January 1993

Fast Parallel Deterministic and Randomized Algorithms for Model Checking

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Sanguthevar Rajasekaran
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-09.

NOTE: Page 2 is missing.

Fast Parallel Deterministic and Randomized Algorithms for Model Checking

Abstract

Model checking is a powerful technique for verification of concurrent systems. One of the potential problems with this technique is state space explosion. There are two ways in which one could cope with state explosion: reducing the search space and searching less space. Most of the existing algorithms are based on the first approach.

One of the successful approach for reducing search space uses Binary Decision Diagrams (BDDs) to represent the system. Systems with a large number of states (of the order of 5 x 10") have been thus verified. But there are limitations to this heuristic approach. Even systems of reasonable complexity have many more states. Also, the BDD approach might fail even on some simple systems. In this paper we propose the use of parallelism to extend the applicability of BDDs in model checking. In particular we present very fast algorithms for model checking that employ BDDs. The algorithms presented are much faster than the best known previous algorithms. We also describe searching less space as an attractive approach to model checking. In this paper we demonstrate the power of this approach. We also suggest the use of randomization in the design of model checking algorithms.

Comments

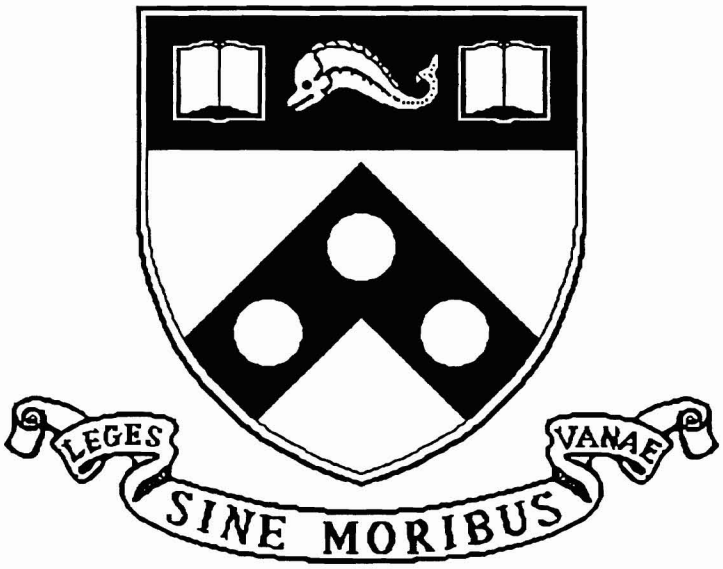
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-09.

NOTE: Page 2 is missing.

Fast Parallel Deterministic and Randomized Algorithms for Model Checking

MS-CIS-93-09
GRASP LAB 342

Insup Lee
Sanguthevar Rajasekaran



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

January 1993

Fast Parallel Deterministic and Randomized Algorithms for Model Checking*

Insup Lee and Sanguthevar Rajasekaran
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

January 26, 1993

Abstract

Model checking is a powerful technique for verification of concurrent systems. One of the potential problems with this technique is state space explosion. There are two ways in which one could cope with state explosion: reducing the search space and searching less space. Most of the existing algorithms are based on the first approach.

One of the successful approach for reducing search space uses Binary Decision Diagrams (BDDs) to represent the system. Systems with a large number of states (of the order of 5×10^{20}) have been thus verified. But there are limitations to this heuristic approach. Even systems of reasonable complexity have many more states. Also, the BDD approach might fail even on some simple systems. In this paper we propose the use of parallelism to extend the applicability of BDDs in model checking. In particular we present very fast algorithms for model checking that employ BDDs. The algorithms presented are much faster than the best known previous algorithms. We also describe searching less space as an attractive approach to model checking. In this paper we demonstrate the power of this approach. We also suggest the use of randomization in the design of model checking algorithms.

1 Introduction

One of the most successful techniques for automatic verification of concurrent systems has been model checking, which was developed by Clarke, Emerson and Sistla [9]. Model checking determines whether a finite state system satisfies a property specified as a formula in propositional branching time temporal logic by computing the set of states that satisfies the formula. This method has been implemented and used to successfully prove the correctness of a large class of hardware circuits.

*This research was supported in part by ONR N00014-89-J-1131, DARPA/NSF CCR90-14621 and ARO DAAL 03-89-C-0031.

1.3 Organization of this Paper

The rest of the paper is organized as follows. Section 2 contains the definition of BDDs and introduces parallel computational models. Sections 3 and 4 describe and analyze parallel algorithms for various operations on BDDs. In Section 5, we present parallel model checking algorithms that employ BDDs and their time complexities. Section 6 has a very simple encoding scheme that is optimal with respect to memory usage. In Section 7 we show how one could use probablism to search less space. In Section 8 we demonstrate the power of randomization in algorithms design. In particular we present a randomized parallel algorithm for the Coarsest Partitioning Problem (with a single function) that is better than the previously best known algorithm. Finally, Section 9 concludes this paper.

2 Preliminaries

2.1 Binary Decision Diagrams

A Binary Decision Diagram is a canonical representation of a boolean formula. Its structure is an acyclic graph. Each node in the BDD is labeled with a variable and has two children corresponding to the two values this variable can take (viz. 0 and 1). There is a total order imposed on the occurrence of variables along any path starting from the root and ending in a leaf. The leaves of the BDD are labeled with a 0 or a 1. Any path starting from the root will end up in a leaf labeled 1 if and only if under the corresponding assignment to the variables along the path, the formula has a value 'true'. To check if a given assignment to variables satisfies the boolean formula, one traverses a path dictated by the assignment from the root of the BDD until a leaf node is reached. This leaf node gives the value of the formula under the given assignment. It has been shown that there is a unique minimal BDD corresponding to any boolean formula under a given ordering of variables [3].

In Figure 1, an example is given. This BDD represents the formula $\bar{a} + bc$, with $a < b < c$. For instance, to check if $a = 1, b = 0, c = 1$ will satisfy the formula we start from the root and traverse the corresponding path to reach the second leaf (from left) which is labeled with a 0.

2.2 Parallel Machine Models

A large number of parallel machine models have been proposed. Some of the widely accepted models are: 1) fixed connection machines, 2) shared memory models, 3) the boolean circuit model, and 4) the parallel comparison trees. Of these 1) and 2) are the most popular. The *time complexity* of a parallel machine is a function of its input size. Precisely, time complexity is a function $g(n)$ that is the maximum over all inputs of size n of the time elapsed when the first processor begins execution until the time the last processor stops execution.

