Implementing Sequential and Parallel Programs for the Homing Sequence Problem

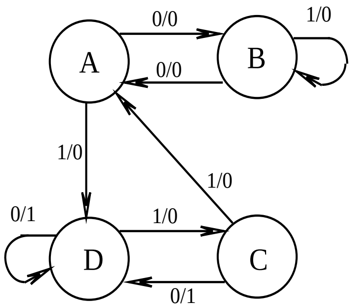
B. Ravikumar and X. Xiong
Department of Computer Science
University of Rhode Island
Kingston, RI 02881
{ravi,xiong}@cs.uri.edu

Abstract

Homing sequences play an important role in the testing of finite state systems and have been used in a number of applications such as hardware fault-detection [7], protocol verification [4], and learning algorithms [11, 3, 1] etc. Here we present a parallel program implementation that finds a homing sequence for an input DFA. Our program can handle *randomly* generated instances with millions of states, and *all* DFA's with thousands of states. In addition to the design, analysis and implementation of the algorithm, we also discuss what constitute good test cases to test programs that deal with finite automata.

1 Introduction

In most applications involving finite automata, the underlying DFA is a black-box whose internal states are not accessible. Testing problem involves determining if the DFA has a certain property using experiments. An experiment consists of applying an input and observing the output. Before performing a test, the DFA should be brought to a known state. An input sequence x with the property that the output on x uniquely determines the state reached (after applying x) is known as a *(preset) homing sequence*. For example, 010 is a homing sequence for the machine described in Figure 1. Table 1 shows the machine's response to the sequence. Note that although 010 produces the same outputs from states C and D, the definition of a homing sequence is not violated since the same state is reached in both cases. However, 01 is not a homing sequence since the outputs produced from states A and B on 01 are the same, but they reach two distinct states after reading this input. Finding a homing sequence in a finite automaton is an important problem in the testing of finite state machines and has been studied and used extensively since mid 1950's. Some recent applications of homing sequence can be found in the following list of papers: [11, 7, 3, 1]. These applications span a wide range of topics: exploration of an unknown environment [11, 3, 1], hardware fault-detection [7], protocol verification [4] etc.



State	Next State,Output input=0	input=1
A	B,0	D,0
B	A,0	B,0
C	D,1	A,0
D	D,1	C,0

Figure 1: A DFA Example

Every minimal finite automaton with n states has a homing sequence of length $O(n^2)$. It is possible to find such a sequence in time $O(n^3)$. In recent applications, the DFA's for which a homing sequence is required have several thousands of states. Such instances are hard to handle on a uniprocessor machine both because of the space as well as time requirements.

A carefully designed parallel program is not only desirable but perhaps the only way to handle such instances. Our goal is to design a parallel algorithm with provably good time complexity which can also be implemented on a real parallel machine. (Even some of the sequential programs used in practice have not been analyzed, or are not guaranteed to work on all inputs.)

Table 1: Response to the Sequence 010

Initial State	Response	Final State
A	000	A
B	001	D
C	101	D
D	101	D

With this goal, we designed two parallel algorithms for this problem. The two algorithms use very different techniques. One of them is a deterministic algorithm of time complexity $O(\sqrt{n}log^2 n)$ and the other one is a randomized parallel algorithm [10]. The randomized algorithm is an RNC algorithm. Both algorithms are theoretically efficient, but they are not useful in practice for several reasons: The RNC algorithm has a large processor requirement (about $O(n^6)$) which clearly limits the size of the inputs it can handle. The deterministic algorithm is very complicated and is communication intensive. Neither algorithm can be implemented without some drastic simplification and/or modification of goals (such as good performance *on average* rather than in the worst-case). Here we present a simple parallel program implementation for this problem.

First, we will state succinctly the objectives of our implementation. Every DFA has many homing sequences and in fact, a very long random sequence is almost surely a homing sequence. (Often, even a short random sequence is a homing sequence, but this is not true when the DFA is selected by an adversary.) Thus our problem is actually an optimization problem; clearly, shorter homing sequences are more desirable. Unfortunately, finding the shortest length homing sequence is NP-hard [11], and NP-hardness holds even when restricted to special cases such as permutation DFA's [8]. Thus any homing sequence of length bounded by polynomial in n, the number of states, is acceptable.

