x_j)$ and of the processors which contain the state(s) $\delta^{-1}(q, x_j)$ for each j.

while (not *done*)

$done =$ **true**;

for $k = 0$ **to** $|\Sigma| - 1$ **do**

(4) Exchange outputs: All-to-all communication takes place as follows: Suppose a state s_t belongs to a processor t and suppose $\delta(s_t, x_j)$ belongs to $myAdd$. The processor will send a message to t containing the block number of $\delta(s_t, x_j)$. The received messages will be put the third row, namely $indexedBuffer[2, 0..n/p+1]$. This implements Step (2) of *ParallelMin1.*

(5) Parallel-sort: Sort *indexedBuffer* using the second and third row together as key. This is Step (3) of *ParallelMin1* 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.

(6) Rename blocks: If two states have the same 'key', they are put in the same block in the refined partition. Put new block numbers in *indexedBuffer*'s second row. This is a global scanning operation on *indexedBuffer* of all processors and implements Step (4) of *ParallelMin 1*.

(7) 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.

(8) **if** (a new block produced) *done* = **false**;

end_of_for;

end_of_while;

$block[0..\frac{n}{p}-1] = indexedBuffer[1,\ 0..\frac{n}{p}-1];$

End.

6 Experimental Results on CM-5

Next, we present our experimental results on CM-5. We have run algorithms discussed 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.

In the first method, we started with a NFA (having number of states between 50 and 120) created randomly as follows: For each state and each input symbol, we create a certain number (2 to 5) transitions on this symbol to randomly selected states, each state being chosen with the same probability. The accepting states are also randomly generated. The NFA is converted to a DFA using subset construction in a breadth-first manner. In the second method, a number of relatively small DFA's are randomly created and a NFA to accept their union is created; this NFA is converted to a DFA. In the third method, a DFA is randomly chosen; it is then made into a NFA by introducing nondeterministic transitions from some of the states. It is converted to a DFA and used as the test case. Finally, to obtain large DFA's, we directly create DFA's by introducing transitions randomly from each state. Note that our experimental set-up is different from the probabilistic model presented in section 3. We wanted to choose DFA's as they arise in real applications. Our choice of input generation was partly motivated by the fact that large DFA's arise when NFA's are converted to DFA's. Presently, we are testing our algorithms on DFA's that model bench mark circuits.

Figure 1: Running Time vs. Problem Size

We report here on the experiments conducted on several dozens of DFA's carefully chosen by trial and error repetition of the above methods. All DFA's were tested on the data parallel as well as message passing program. Only some of these DFA's could be successfully minimized using the parallel version of Hopcroft's algorithm. Figure 1 shows the running time in seconds for data parallel and message passing implementation as DFA sizes increase.

The time to run the message passing program was taken as the maximum elapsed CPU time of all nodes (provided by the CM-5 CMMD library). This timing is reliable since all nodes are synchronized at the beginning and end of the program (and also in each iteration). The time to run the data parallel program (elapsed CPU time) was collected by the system functions available in C*.

The experiments showed that for almost all DFA instances tested, the number of iterations of the outer loop was less or equal to 10 (except one with 14). Many of them were less than 6. However, the number of iterations of the outer loop in parallelized Hopcroft's algorithm is much bigger. These observations agree with the results of section 3.

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 larger. But when the input size exceeds this, nonlinearity shows up, especially for the message passing program.

To study the scalability of our programs, we chose

Figure 2: Scalability

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.

Although we only present results for DFA's with size up to one million, our tests included DFA's of size up to 4 million states. Specifically, we tested a DFA with 4 million states on 32 node CM-5 using the data-parallel program. The total run time was about 637.5 seconds (9.5 minutes), in which 570.2 seconds were spent on sorting. The used space reported by the job controller was 683M bytes. We also tested a 2 million state DFA using the message passing program on 32 node CM-5. The total time was about 2221.6 seconds (37.0 minutes), in which 2210.0 seconds were spent on sorting. Space used was 216M. The data parallel program is better than message passing program because it has a more efficient sorting algorithm. But message passing algorithm seems to be more economical in the use of storage.

Problems like sorting, graph connectivity and DFA minimization belong to the 'low complexity' domain. Parallel programs for such problems are appropriate only when the input size is very large. We found that our parallel programs outperformed the sequential version of Hopcroft's algorithm running on a Sparc IPX sometimes for sizes as small as 70000. Since the programs have shown scalability, it should be possible to use our parallel programs to minimize DFA's with several millions of states by employing a bigger configuration of CM-5 and/or with more storage.

Some final remarks about our experiments:

- The programs ran on the CM-5 time-sharing mode, not dedicated mode. So time-sharing swapping between jobs might have slowed down the performance. On a dedicated mode, the programs might perform better.

- We used some timers (3 to 5) to get timing information about different program segments. Some extra statements were inserted in the programs to trace their running (e.g., to get the loop number). These also slightly slowed down our programs.

- Both implementations used a standard library or a previously implemented program for parallel sorting. This made our program design and testing simpler. But this also might have made our programs inefficient since these routines were not specially coded to conform to our data structures. Experiments show that most of the time is spent on sorting, about 50% to 90% (and as high as 99% for larger inputs). In any case, step (3) does not require sorting. Replacing the sorting routine by a more efficient routine to group identical keys possibly improve the performance of both programs. This is also likely to make the message processing program competitive with the data parallel program.

- We did not try all possible means to optimize our code, such as using the best possible library functions, the best communication method, etc. We hope to do this in the future.