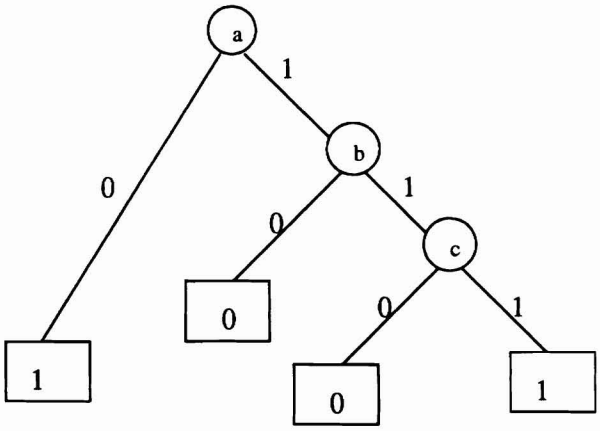


Figure 1: A Binary Decision Diagram

A fixed connection network is a directed graph $G(V, E)$ whose nodes represent processors and whose edges represent communication links between processors. Usually we assume that the degree of each node is either a constant or a slowly increasing function of the number of nodes in the graph. Fixed connection networks are supposed to be the most practical models. The Connection Machine, Intel Hypercube, ILLIAC IV, Butterfly, etc. are examples of fixed connection machines.

In shared memory models (also known as PRAMs), a number (call it P) of processors work synchronously communicating with each other with the help of a common block of memory accessible by all. Each processor is a random access machine. Every step of the algorithm is an arithmetic operation, a comparison, or a memory access. Several conventions are possible to resolve read or write conflicts that might arise while accessing the shared memory. EREW PRAM is the shared memory model where no simultaneous read or write is allowed on any cell of the shared memory. CREW PRAM is a variation which permits concurrent read but not concurrent write. And finally, CRCW PRAM model allows both concurrent read and concurrent write. Read or write conflicts in the above models are taken care of with a priority scheme.

The parallel run time T of any algorithm for solving a given problem can not be less than $\frac{S}{P}$ where P is the number of processors employed and S is the run time of the best known sequential algorithm for solving the same problem. We say a parallel algorithm is *optimal* if it satisfies the equality: $PT = O(S)$. The product PT is referred to as *work done* by the parallel algorithm.

The model assumed in this paper is the PRAM. Though a PRAM is supposed to be impractical, it is easy to design algorithms on this model and usually algorithms developed for this model can be easily mapped on to more practical models. Also there is a simulation algorithm that will map any PRAM algorithm into an algorithm for the hypercube network (the CM being one) with at the most a logarithmic factor of slow down [15]. Thus all the time bounds mentioned in this paper will apply to the CM if multiplied by a logarithmic factor.

2.3 Some Useful Lemmas

In this section we state some results which will prove useful in interpreting results presented in this paper.

Lemma 2.1 *[2] If W is the total number of operations performed by all the processors using a parallel algorithm in time T, we can simulate this algorithm using P processors such that the new algorithm runs in time $\lfloor \frac{W}{P} \rfloor + T$.*

As a consequence of the above Lemma we can also get:

Lemma 2.2 *If a problem π can be solved in time T using P processors, we can solve the same problem using P' processors (for any $P' \leq P$) in time $O\left(\frac{PT}{P'}\right)$.*

Definition. Given a sequence of numbers $k_1, k_2, \ldots, k_n$, the problem of *prefix sums computation* is to output the numbers $k_1, k_1 + k_2, \ldots, k_1 + k_2 + \ldots + k_n$.

The following Lemma is a folklore [12]:

Lemma 2.3 *Prefix sums of a sequence of n numbers can be computed in $O(\log n)$ time using $\frac{n}{\log n}$ EREW PRAM processors.*

3 A Fast Parallel Algorithm for Reducing a BDD

One of the frequently used operations in the model checking algorithm of [6] is reducing a BDD. The problem of reducing a BDD is to take as input an arbitrary BDD and reduce it into an equivalent BDD with as few states as possible. Bryant [3] has shown that there is a unique minimal BDD corresponding to any boolean formula under a given ordering for the variables. He has also presented an efficient sequential algorithm for reduction. In this section we present an efficient parallel algorithm for BDD reduction. We also analyze the sequential and parallel complexity of Bryant's algorithm.

3.1 Bryant's Algorithm

Let $G(V, E)$ be the given BDD for a boolean formula on n variables that has to be reduced. Then Bryant's algorithm can be shown to have a run time of $O(n + |V| \log |V|)$ (even though no such bound is explicitly mentioned in [3]). A brief description of this algorithm is in order to be able to follow the rest of the discussion. His algorithm processes nodes level by level starting from the leaves and proceeding toward the root. The leaves of G are associated with values (0 or 1), whereas the rest of the nodes are associated with variables. The algorithm will assign unique labels to the nodes such that all the equivalent nodes get the same label. In the case of leaves, all the nodes with a 0 value get the label 1 and the nodes with a 1 value get the label 2.

As far the rest of the nodes two simple rules are used to label them. If v is any node in G, let $LEFT(v)$ and $RIGHT(v)$ stand for the left and right children of v respectively. Let $LABEL(v)$ denote the label assigned to v. The rules are: 1) If there are two nodes v_1 and v_2 such that $LABEL(LEFT(v_1)) = LABEL(LEFT(v_2))$ and $LABEL(RIGHT(v_1)) = LABEL(RIGHT(v_2))$, then v_1 and v_2 should get the same label; 2) If $LABEL(LEFT(v)) = LABEL(RIGHT(v))$ for any node v in G, then we should set $LABEL(v) = LABEL(LEFT(v))$, since this implies that the node v is redundant and can be eliminated.

Assume inductively that all the nodes in level $i+1$ or below of G have been processed (i.e., each node in these levels has been assigned a label). Now we describe how to label the nodes in level i. Let v be an arbitrary node in level i. If $LABEL(LEFT(v)) = LABEL(RIGHT(v))$ we immediately set $LABEL(v) = LABEL(LEFT(v))$ and the processing of node v is over. If not, we create a tuple $(LABEL(LEFT(v)), LABEL(RIGHT(v)))$ corresponding to this node. For all the nodes in level i that have not been assigned a label yet, we take their tuples and sort these tuples in lexicographical order. The effect of this sorting is to group all the nodes with the same tuple together. Clearly all the nodes with the same tuple should be assigned the same label.