The parallel computers on which our algorithms have been implemented are IBM SP-2 and CM-5. These are *coarse-grained* parallel computers. The number of processors is relatively small (512 or 1024), and each processor is a powerful stand-alone workstation. We require our parallel program to satisfy the following criteria: (i) The program should work on *very large* size randomly chosen input DFA's. Our goal is to handle DFA's with close to 1 million states. It seems unlikely that *every* DFA with such large size can be handled even by a parallel program running on the fastest and the largest super computer since the size of the output itself is of the order of 10^{12}. On the other hand, we would like our parallel program to be able to handle *every input DFA with a few thousand states.* These two cases (random instances of the order 10^6 and every instance of order 10^3) are already beyond the limits of serial programs. The main reason is the lack of memory to store all the data structures needed to implement the algorithm. We require our parallel program to push the limit on the size of the input instances that can be handled both in the worst-case as well as in the typical case. (ii) Our programs should handle *implicit* representation of DFA's. I.e., the transition function δ may not be given as a table, but accessible by calling a function. This representation is useful in hardware testing applications in which the DFA is defined by a sequential circuit. (iii) The program should scale well. This means, the total number of operations performed (including the communication and other message exchange operations) by the program should not drastically increase when the number of processors used increases. (iv) The program should exhibit a reasonable speed-up. This means that the same instance should run faster when the number of processors is increased provided that the instance is not too small. Finally, the program should have predictable performance and be transportable across different parallel machine platforms. We also wanted to study this problem as a test case for determining the practical effectiveness of parallel algorithm design techniques. We are not aware of any other parallel program implementation for this problem although many sequential programs have been designed and implemented.

In this paper, we will describe a simple parallel program which satisfies most of the above requirements. In section 2, we will present some basic definitions and describe the main idea behind our parallel program. In section 3, we describe the implementation details and analyze the time and communication complexity of our program. In section 4, we present the experimental performance of our program. We conclude with directions for future work.

2 Definitions and Preliminaries

A finite automaton (DFA) or a finite state system M is a 5-tuple $M = < Q, I, O, \delta, \lambda >$ where Q is a finite set of states, I is a finite set of input symbols, O is a finite set of output symbols, $\delta : Q \times I \rightarrow Q$ is the transition function and $\lambda : Q \times I \rightarrow O$ is the output function. Note that we use the *Mealy machine* model but all our results hold for *Moore machines* as well. Following [11], we will use the convenient abbreviation $q < x >$ to denote $\lambda(q, x)$ and qx to denote $\delta(q, x)$ throughout. If $R \subset Q$ and $x \in \Sigma^*$, we define $\delta(R, x) = \{\delta(r, x) | r \in R\}$. As above, we abbreviate $\delta(R, x)$ by abbreviated Rx. In a similar way, we define $\lambda(R, x)$ and abbreviate it $R < x >$. A string x is said to be a distinguishing string for two states $p, q \in Q$ if $p < x > \neq q < x >$.

Let $M = < Q, I, O, \delta, \lambda >$ be a DFA. The following definitions are with respect to M. Let $A \subseteq Q$. A preset homing sequence for the collection A is a string x such that for any $p, q \in A$, if $p < x > = q < x >$ then $px = qx$. An adaptive homing sequence for the collection A is a tree T whose internal nodes labeled by symbols of I, whose edges are labeled by symbols of O and each of whose leaf is labeled by a state in Q, with the property that if M starts in any state in A and the input at the root is applied and the edge labeled by the output is traversed and the input at the next node is applied etc. then M will be in the state indicated by the label associated with the leaf reached.

The study of homing sequences was initiated by Moore in the classical work [6]. Early work by him and others showed the existence of a homing sequence of length $O(n^2)$ for any minimal DFA M with n states. From now on, we will assume that the input DFA is minimized. A sequential algorithm (taken from [11]) to construct a homing sequence is presented in *HSEQ*. In line 4, the algorithm requires a distinguishing string x for the states ph and qh. We assume that distinguishing strings for all pairs have been already computed and stored in a table. These strings can be obtained by a modified DFA minimization algorithm and the existence of such a string is guaranteed for all pairs of states by the minimality of M. In fact, such a modified minimization algorithm can find the shortest distinguishing string for all pairs of states in time $O(n^2)$. It is well-known (see e.g. [5]) that for any pair of states in an n state DFA, a distinguishing string of length at most $n-1$ exists. It is also not difficult to see that the number of iterations of the *while* loop is bounded by $n-1$ since the quantity $|Q<h>|$ increases by at least 1 after each iteration and $|Q<h>|$ is bounded by n. Thus the length of the output h is bounded by $(n-1)^2$. In the next section, we will show how this algorithm can be implemented efficiently in practice.