7 Conclusions

We presented parallel implementations of an algorithm to minimize a DFA. This problem has a wide range of applications. The classical applications involved input instances of size up to a few thousands and this range can be handled very well by the sequential algorithm of Hopcroft whose time complexity is $O(n \log n)$. However, some recent applications involve DFA's that are too large even for the most powerful workstations. It is for such applications that we need a parallel algorithm that works well in practice. The problem of designing a parallel algorithm for this problem has received wide attention [16, 17, 5] but none of these algorithms are realistic. Although they have a desirable time complexity of $O(\log^2 n)$, the number of processors n^6 needed to achieve this time complexity is astronomical for n as small as 20. The next alternative is to parallelize Hopcroft's algorithm, the fastest sequential algorithm for this problem. This algorithm does not parallelize well as explained in section 3. Finally, we parallelized a simple sequential algorithm of time complexity $O(n^2)$ and found it to have a good performance over a wide range of input sizes. Part of the reason for its success is that although the total work performed by this algorithm in the worst case is rather prohibitive (more than n^2) its performance is very good on most of the instances. Our analysis explains this by showing that its time complexity is $O(\frac{n \ \log^2 n}{p} + \log p \log n)$ using p processors on almost all instances of size n.

We conclude by presenting some questions that arose in the course of our study. Two interesting theoretical problems are:(i) Design a sublinear time parallel algorithm for DFA minimization such that the total work done is quadratic (or less). (ii) Show that Hopcroft's algorithm is inherently sequential (in the sense of [10]). A practical problem is to improve our programs so that they achieve comparable performance on *all* instances.

8 Acknowledgments

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

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, "Compilers: Principles, Techniques and Tools", *Addison-Wesley Publishing Co.*, Reading, Mass., 1988.

[2] W. Brauer, "On Minimizing Finite Automata", *EATCS Bulletin* 35, pp. 113-116, 1988.

[3] M.K. Brown and J.G. Wilpon, "A Grammar Compiler for Connected Speech Recognition", *IEEE Transactions on Signal Processing*, Vol.39, No.1, January 1991.

[4] S. Cho and D.T.Huynh, "The Parallel Complexity of Coarsest Set Partition Problems", *Information Processing Letters* 42 pp.89-94, 1992

[5] S. Cho and D.T.Huynh, "The parallel complexity of Finite-State Automata Problems", *Information and Computation*, Vol.97, No.1, pp.1-22, 1992.

[6] T. H. Cormen, C. Leiserson and R. L. Rivest, "Algorithms", *McGraw Hill*, NY, 1990.

[7] I. Foster, "Designing and Building Parallel Programs", *Addison-Wesley Publishing Co.*, Reading, Mass., 1995.

[8] Y. Freund et al., "Efficient Learning of Typical Finite Automata from Random Walks", *25th ACM Symposium on Theory of Computing*, pp.315-324, 1993.

[9] D. Gries, "Describing an Algorithm by Hopcroft", *Acta Informatica* 2, pp.97-103, 1973.

[10] R. Greenlaw, "A Model Classifying Algorithms as Inherently Sequential with Applications to Graph Searching", *Information and Computation*, Vol.97, No.2, pp.133-149, 1992.

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

[12] J. Hopcroft, "An $n \log n$ Algorithm for Minimizing States in a Finite Automaton", *Theory of Machines and Computations*, Academic Press, New York, pp.189-196, 1971.

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

[14] D.A. Huffman, "The Synthesis of Sequential Switching Circuits", *Journal of Franklin Institute*, 257(3), pp.161-190, 1954.

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

[16] J. JáJá and K.W. Ryu, "An Efficient Parallel Algorithm for the Single Function Coarsest Partition Problem", *ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[17] J. JáJá and S.R. Kosaraju, "Parallel Algorithms for Planar Graph Isomorphism and Related Problems", *IEEE Transactions on Circuits and Systems*, Vol.35, No.3, March 1988.

[18] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines — A Survey", *Unpublished Manuscript,* 1994.

[19] E.F. Moore, "Gedanken-experiments on Sequential Circuits" *Automata Studies*, (Ed: C. Shannon and J. McCarthy), *Princeton University Press*, Princeton, NJ, pp.129-153, 1956.

[20] I. Pomeranz and S.M. Reddy, "Application of Homing Sequences to Sequential Circuit Testing", *IEEE Transactions on Computers*, Vol. 43, No.5, May 1994.

[21] D. Revuz, "Minimisation of Acyclic Deterministic Automata in Linear Time" *Theoretical Computer Science*, 92, pp.181-189, 1992.

[22] Y.N. Srikant, “A Parallel Algorithm for the Minimization of Finite State Automata”, *Intern. J. Computer Math.* Vol. 32, pp.1-11, 1990.

[23] Thinking Machines Corporation, “C* Programming Guide”, *Thinking Machines Corporation*, Cambridge, Mass., May, 1993.

[24] Thinking Machines Corporation, “CMMD User Guide”, *Thinking Machines Corporation*, Cambridge, Mass., May, 1993.

[25] B.A. Trakhtenbrot and Ya.M. Barzdin’, “Finite Automata: Behavior and Synthesis”, *North-Holland Publishing Company*, 1973.

[26] A. Tridgell and R.P. Brent, “An Implementation of a General-Purpose Parallel Sorting Algorithm”, *Report TR-CS-93-01, Computer Science Laboratory, Australian National University*, February, 1993.

[27] B.W. Watson, “A Taxonomy of Finite Automaton Minimization Algorithms”, *Technical Report of Faculty of Mathematics and Computer Science, Eindhoven University of Technology*, The Netherlands, 1994.

[28] D. Wood, “Theory of Computation”, *John-Wiley & Sons*, NY, 1987.

[29] X. Xiong, *Ph.D. Thesis (in preparation).*