Let m_i be the number of distinct tuples at this level. Let n_i and N_i denote the largest label assigned to any node at level i and the number of nodes at level i respectively, for any $i \leq n$. Nodes at level i are labeled with integers in the range $n_{i+1}+1, n_{i+1}+2, \ldots, n_{i+1}+m_i = n_i$. This completes the processing of all the nodes at level i. Here is a detailed version of the algorithm:

Algorithm Reduce($G(V,E)$);

(* LIST is a list of lists. LIST(i) is a list of nodes at level i, for $1 \leq i \leq (n+1)$.
LABEL is an array such that $LABEL(v)$ is the label assigned to node $v \in V$.
*)

for each node v in LIST($n+1$) *do*
(* Process the leaves *)

if the node v has a value 0 *then* LABEL(v) := 1 *else* LABEL(v) := 2;
nextlabel:=3;

for i := n *downto* 1 *do* (* Process the nodes in level i, $n \geq i \geq 1$ *)

TEMP:= $\emptyset$; (* TEMP is the list of nodes at level i that should get new labels *)
for each node v in LIST(i) *do*
if LABEL(LEFT(v))=LABEL(RIGHT(v)) *then*
LABEL(v) := LABEL(LEFT(v))
else

add v to TEMP with a corresponding 'key' of (LABEL(LEFT(v)), LABEL(RIGHT(v)));

Sort the nodes in TEMP with respect to their keys in lexicographical order;

Scan through this sorted list and assign labels to nodes starting from nextlabel, such that nodes with the same key get the same label;
nextlabel := nextlabel $+m_i$;
(* m_i is the number of distinct keys in TEMP *)

Clearly, the amount of time needed to process nodes at level i is dominated by sorting of the tuples. If there are N_i nodes at level i, this sorting can be done in $O(N_i \log N_i)$ time [1]. Therefore the run time of the algorithm is $\sum_{i=1}^{n} O(N_i \log N_i) = O(|V| \log |V| + n)$. Thus we get the following

Theorem 3.1 *Bryant's algorithm for BDD reduction runs in time $O(|V| \log |V| + n)$ where V is the number of nodes in the BDD and n is the number of variables in the formula.*

3.2 The Parallel Algorithm

In [14] Kimura and Clarke present a parallel algorithm for BDD reduction. This algorithm is based on the observation that one could think of a BDD as a deterministic finite state automaton (DFA). Though no explicit mention is made in the paper about the complexity of their algorithm, it is straightforward to realize that this algorithm can be implemented on a CREW PRAM in time $T = O(n + n \log \frac{|V|}{n})$ using $\frac{|V| \log |V|}{T}$ processors. However, their algorithm makes the following assumption: The nodes of the BDD are not explicitly associated with variables; the level of any node uniquely determines the variable associated with it. This assumption may not hold always. For instance in Figure 1, the leftmost leaf and the node associated with b are at the same level, but they do not correspond to the same variable. Thus in order to make their algorithm applicable to all inputs, one has to perform some preprocessing. Later in this section we demonstrate how to accomplish this preprocessing in parallel.

The same time bound can be achieved when we implement the Bryant's algorithm (described in the previous section) on a CREW PRAM. In this parallel version of Bryant's algorithm no assumption is needed on the structure of the input BDD. An analysis of the parallel run time of Bryant's algorithm follows: To process the nodes at level i, the tuples can be created (if there is a need) in time $O(1)$ using N_i processors. Total work done is $O(N_i)$. The next step is to sort the tuples (there can be at the most N_i tuples). This can be done in $O(\log N_i)$ time using N_i processors [12]. Total work done for sorting is $O(N_i \log N_i)$.

Thus the parallel run time of Bryant's algorithm is $\sum_{i=1}^{n} O(\log N_i)$ which is $O(n + n \log \frac{|V|}{n}) = T$, say. Total work done (i.e., number of operations) in all the n levels is $\sum_{i=1}^{n} O(N_i + N_i \log N_i) = O(|V| \log |V|)$.

Therefore, using Brent's Lemma 2.1, this algorithm runs in time $O(n + n \log \frac{|V|}{n})$ using $\frac{|V| \log |V|}{T}$ processors. Thus we get:

Lemma 3.1 *Bryant's algorithm can be implemented on a CREW PRAM in time* $T = O(n + n \log \frac{|V|}{n})$ *using* $\frac{|V| \log |V|}{T}$ *processors. Here* $|V|$ *and* n *stand for the number of nodes and the number of variables in the BDD.*

Therefore as far as the work done is concerned, Bryant's algorithm can be implemented optimally on the CREW PRAM. But the question is can we reduce the run time further keeping the work done the same or nearly the same? Ideally we would like to have a run time which is only a polynomial in the logarithms of $|V|$ and n. Next we present an algorithm for BDD reduction which runs in $O(\log^2 |V|)$ time using only M_V processors, where M_V is the number of processors needed to reduce a DFA with $|V|$ nodes into minimal form in $O(\log^2 |V|)$ time. We also first make the same assumption about the structure of the input BDD as in [14], namely that the level of any node uniquely determines the variable associated with the node. We call a BDD that satisfies this assumption as a 'BDD in restricted form' from hereon. Later we show how to preprocess the input in parallel such that the resultant BDD will satisfy this assumption. Our algorithm will also be based on the fact that one could associate a BDD in this restricted form with a DFA.

A DFA A is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the set of states of A, Σ is the alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the state transition function, q_0 is the start state, and F is the set of accepting states. A is said to accept a string $a_0, a_1, \ldots, a_n$ (where $a_i \in \Sigma$ for each i) if and only if $\delta(\ldots \delta(\delta(q_0, a_0), a_1) \ldots, a_n) \in F$.

A BDD B in restricted form can be associated with a DFA A in the following manner: Σ for A is $\{0, 1\}$. Q is the set of nodes of B. The start state of the automaton A is the root of B. The only accepting states of A will be the leaves that have a 1 in them. There is a one-to-one correspondence between the nodes of B and the states of A. For any state s in A, let CORR(s) stand for the corresponding node in B. If δ is the transition function of A, then, for any state s in A, $\delta(s, 0) = t$ ($\delta(s, 1) = t$) if and only if CORR(t) is the left (right) child of CORR(s). This correspondence simply means that the transition graph of A is the same as the DAG representing the BDD. Thus the task of converting a BDD into a DFA is trivial.

Also notice that if B is a BDD on n variables and if A is the corresponding DFA, then the following holds: If a binary string of length n is accepted by A, then the corresponding assignment to variables will satisfy the boolean formula that B represents.