procedure *HSEQ*;
  begin
    $h \leftarrow \epsilon$;
    while $\exists\ p, q \in Q$ s.t. $p<h>=q<h>$ and $ph \neq qh$ **do**
    begin
      Let x be a distinguishing string for states ph and qh;
      $h \leftarrow hx$;
      $Q \leftarrow Qh$;
    end
    output(h);
  end

A Sequential Program

We will modify the above algorithm to compute a homing sequence for a subset $X \subseteq Q$ of states. (Recall the definition of a homing sequence *for a given set* X as give in Section 2.) It is not difficult to see that algorithm *SIMPLE_HSEQ* achieves this. We will first briefly describe how the algorithm *SIMPLE_HSEQ* can be implemented as a serial program on a uniprocessor. At any stage, the program maintains a prefix h of the homing sequence and the set $X<h>$ of states reached on h starting from some state in X. It actually maintains a partition of states in $X<h>$ so that two states $p, q \in X<h>$ belong to the same partition if and only if there exist states $p, q \in X$ so that $p<h>=q<h>$ but $ph! = qh$. With this information structure, it is easy to implement the *while* condition. Simply any two states belonging to the same class in the partition can be selected. It is not difficult to see that number of iterations of the *while* loop is bounded by $|X|-1$ and hence the total time complexity of this algorithm is at most $O(n^2|X|)$.

```
procedure SIMPLE_HSEQ(X);
{Sequential algorithm to find a homing sequence for a set X ⊆ Q}
  begin
      h ← ε;
      while ∃ p, q ∈ X s.t. p < h >= q < h > and ph ≠ qh do
      begin
         Let x be a distinguishing string for states ph and qh;
         h ← hx;
         X ← Xh;
      end;
      output(h);
  end.
```

Program for Homing a Subset of Q

A high-level description of the parallel algorithm is as follows. Suppose there are p processors P_1, ..., P_p. The state set $X = Q$ is divided into Q_1, Q_2, ..., Q_p each with roughly the same number of states. All processors P_i find a homing sequence for the states Q_i in parallel, but each processor essentially runs the sequential algorithm described in *SIMPLE_HSEQ*. Then, each processor P_i communicates the string h_i it found to other processors. Based on some conditions, the processors select one of the strings h_i, say h' and output it as part of the final output, and also determine $Q < h' >$, the states reached on input h' from all possible initial states. The set, after some processing, becomes the new set X of states. It is again divided into subsets and assigned to various processors and another cycle is repeated. When the current set X becomes small enough, we will simply continue the algorithm on a single processor. The concatenation of the sequence of strings produced after each round is the final output.

3 Implementation Details

In this section, we will present the implementation details of the program described in Section 2. We will first present the primitive operations of the parallel machine model. The program was designed based on *message passing* style. In this model, each processor will be assigned data and the same program will be implemented on each. But conditional statements can use the processor ID so that different processors can execute different segments of the program. The processors share information by communicating with each other. Given below is a detailed high-level description of our program. The program has been implemented in C using the communication library CMMD of CM-5.

Program *Parallel_Homing*

Input:A DFA $M = < Q, I, O, \delta, \lambda >$. The program uses k processors (k can be specified). The state set Q is partitioned into Q_1, Q_2, ..., Q_k of equal size. Q_i is assigned to the processor P_i.

Output: A homing sequence of M.

Comment: At each stage, a set of states will be assigned to each processor. The collection of all such states is called current_set. Further, each state in the current_set also has a key associated with it, namely the class number in the partition of current_set. In the following, when we denote the pair (state label, partition number) as the *extended state label.*

Begin

Initialize the current_set to $Q0$, where 0 is some symbol in Σ; Initialize the key associated with state p as $\lambda(p, 0)$. Initialize output to empty string;

while (the size of current_set $> k$) **do** in parallel

1. Processor P_i finds a homing sequence h_i for Q_i using the (sequential) algorithm *SIMPLE_HSEQ*.
2. Processors communicate h_i to a single processor, say P_1 which selects one string h' from h_i's using some criteria. (See the discussion below.)
3. P_1 sends h' to all processors. h' is appended to the output list.