There are many parallel algorithms in the literature for minimizing a DFA. One of the fastest algorithms has been given by Já Já and Kosaraju [13]:

Theorem 3.2 *If* G *is a DFA with* n *states, we could obtain the minimal form of* G *in* $O(\log^2 n)$ *time on the EREW PRAM using* n^4 *processors.*

The above algorithm can be used in the reduction of BDDs: 1) Given a BDD B, convert it into a DFA A in a straight forward way (as explained above). This can be done in $O(1)$ time using $|V|$ CREW PRAM processors; 2) Minimize this DFA using any good algorithm. This will take $O(\log^2 |V|)$ time using M_V CREW PRAM processors; and finally 3) Convert the minimal DFA into a BDD. This can again be accomplished in time $O(1)$ using $|V|$ processors. The total work done in this parallel algorithm is $O(M_V \log^2 |V|)$. It is an open problem if one could reduce the work done to $O(|V| \log |V|)$ keeping the time bound the same. The above algorithm together with an application of Brent's Lemma 2.2 yields the following

Theorem 3.3 *A BDD with $|V|$ nodes can be reduced in time $O\left(\frac{M_V \log^2 |V|}{P} + \log^2 |V|\right)$ using P processors (as long as $P \leq M_V$).*

3.3 Converting a BDD into Restricted Form

The algorithm given in the previous section for BDD reduction assumes that the BDD B is in restricted form, i.e., the level of any node uniquely determines the associated variable. But usually in practice we can not expect the input BDD to be in this form. In this section we show how to convert an arbitrary BDD into restricted form.

The idea is to ensure that every path from the root to a leaf is of the same length. Consider a BDD B on n variables $x_1, x_2, \ldots, x_n$, with a variable ordering of $x_1 < x_2 \ldots < x_n$. We may have to introduce redundant nodes in the BDD in order to bring it to restricted form. (We say a node is redundant if on any input transition from this node leads to the same node.) For instance if there is a node associated with x_1 and if one of its children is associated with x_4, then we introduce two nodes associated with x_2 and x_3 along the path from x_1 to x_4. From x_2 there is a transition to x_3 on either input, and also there is a transition from x_3 to x_4 on either input.

In general, for any $i < j$, if a node associated with x_i has a child associated with x_j, the path from x_i to x_j is completed with redundant nodes with the variables: $x_{i+1}, x_{i+2}, \ldots, x_{j-1}$. Without loss of generality assume that the nodes of B are named $1, 2, \ldots, |V|$. A detailed description of the parallel algorithm follows:

Algorithm Restrict($G(V, E)$);

(* N_1 and N_2 are arrays of size $|V|$. For each node $v \in V$, $N_1(v)$ ($N_2(v)$) is the number of nodes to be introduced between v and its left (right) child. *)

for $i := 1$ *to* $|V|$ *do*

Let x_p, x_q, x_r be the variables associated with i, its left child and its right child respectively;
$N_1(i) := q - p - 1$; $N_2(i) := r - p - 1$;

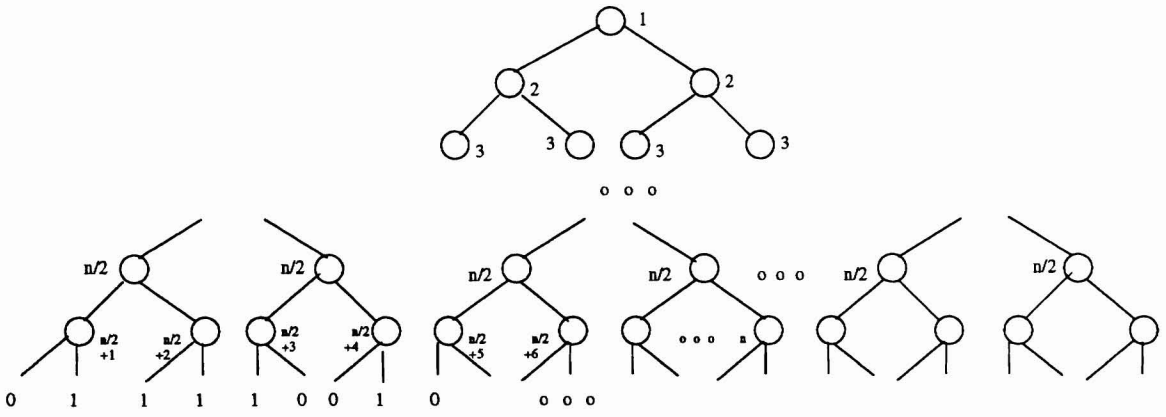


Figure 2: A bad input for algorithm Restrict

Compute the prefix sums of $N_1(1), N_1(2), \ldots, N_1(|V|), N_2(1), N_2(2), \ldots, N_2(|V|)$;
Let $N = \sum_{l=1}^{|V|}(N_1(l) + N_2(l))$;
If we have N processors we could create all the redundant paths in $O(1)$ time, each processor being in charge of creating a single redundant node; Realize that the scheduling problem of deciding which processor should create which node is solved by the above prefix sums computation.

Analysis. We could compute all the $N_1(i)$'s and $N_2(i)$'s in $O(1)$ time given $|V|$ processors. Prefix sums can be computed in $O(\log|V|)$ time using $\frac{|V|}{\log|V|}$ processors (cf. Lemma 2.3). Total work done in these two steps is $O(|V|)$. Finally, creation of redundant nodes can be completed in $O(1)$ time given N processors. Total work done in this step is $O(N)$. Thus applying Lemma 2.1 we get the following

Theorem 3.4 *An arbitrary BDD can be converted into restricted form in time* $O(\log|V| + \frac{N+|V|}{P})$ *using* P *processors.*

An interesting question is how large can N be? Clearly an upper bound on N is $O(|V|n)$. In fact there are examples for which N could be $\Omega(|V|n)$. A description of one such BDD B follows: Say B is on n variables. Up to level $\frac{n}{2}$, B is a complete binary tree. There are only two more levels. Each node at level $\frac{n}{2}$ has two children. These children are labeled (from left to right) with $x_{\frac{n}{2}+1}, x_{\frac{n}{2}+2}, \ldots, x_n$ in a cyclic fashion. The leaves have arbitrary labels (from $\{0,1\}$). See Figure 2.

The number of nodes in the BDD is $2^{\frac{n}{2}+2} - 1$. Number of redundant nodes that should be introduced in order to convert B into restricted form is $\geq 2^{\frac{n}{2}}(\frac{n}{2} - 1)$, implying that the size of B in restricted form is $\Omega(|V|n)$.

We could perform the following preprocessing in the algorithm Restrict, which may be helpful on certain inputs: Search through B to identify variables that do not occur in the label of any node

or which occur only in redundant form. We could reduce the number of levels in the output BDD accordingly. Identifying such variables can be done for example by sorting the nodes with respect to their labels. This will take deterministic time $O(\log|V|)$ on a $|V|$-processor CREW PRAM [12], or randomized $O(\log|V|)$ time using $\frac{|V|}{\log|V|}$ CRCW PRAM processors [18].

4 Other Operations on BDDs

In applying BDDs to model checking we must be able to perform the following operations on BDDs: 1) Boolean AND; 2) Boolean OR; and 3) Boolean NOT. Other operations such EXOR, existential quantification, etc. can be expressed as a sequence of the above three elementary operations. Thus we will focus our attention on these three operations.

Kimura and Clarke make use of the DFA representation of BDDs for the simple reason that standard algorithms available for DFAs manipulation can then be employed. If B_1 and B_2 are any two BDDs, let A_1 and A_2 be their corresponding DFAs. The boolean AND of the two BDDs can be constructed by intersecting A_1 and A_2. Similarly the boolean OR and the boolean NOT correspond to the union, and the complement operations in the DFAs domain.

In this section we show how to perform the AND of two given BDDs. The other boolean operations can be handled along the same lines. Elegant sequential and parallel algorithms are given in [14]. Here we give a very simple parallel algorithm which is much faster than that in [14]. If there are N_1 states in A_1 and N_2 states in A_2, then this algorithm takes constant time to construct the product automaton using $N_1 \times N_2$ CREW PRAM processors. But the resultant product automaton may not be minimal. We then apply the algorithm of Lemma 3.3. This minimization will take $O(\log^2(N_1N_2))$ time using $M_{N_1N_2}$ EREW PRAM processors.

Let $q_{1,1}, q_{1,2}, \ldots, q_{1,N_1}$ be the states of A_1 and $q_{2,1}, q_{2,2}, \ldots, q_{2,N_2}$ be the states of A_2. The only possible states of the product machine are tuples of the form $(q_{1,i}, q_{2,j})$ where $q_{1,i}$ is any state of A_1 and $q_{2,j}$ is any state of A_2 at the **same level**. The arcs going out of any such node in the product can also be determined easily. For instance if there is an arc from $q_{1,i}$ to $q_{1,k}$ on input 0 and there is an arc from $q_{2,j}$ to $q_{2,l}$ on the same input, then there will be an arc from $(q_{1,i}, q_{2,j})$ to $(q_{1,k}, q_{2,l})$ in the product machine on input 0. In general, the arcs going out of any such node can be determined in $O(1)$ time using a single processor. A description of the algorithm follows: For any state s in A_1, IN1(s) is a list of incident edges and OUT10(s), OUT11(s) arrays of outgoing edges (on inputs 0 and 1 respectively). Similarly define IN2(), OUT20(), OUT21() for A_2, and IN3(), IN40(), and IN41() for the product automaton. Each node in the product automaton is labeled as $\langle i, j\rangle$ for $1 \leq i \leq N_1$ and $1 \leq j \leq N_2$. A processor is in-charge of each such node. The corresponding processor will also be named with the tuple $\langle i, j\rangle$.

Algorithm Product(A_1, A_2);

type

edge=record
node : the other end of the edge; input : 0 .. 1;

end;

var

IN1, OUT1 : array[1..N_1] of list of edge;
IN2, OUT2 : array[1..N_2] of list of edge;
IN3, OUT3 : array[1..N_1, 1..N_2] of list of edge;

for $1 \leq i \leq N_1$; $1 \leq j \leq N_2$ processor $\langle i, j \rangle$ *does* in parallel:

if the level of node i in A_1 and the level of j in A_2 are not the same *then quit*;
Let the transition state from i on input 0 be ℓ (this information is obtained from OUT1()) and the transition state from j be q on the same input;
Draw an edge from $\langle i, j \rangle$ to $\langle \ell, q \rangle$, i.e., set OUT30$[i, j]$.node:= $\langle \ell, q \rangle$;
Let the transition state from i on input 1 be r and the transition state from j be t on the same input;
Set OUT31$[i, j]$.node:= $\langle r, t \rangle$;

(* Notice that by now, the product automaton has been constructed; But we need to fill the array IN3 with the necessary information: *)
Sort the nodes such that all the nodes that have the same transition state on input 0 are grouped together, and all the nodes with the same transition state on input 1 are grouped together;
Now easily fill in the array IN3;

Minimize the resultant automaton using Lemma 3.3;

If we have $N_1 \times N_2$ processors, we can construct the product machine in $O(1)$ time. Sorting of the nodes to fill in IN3 can be done in $O(\log(N_1 N_2))$ time using $N_1 N_2$ processors. However there may be many nodes in the automaton thus constructed which are either impossible or can not be reached from the start state. For instance if there are two nodes $q_{1,i}$ and $q_{2,j}$ at the same level in A_1 and A_2 respectively, and if the incoming arc to $q_{1,i}$ is labeled with a 0 and the incoming arc to $q_{2,j}$ is labeled with a 1, then we don't have consider the node $(q_{1,i}, q_{2,j})$ since such a node is clearly impossible. There may still remain nodes which are not reachable from the start node. We will eliminate these nodes when we apply the minimization algorithm. The minimization algorithm will be modified in order to perform this. Thus we have the following

Theorem 4.1 *The product of two automata with N_1 and N_2 nodes respectively can be constructed in $O(\frac{M_{N_1N_2}\log^2(N_1N_2)}{P}+\log^2(N_1N_2))$ time using P CREW PRAM processors (for any $P \leq M_{N_1N_2}$).*

5 Model Checking

Model checking has been used widely to verify finite state systems. Model checking procedure typically involves the following steps: 1) Obtain the state transition graph of the system. If the system consists of say n components, the number of states in the global system will be exponential in n causing the state space explosion. But for a moment assume that the state transition graph has been constructed; 2) Express the properties to be verified (like fairness, deadlock free, liveness, etc.) as formulas in an appropriate logic (such as CTL); 3) Perform a reachability analysis in the state transition graph to identify all the states that satisfy the formula derived in step 2.

Even for reasonably large values of n, it is impossible to construct an exponentially sized graph. Thus it becomes necessary to represent the system 'symbolically', i.e., we should come up with efficient encoding schemes. BDDs have been used successfully in verifying systems with even 5×10^{20} states. However it should be pointed out any such clever encoding scheme is at best a heuristic and is bound to fail in the worst case unless $\mathcal{P} = \mathcal{NP}$.

Consider a boolean circuit on n variables that we want to verify. Any assignment of values to these n variables can be thought of as a state of the system. One could represent the states of such a system as well as the transition graph of the system as BDDs in a straight forward way [6].

5.1 Algorithms for Reachability Analysis

Let V stand for the state variables of the system to be verified. Let $N(V,V')$ stand for the BDD corresponding to the transition graph of the system. Here $V' = V$ and stands for the next states. The problem of identifying the set of reachable states from a set of start states is an essential step in model checking. We present an efficient parallel algorithm in this section for reachability analysis. This algorithm makes use of procedures for manipulating BDDs. In particular, the following operations on BDDs will be used: OR, AND, substitution (for a variable), and existential quantification.