4. Processor P_i computes the set Q_ih' of states reached on input h' and the output strings produced $Q_i < h' >$ from states Q_i. (This is done sequentially by each processor.) current_set = $\bigcup Q_ih'$. At this point, processor P_i holds the states Q_ih' and the partition label for state ph' is assigned as the partition label of p. (I.e. the old label is retained.)
5. Duplicates are removed from current_set. This is done as follows: First current_set is globally sorted using the extended state label as the key. This brings together identical reached states belonging to the same equivalence class of the old partition. Processor P_i scans Q_ih' and removes duplicates. Then, adjacent processors communicate and remove shared duplicates, if any. Let the resulting set be renamed current_set.
6. The partition Q_jh' is refined so that two states p, q belong to the same equivalence class only if there exist $p', q' \in Q_j$ such that $p' < h' >= q' < h' >$ and $p = p'h' \neq q'h' = q$. This results in a new partition label (which could be the old label) for each state in the current set. (This step is elaborated further below.)
7. Singleton members of the new partition are deleted. current_set is set to the resulting collection.
8. current_set is divided into Q_1, ..., Q_k.

end_while

Transfer the current_set to a single processor.

Call the sequential program to find a homing sequence h for current_set; Append h to the output list.

end.

We now analyze the main computational steps and the communication operations of the algorithm *in the worst-case.* We will analyze these two aspects of the algorithm for each iteration. To get the total estimate, we need to multiply it by the number of iterations. We do not know a tight bound on the number of iterations. The obvious bound of n (the number of states) seems too pessimistic, even in the worst-case. But on randomly generated test cases, this number turns out to be a small constant. In what follows, we will present the computational cost and communication cost of each step separately.

Computational Cost of the Program

Since each processor is assigned $\frac{n}{k}$ states, step (1) takes $O(\frac{n^3}{k})$ operations.

In step (2), we can choose any one h_i using some heuristic. For example, we may pick the shortest one since we want a short overall output string. But the rationale for selecting the longest one is equally compelling. The longer ones may home in many states belonging to the other classes of the partition. For random DFA's, experiments indicate that longer h_i's are better. But if h_i is too long, sorting of outputs becomes expensive. So in our implementation, we choose the longest h_i such that the output of each state can be packed into an integer. We then can use a standard sorting subroutine. The cost of step (2) is $O(k)$ comparison operations.

In step (4), processor P_i computes the sets $Q_i h'$ and $Q_i < h' >$. This is done in a straight-forward manner in $O(\frac{n}{k} \cdot |h'|)$ time.

Step (5) is aimed at grouping identical states together in order to remove duplicates. This is done in $O(\frac{n}{k} \cdot \log n)$ time by parallel sorting. Sorting is carried out within each segment of the partition by using the extended state label as the key.

Step (6) is implemented as follows. A global sorting is performed using the pair ($q < h' >$, partition label) as the key for each $q \in current\_set$. After sorting, every adjacent key pair is examined and a split is made at places where the output strings differ. The complexity for this step is $O(\frac{n \ log \ n}{k})$.

Step (7) is implemented by scanning the listed sorted in Step (6) locally by each processor. This is easily seen to take at most $O(\frac{n}{k})$ steps.

In Step (8), the number of remaining states is computed in $O(\log k)$ time by a prefix summation. The redistribution of *current_set* is part of the parallel sorting step (6).

The dominant cost in the above is the cost of step (1). If the number of iterations of the main loop of the parallel program is not large, then we would have a speed-up close to p. Although this is not true in the worst-case, it appears to be true when the input instances are randomly generated. In any case, the goal of our program design is to achieve a good performance *in practice* and so the best test of our design is the experimental performance. This is discussed in the next section.

Communication Cost of the Program

The most desirable feature of our program is that it uses very few communication operations. The communication steps are (2), (3) and the communication cost for two parallel sorting steps. The dominant communication cost is for the sorting step. Many communication efficient programs have been designed and we can use any of them. Our implementation is based on a program due to [12].

4 Experimental Results on CM-5

We ran the program presented above for some DFA's of different sizes on a CM-5 with 32 nodes. Each node has 4 vector units. Since our main goal is to demonstrate that our program works well *in practice*, the issue of creating good test data is critical. It appears that very little work has been done in collecting good bench marks and test data comprising of generic DFA's. Specific applications such as hardware fault-detection have some bench marks (e.g. ISCAS'89). But these circuits have a large number of outputs and hence are not particularly difficult instances. The worst-case DFA's with n states (those in which the shortest homing sequence is of length $n(n-1)/2$) are known [5]. A standard way to create test data is to generate instances randomly. The limitations of random models are well-known. (Even for some NP-hard problems, there are simple algorithms that solve almost all instances very efficiently.) But in the absence of any real data, random models provide a reasonable alternative. In the case of automata, two random graph models have been studied, mainly by Russian mathematicians. For want of better terminology, we will call them random model and the uniform random model. In the random model, a random instance with n states over input alphabet I and output alphabet O is produced by assigning $\delta(q, a)$ a state with probability $1/n$ and by assigning $\lambda(q, a)$ an output symbol with probability $1/|O|$. The uniform model is defined as follows: δ is selected arbitrarily (i.e., an adversary can select it) but the output function is chosen randomly. A number of interesting results about homing sequences are known under such models. For example, almost all DFA's with n states have a homing sequence of length $O(log\ n)$ under random model; almost all DFA's with n states have a *local* homing sequence of length $O(log\ n)$ for each state under uniform random model. (A local homing sequence for a state p is a string x such that based on the output, we can determine if the state reached is px.)

We propose to use a variety of test cases (including all the ones presented above) to test our program. Here we will present the test results only for two cases: (i) randomly created DFA and (ii) DFA created using uniform random model in which the δ function is chosen as in the worst-case DFA's. The random instances were created as follows. The input and output alphabet were both of size 2. The size of the DFA was selected in the range from a few thousands to half a million.

A DFA with equivalent states may not have a homing sequence. So we assume that DFA's to be homed are minimized and thus have no equivalent states. Actually, our program still works if the input DFA is not minimized, but it will not be efficient on such instances. A random DFA generated as above may have equivalent states, but experiments show that this is unlikely.

Figure 2 shows the running time in seconds for the program for the randomly generated DFA. The time to run the message passing program was taken as elapsed CPU time of one node (provided by the CM-5 CMMD library), but not including the time taken by the sequential algorithm in the last step. This timing for parallel part is reliable since all nodes are synchronized at the beginning and end of the program (and also in each iteration).

To study the scalability of the program, we chose a DFA with 256K states and find homing sequence for it by using a CM-5 configured with 32, 64, 128 and 256 nodes. Figure 3 shows the results of running time. The program seems to scale well when the number of nodes is smaller than 256. It may also have good scalability for more nodes if the input size becomes large.

We also tested our program with DFA's generated using uniform random model. Here, the state transition of the DFA is fixed arbitrarily and only the