If Φ is a BDD involving a variable say v, the substitution operation is defined as follows: $[\Phi]_{v=0}$, (or $[\Phi]_{v=1}$) is nothing but the BDD for the formula of Φ when the value of v is substituted as 0 (or 1). This operation can easily be performed in $O(\log|V|)$ time using $\frac{|V|}{\log|V|}$ processors where V is the set of nodes in Φ. Existential quantification is defined as: $\exists x\Phi$. Given the BDD for Φ, we could obtain the BDD for $\exists v\Phi$ as follows: $[\Phi]_{v=0} \vee [\Phi]_{v=1}$.

Let $V = \{x_1, x_2, \ldots, x_n\}$ and let B_0 stand for the BDD of the set of start states. Let B_i (for $i = 1, 2, \ldots$) stand for the BDD of states that are reachable with i or less transitions from the start states. ($B_i(V)$ is also used to denote B_i.) One could compute the BDD B_i, given B_{i-1} and

$N(V,V')$ as follows:

Algorithm Reach;
(* B and C are BDDs *)

$C := \emptyset$; $B := B_0$;
repeat until $B = C$
$C := B$;
for $j := 1$ *to* n *do*
$B(V) := B(V) \bigvee \exists_{x_j \in V'} [B(V') \wedge N(V', V)]$

In the above algorithm, B_i is nothing but B after i iterations of the outer *for* loop. If the number of nodes in B_{i-1} is N_{i-1}, then we could execute one run of the *for* loop in time $O(\log^2(N_i'))$ using $M_{(N_i')^2}$ processors, where N_i' is defined to be the maximum size of any BDD generated in computing B_i. Thus one iteration of the *repeat* loop runs in time $O(n \log^2(N_i'))$ time using $M_{(N_i')^2}$ processors. The above fixed point computation can be speeded up by iterated squaring technique. If m iterations are needed to compute the fixed point, only $O(\log m)$ iterated squaring steps are needed. Thus we arrive at the following

Theorem 5.1 *Each iteration of the above analysis can be performed in* $O\left(\frac{nM_{N^2}(\log^2 N)}{P} + n\log^2 N\right)$ *time using* P *CREW PRAM processors (for any* $P \leq M_{N^2}$*), where* N *is defined to be the maximum size of any BDD generated in the process. If* m *iterations are needed to compute the fixed point, we can reduce this number to* $O(\log m)$ *with the help of the same number of processors.*

There may be some problem with the repeated squaring trick, as has been pointed out in [5]. The size of the intermediate BDDs may be much larger than the final BDD, in which case it may help to perform the iterations sequentially. But each phase of the iteration can be substantially speeded up as explained above.

5.2 Checking for Other Conditions

One may want to check for liveliness, fairness, absence of deadlock, etc. The approach is essentially the same. Given any condition, we express it as a formula in an appropriate logic and compute all the states that satisfy this condition. The steps involved in such a computation will be analogous to those in the reachability analysis, and can be completed along similar lines.

In summary, the efficient parallel algorithms proposed in this paper for BDDs manipulation can be put to use in model checking. This will potentially speed up the computing time by several orders of magnitude.

6 A New State Graph Encoding

Consider a concurrent system with n components. Let $G_i(V_i, E_i)$ (for $i = 1, 2, \ldots, n$) be the state graphs of the processes and let $G(V, E)$ be the state graph of the global system. Assume w.l.o.g. that $|V_i| = 2^{k_i}$ for some integer k_i and for all i in the range $[1, n]$. Then we could map the nodes of G uniquely into the integers $\{1, 2, \ldots, |V|\}$. Given an integer N in the interval $[1, |V|]$, the corresponding node of G can be inferred readily: the first k_1 bits correspond to the state of the first process, the next k_2 bits correspond to the state of the second process, and so on.

Likewise, we could also map the edges of G uniquely into the integers $\{1, 2, \ldots, |V|^2\}$. (The cardinality of this set could be reduced to $|E|$. But for simplicity of discussion assume this range.) If M is any integer in the interval $[1, |V|^2]$, the first $\log |V|$ bits and the second $\log |V|$ bits of M represent the two end vertices of the corresponding edge.

The above simple encoding has the following property: The vertices of G are simply $\{1, 2, \ldots, |V|\}$. Also, given an integer M in the interval $[1, |V|^2]$, we could check if it is an edge of G or not, in $O(\log n)$ time using $\frac{n}{\log n}$ processors. One of the major advantages of this encoding is that we never have to generate and store the whole of G; we could generate the nodes and edges on the fly. This encoding scheme and the BDDs can be thought of as two extremes of a spectrum of encoding schemes.

BDD is a very efficient encoding scheme and it makes use of the special structure of the graph to be encoded, namely, that the graph represents a boolean function. The very structure of BDDs enables the design of efficient algorithms for their manipulation. On the other hand, the simple encoding scheme we mentioned above applies to an arbitrary graph. Though the memory usage is minimal, manipulating the graphs might be time consuming under this encoding scheme. An interesting and important question is: Are there other encoding schemes that are in between these two encoding schemes, whose applicability will be more general than that of BDDs and which will lead to more efficient manipulation algorithms than the above simple encoding scheme? For instance, one could obtain a number of encoding schemes in the following manner: Partition the components of the concurrent system. Use BDDs to represent each group and employ the above encoding scheme for the conjunction of groups.

7 Searching Less Space–Probabilistic Model Checking

The idea of probabilistic model checking is to prune the search in the state graph using information about the transition probabilities. If $G(V, E)$ is the state graph of a single process, edges of this graph could be labeled with probabilities in the following manner: Edge $\langle i, j \rangle$ is labeled with p_{ij} if the probability of a transition from state i to state j is p_{ij}. If we have such a labeled graph for each one of the n processes, we could obtain a labeled graph for the composition of these processes as well (using basic rules for composing probabilities).

In probabilistic model checking, for instance, we could search only certain portions of the state graph and make high probability inferences. As an example consider a system of n processes in which we would like to detect any deadlocks that may be present. We randomly select and explore only a (small) portion of the graph and based on the information gained, we output (a confidence interval on) the probability of a deadlock occurring in the system. The nodes and edges to be explored can be generated on the fly; there is no need for generating the whole state graph.

If we can afford to visit only a small portion of the state graph, it is quite reasonable to expect that our answer may not be correct in the worst case. But we can show that with high probability our assertions will be correct. To illustrate our idea, consider the following simple scenario: Assume that each node has information about whether or not it is a deadlock node. Also assume that the deadlocks are uniformly distributed in the following sense: Starting from a node in the graph, if we perform a random walk in the graph for t steps (for some t), the probability, p_t, of encountering a deadlock node is independent of the starting node.

Under the above assumption it turns out that we only have to explore the graph (for t steps) starting from $L = O(\log N)$ random nodes in the state graph (here N is the number of nodes in the composed state graph), before we'll be able to specify a confidence interval on the probability of a deadlock. Similar random sampling techniques have been employed in designing optimal sequential and parallel algorithms for various problems (see e.g., [19] and [18]).

If $L(= \theta(\alpha \log N))$ is the total number of random origins explored and if M was the number of origins starting from which a deadlock node was visited, we conclude that the total fraction of deadlock nodes in the whole system will be in the interval $\left[(1-\epsilon)\frac{M}{L},\ (1+\epsilon)\frac{M}{L}\right]$, where ϵ is any fixed number ($0 < \epsilon < 1$). In the special case $M = 0$, this interval reduces to a single point, namely, 0. The probability that our conclusion is correct can be shown to be $\geq 1 - \exp\left(\frac{-\epsilon^2 \alpha \log N}{2}\right)$, using the sampling lemmas proven in [19]. For instance we can make this probability $\geq 1 - \frac{1}{N^2}$ (which is very nearly equal to 1) for proper choices of ϵ and α. If we desire a narrow confidence interval (i.e., a small ϵ), naturally we will have to spend more time (i.e., we have to choose a bigger α).

The above discussion applies to any representation of the concurrent system, in particular to CTSMP as well as BDD representations. We propose to investigate the above idea of probabilistic search in general instances of model checking.

The encoding scheme we mentioned in the last section seems to yield easily to parallelism. For example, if G has a regular structure (for example G could be a complete graph), and if we have P processors we could deterministically partition the task of reachability analysis equally among the P processors and achieve optimal speedup. This involves each processor analyzing different portions of the state graph in parallel. Even if G has no known structure we could use the following randomized strategy for obtaining near optimal speedup. Each processor starts from a random node in G and starts exploring. For example, each processor could perform a depth restricted DFS starting from a random node. We can show that with high probability the load will be equally (up to a small multiplicative constant) shared by all the P processors.

This encoding scheme is also well suited in the context of probabilistic model checking. Probablism on this encoding will enable us to prune the search considerably. Any processor while exploring a path could give up this path if the path has a very low associated probability. Notice also that the memory requirements are very reasonable. Even if one performs a (sequential) recursive DFS of the whole graph, at any time we need only a memory that is polynomial in n for the stack.

8 Randomization–A Powerful Technique

We believe that both of our approaches to state space explosion can be enhanced with the wonderful technique of randomization. Since its introduction in 1976, randomization has been used to solve myriads of computational problems. Recent implementation results show that this technique is very practical as well. In this section we give a brief introduction to randomized algorithms. One of the goals of this project will be to identify steps in model checking which can benefit from randomization. We have some preliminary results in this direction as well, as we mention toward the end of this section.

Classical approaches to introducing randomness in algorithms typically assume a distribution on possible inputs and, compute the expected performance of various (deterministic) algorithms. Quick sort is a good example. If one assumes that each input permutation is equally likely to occur, Quick sort runs in an expected $O(n \log n)$ time to sort n numbers. The credibility of such an approach critically depends on the assumption made on the inputs. There may be applications where the input distribution is quite different from the one used in the probabilistic analysis.

As an attractive alternative, Rabin [16] proposed introducing randomness in the algorithm itself. A randomized algorithm is one wherein certain decisions are made based on the outcomes of coin flips. No matter what the input is, a large fraction of all possible outcomes for the coin flips will ensure 'good performance' of the algorithm. Thus the two approaches differ in the probability space used for analysis. In the former one considers the space of all possible inputs and in the later one employs the space of all possible outcomes for coin flips.

To give a flavor for the above notions, we now give an example of a randomized algorithm. We are given a polynomial of n variables $f(x_1, \ldots, x_n)$ over a field F. It is required to check if f is identically zero. We generate a random $n-$vector $(r_1, \ldots, r_n)$ $(r_i \in F, i = 1, \ldots, n)$ and check if $f(r_1, \ldots, r_n) = 0$. We repeat this for k independent random vectors. If there was at least one vector on which f evaluated to a non zero value, of course f is nonzero. If f evaluated to zero on all the k vectors tried, we conclude f is zero. It can be shown that the probability of error in our conclusion will be very small if we choose a sufficiently large k. In comparison, the best known deterministic algorithm for this problem is much more complicated and has a much higher time bound.

Notice that the algorithm we mentioned in section 7 for checking for the absence of deadlock (i.e., the case when $M = 0$) resembles the above algorithm for polynomial identity. The same algorithm could be used to check for other conditions such as liveness, fairness, etc. In fact the

search algorithm presented there is a specific randomized algorithm. However, randomization is a powerful tool that can be used in many different ways in model checking.

Two extremely important advantages of randomized algorithms are their simplicity and efficiency. This fact has been demonstrated many a time in this decade by the enormous number of algorithms that have been designed for several vital problems. We list some of these problems now (This list by no means is exclusive): packet routing, parallel sorting, the maximal independent set problem, parallel connectivity, parallel DFS, parallel dictionary operations, computational geometry problems such as Voronoi diagram, convex hull, etc. A survey on randomized parallel algorithms for comparison problems is [19].

We have already obtained some success in applying randomization to a problem related to model checking using BDDs. As was mentioned before, DFA minimization is an important step in manipulating BDDs. DFA minimization can be thought of as Coarsest Partition Problem (CPP) with two functions. Recently, Sarnath [20] has given a simple algorithm for CPP with one function. His algorithm runs in time $O(\log^2 n)$ on an n-processor CREW PRAM. In this section we present a randomized algorithm for the same problem that runs in time $O(\log^* n \log n)$ using $\frac{n}{\log^* n}$ CRCW PRAM processors. Though this algorithm does not immediately imply an optimal algorithm for DFA minimization, we believe that the ideas employed can be extended to DFA minimization. We plan to investigate this further.

Next we give a brief summary of Sarnath's algorithm and our improvement of this algorithm.

Coarsest Partition Problem Given a set S of n elements $a_1, a_2, \ldots, a_n$, a partition $\pi = (\pi_1, \pi_2, \ldots, \pi_k)$ of S and a function $f : S \rightarrow S$, the coarsest partition problem (CPP) is to find the coarsest partition $\pi' = (\pi'_1, \pi'_2, \ldots, \pi'_\ell)$ such that:

1. $\forall a_i, a_j \in S, a_i, a_j \in \pi'_m \Rightarrow \exists p$ such that $f(a_i), f(a_j) \in \pi'_p$; and
2. $\forall m, 1 \leq m \leq \ell$, $\exists p, 1 \leq p \leq k$ such that $\pi'_m \subseteq \pi_p$.