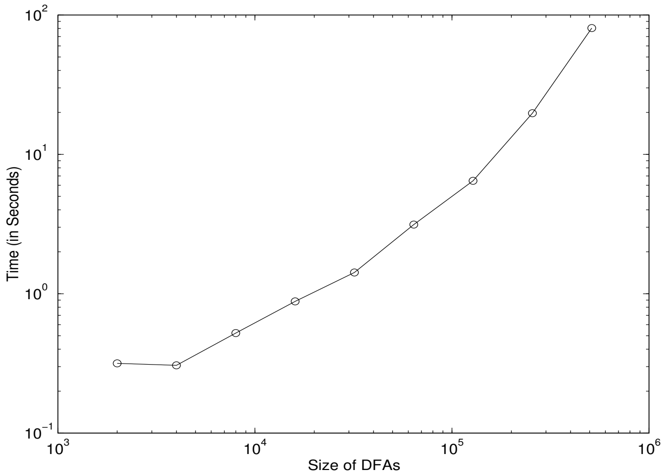


Figure 2: Running Time vs. Problem Size of the Parallel Program

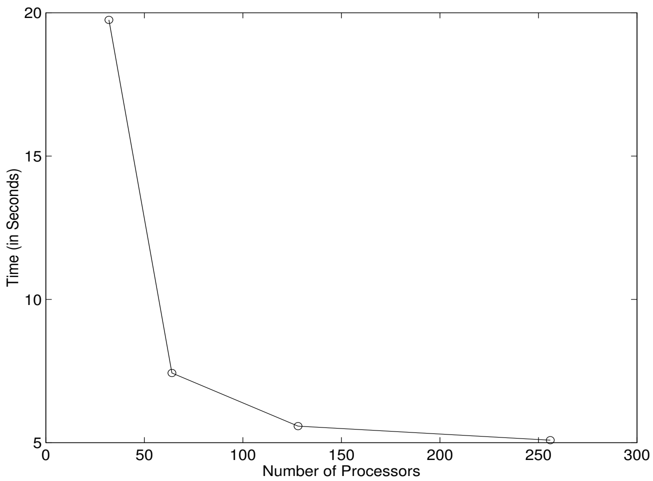


Figure 3: Scalability of the Parallel Program

output function is chosen randomly. The most difficult DFA (i.e., the one for which the program takes maximum time) under this model seems to be the one obtained by fixing the transition function as that for the worst-case DFA. When we do this, our program's elapsed CPU time grows significantly larger than that for the random DFA's tested above. Currently our program cam handle these DFA's with up to a few thousand states. Going beyond 10^5 in this model may not be easy since in these cases, the output length of even the shortest homing sequence can be 10^{10}.

5 Conclusions

In this paper, we presented a practical parallel program implementation for the homing sequence problem. Until recently, implementing parallel programs with good performance was feasible only for well-structured problems such as vector or matrix computations. Data parallel style offered a simple and effective solutions for such problems. Recent advances in the design of parallel programming languages and other system tools, combined with coarse-grained parallel machines are extending the scope of parallel programs to nonnumerical problems. Our goal is to design a library of parallel programs for all the basic problems related to finite automata. So far we have considered two problems in some detail. They are: DFA minimization [9] and the homing sequence problem. These two problems have offered some understanding about the nature of parallel computation for such applications. There is no doubt that parallel programming will become standard in the near future. MPI running on a network of workstations offers a good speed-up at modest hardware cost. Thus it is of interest to develop parallel program library for all fundamental problems. Our preliminary work along these lines appears to be promising.

There are a number of directions in which this work can be extended. We will list some of them below.

(i) Design sequential or parallel program for the adaptive homing sequence. These are decision trees in which the input symbol to be applied next can depend on the outputs to the previous input symbols. In previous work (e.g. [11]) it has been noted that the adaptive sequences are relatively shorter (although not in the worst-case).

(ii) Design sequential or parallel programs for synchronizing, distinguishing or testing sequences. These are closely related to homing sequences and are also of interest in automata testing and inference problems.

(iii) Issues of programming language and environment. Recently [2] has argued the effectiveness of functional language NESL in parallel program design. We would like to look at NESL or similar languages for implementing automata. Also of interest is to develop a parallel version of Grail or similar programs.

References

[1] M. A. Bender and D. K. Slonim, "The power of team exploration: Two robots can learn unlabeled directed graphs", *35th Annual IEEE Symposium on Foundations of Computer Science*, pp. 75-85 (1994).

[2] G. Blelloach and J. Harwick, "Class Notes: Programming Parallel Algorithms", *Technical Report CMU-CS-93-115, Carnegie Mellon University.*

[3] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. Schapire and L. Sellie, "Efficient Learning of typical automata from random walks", *Proc. of 25th Annual ACM Symposium on Theory of Computing, pp. 315-324 (1993).*

[4] G. Holzmann, *Design and validation of protocols*, Englewood Cliffs, NJ, Prentice-Hall (1990).

[5] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill Publishers Inc. Second Edition (1978).

[6] E. F. Moore, "Gedanken-experiments on sequential machines", pp. 129-153, *Automata Studies*, Ed: McCarthy and Shannon, Princeton University Press, Princeton, NJ (1956).

[7] I. Pomeranz and S. Reddy, "Application of homing sequences in the fault-detection of sequential circuits, *IEEE Transactions on Computers*, (1994).

[8] B. Ravikumar, "Performance bounds of algorithms for testing automata", (in preparation).

[9] B. Ravikumar and X. Xiong, "A parallel algorithm for minimization of finite automata", in the *Proceedings of the 10th International Parallel Processing Symposium* (IPPS '96), Honolulu, Hawaii, April 1996, pages 187-191.

[10] B. Ravikumar and X. Xiong, "Randomized parallel algorithms for the homing sequence problem", to appear in *the International Conference on Parallel Processing* (ICPP'96), Bloomingdale, Illinois, August 1996.

[11] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences", *Information and Computation*, vol. 103, pp. 299-347, (1993).

[12] 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.