The crucial lemma proven in [20] is the following: Two elements a_i and a_j of S will belong to different blocks in the final partition if and only if $\exists k, 0 \leq k \leq n(n-1)/2$, such that $f^k(a_i)$ and $f_k(a_j)$ belong to different initial blocks. (Here $f^0(a) = a$ and $f^k(a) = f(f^{k-1}(a))$).

The idea of the CPP algorithm is to make use of the above fact and the 'pointer jumping' or 'doubling trick'. In order to check if two elements a_i and a_j from the same initial partition will belong to the same final partition we can evaluate $f^k(a_i)$ and $f^k(a_j)$ for $k = 1, 2, \ldots, n(n-1)/2$. We can stop as soon as we find a k for which the have different values. Instead of checking this condition for each k, we can use the doubling trick and reduce the number of evaluations drastically.

Here is a description of the algorithm: P is an array of size n. After t iterations of the *for* loop, $P[i]$ holds the index of $f^{2^t}(a_i)$. Initially, $P[i]$ will have a value j if $f(a_i) = a_j$. L is another array of size n. To begin with L will have the initial labels (i.e., partition labels) of elements. As the

algorithm proceeds, the labels will get changed such that at any given time, two elements will have the same label if they have not been so far shown to belong to two different final partitions.

for $i := 1$ to $2 \log n$ *do*

1.1 *for each* i *in parallel:* $L[i] := n * (L[i] - 1) + L[P[i]]$;
1.2 *for each* i *in parallel:* $P[i] := P[P[i]]$;
2. Sort the array L and assign $L[i] :=$ Rank of $L[i]$ in L;

The crucial observation we make is that step 2 in the above *for* loop can be replaced with a more efficient operation. Sorting is an expensive step. In step 1.1, each element is assigned a label in the range $[0, n^2]$ such that elements that belong to the same partition have the same label. In step 2, we reassign a label to each element from the range $[0, n]$ with the help of sorting (now also elements in the same partition will get the same label). We can indeed get the same effect by making use of an algorithm for 'representative selection'. There is a randomized algorithm for this [11] that runs in $O(\log^* n)$ time using $\frac{n}{\log^* n}$ CRCW PRAM processors.

The representative selection algorithm runs as follows: A random hash function $h : \{1, 2, \ldots, N\} \rightarrow \{1, 2, \ldots, cn\}$ (c being a constant > 1) is used, where $[1, N]$ is the range of values that the elements can take. (In the CPP algorithm above, $N = n^2$). Each element i tries to write in common memory cell $h(i)$. If there are more than one processors that try to write in the same cell, one of them succeeds. All the elements with the same value i have now chosen a representative (i.e., a label in the range $[1, cn]$). One can show that with high probability almost all the elements will have chosen a representative in the first attempt. Those which do not succeed in the first attempt, participate in future rounds. [11] show that no more than $O(\log^* n)$ attempts will be needed before every element has a representative.

It is still an open problem whether one could obtain a parallel algorithm for DFA minimization that is optimal.

9 Conclusions

We have presented algorithms for model checking. Our approach to state explosion falls under two categories: 1) Reducing the search space, and 2) Searching less space. In particular, we have given efficient parallel algorithms for manipulating BDDs. These algorithms are of independent interest in many application domains. As an application we have shown how to perform reachability analysis in model checking efficiently. The BDD algorithms could be used for instance in solving the satisfiability problem also in addition to many other applications. We have also offered probabilistic searching as an attractive alternative to existing approaches. We have also shown that randomization could help in the design of algorithms for model checking.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publications, 1974.

[2] R.P. Brent, The Parallel Evaluation of General Arithmetic Expressions, Journal of the ACM, 21(2), pp. 201-208, 1974.

[3] R.E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Trans. on Computers, Vol. C-35, No. 8, 1986, pp. 677-691.

[4] J.R. Burch, E.M. Clarke, and D.E. Long, Symbolic Model Checking with Partitioned Transition Relations, in Proc. VLSI Conference, Edinburgh, 1991.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill, Sequential Circuit Verification Using Symbolic Model Checking, Draft, 1990.

[6] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, Symbolic Model Checking: 10^{20} States and Beyond, in Proc. International Workshop on Formal Methods in VLSI Design, 1991.

[7] E.M. Clarke and O. Grümberg, Research on Automatic Verification of Finite-State Concurrent Systems, Ann. Rev. Comput. Sci., 1987, pp. 269-290.

[8] E.M. Clarke, D.E. Long, and K.L. McMillan, Compositional Model Checking, manuscript, 1991.

[9] E.M. Clarke, E.A. Emerson, and A.P. Sistla, Automatic Verification of Finite State Concurrent System Using Temporal Logic Specification, ACM Transactions on Programming Languages and Systems, 8 (2): 244-263, April 1986.

[10] R. Gerber, and I. Lee, Specification and Analysis of Resource-Bound Real-Time Systems, Technical Report, Dept. of CIS, Univ. of Pennsylvania, 1991.

[11] J. Gil, Y. Matias, and U. Vishkin, Towards a Theory of Nearly Constant Time Parallel Algorithms, Proc. IEEE Symp. on Foundations of Computer Science, 1991, pp. 698-710.

[12] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Company, 1992.

[13] J. JáJá and S.R. Kosaraju, Parallel Algorithms for Planar Graph Isomorphism and Related Problems, IEEE Trans. on Circuits and Systems, 35(3), pp. 304-310, 1988.

[14] S. Kimura and E.M. Clarke, A Parallel Algorithm for Constructing Binary Decision Diagrams, in Proc. IEEE International Conference on Computer Design, pp. 220-223, 1990.

[15] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes*, Morgan-Kaufman Publishers, 1992.

[16] M.O. Rabin, Probabilistic Algorithms, in: Traub,J.F., ed., *Algorithms and Complexity*, Academic Press, New York, 1976, pp.21-36.

[17] S. Rajasekaran, and J.H. Reif, Derivation of Randomized Sorting and Selection Algorithms, Technical Report, Aiken Computing Lab., Harvard University, 1985.

[18] S. Rajasekaran, and J.H. Reif, Optimal and Sub-Logarithmic Time Randomized Parallel Sorting Algorithms, SIAM Journal on Computing, vol. 18, no. 3, 1989, pp. 594-607.

[19] S. Rajasekaran and S. Sen, Random Sampling Techniques for Parallel Algorithms Design, in *Synthesis of Parallel Algorithms*, edited by J.H. Reif, Morgan-Kaufmann Publishers, 1992.

[20] R. Sarnath, An Improved Algorithm for DFA Minimization, to be presented in the Thirtieth Annual Allerton Conference on Communication, Control, and Computing, Illinois, Oct. 1992.