A Parallel Algorithm for Constructing Binary Decision Diagrams

S. Kimura
Kobe University, Japan

E. M. Clarke
Carnegie Mellon University, Pittsburgh

Abstract Ordered binary decision diagrams [2] are widely used for representing Boolean functions in various CAD applications. This paper gives a parallel algorithm for constructing such graphs and describes the performance of this algorithm on a 16 processor Encore Multimax. The execution statistics that we have obtained for a number of examples show that our algorithm achieves a high degree of parallelism. In particular, with ten processors our algorithm is almost an order of magnitude faster on some examples than the program described in [6]. Moreover, on many examples it exhibits essentially linear speedup as the number of processors is increased.

Our approach to binary decision diagrams is somewhat different from the one used in [2]. We view the binary decision diagram for an n-argument boolean function \( f \) as the minimal finite state machine for the set of Boolean vectors of length \( n \) that satisfy \( f \) (i.e. the set of vectors in \( f^{-1}(1) \)). Because the minimal finite automaton for a regular language is unique up to isomorphism, it is easy to argue that this representation provides a canonical form for Boolean functions. Boolean operations involving NOT, AND, OR, etc. are implemented by the standard constructions for complement, intersection, and union of the finite languages accepted by these automata. In general, each of these operations involves building a product automaton and then minimizing it.

We discuss a parallel algorithm for computing the product of two automata and for minimizing the result. When we construct a binary decision graph, our algorithm follows the syntactic structure of the Boolean formula. First, the level of each Boolean operation is determined. Operations in the same level can be performed in parallel. If there are few operations at some level, then these operations are divided into a sequence of sub-operations that can be processed in parallel.

\[ \text{The second author was partially supported by NSF grant CCR-87-226-33 and by the Defense Advanced Research Projects Agency ARPA Order No. 4976.} \]
1. Introduction

The ordered binary decision diagram [2] is an acyclic graph representation for Boolean functions. Because this representation provides a canonical form (i.e., two functions are equivalent if and only if they have the same form) and is quite succinct in most cases, it has become widely used in CAD applications. However, the construction of binary decision diagrams for certain large or particularly complex Boolean functions can be very time consuming. Consequently, it is important to find ways of speeding up the construction process. This paper describes a parallel algorithm for this task. The algorithm has been implemented on a 16 processor Encore Multimax and tested on several standard examples. In the next two months we hope to have the program running on the 64 processor RP3 at IBM Yorktown Heights.

Our approach to binary decision diagrams uses some simple ideas from finite automata theory. An n-argument Boolean function can identified with the set of Boolean vectors that make it true. For example, the function denoted by the Boolean expression \( x_1 \cdot x_2 + \neg x_2 \cdot x_3 \) is uniquely determined by the set of vectors \{(1,1,0), (1,1,1), (0,0,1), (1,0,1)\}. The corresponding set of strings \{110, 111, 001, 101\} is a finite language. Since all finite languages are regular, there is minimal finite automaton that accepts this set. This automaton provides a canonical representation for the original Boolean function. Logical operations on Boolean functions can be implemented by set operations on the languages accepted by the finite automata: AND corresponds to the set intersection, OR corresponds to the set union, and NOT corresponds to the set difference ((the universal set) \( -\) (the specified set)). Standard constructions from elementary automata theory can be used to build the binary decision diagram for a composite Boolean function from the decision diagrams for the atomic proposition symbols in the formula.

There are several (relatively minor) differences between our notion of a binary decision diagram and the one given in [2]. In the sequential case these differences should have little effect on the complexity of either algorithm.

- Our graphs represent only the set of strings that are in the language (i.e., the set of Boolean vectors which make the function true). The original version keeps the set of strings for which the function is true, as well as the set for which it is false.

- In our scheme, it is unnecessary to label the nodes of the graph with information about the corresponding boolean variable. The depth of the node in the graph uniquely determines its label.

Nevertheless, we believe that there are some important reasons for viewing binary decision diagrams as automata. Minimization of finite automata is a well-understood task for which good algorithms are available. In fact, many of the important properties of binary decision diagrams follow directly from properties of the minimization procedure. A typical example is the normal form property (the proof of this property in Bryant's paper is not so straightforward). Moreover, powerful techniques from Automata and Formal Language Theory can
be used to investigate questions like what properties of a boolean function determine the size of its binary decision diagram. We have obtained some results of this type that we hope to present in a future paper.

In the construction of a binary decision diagram corresponding to a Boolean function, a parse tree of the function is used, where leaf nodes correspond to input variables, and non-leaf nodes correspond to Boolean operation. The level of each node is defined from leaf nodes to the top of the tree, and operations at the same level are performed in parallel. If there are only a few operations in some level, these operations are divided into several sub-operations to extract additional parallelism. Using the above method, we keep rather high parallelism in the construction.

Our paper is organized as follows: In Section 2, we review some of basic terminology on finite automata and binary decision diagrams. Section 3 describes the algorithm for building the product automaton and minimizing it. Section 4 describes the parallel algorithm and gives performance statistics for a number of examples. The paper concludes in Section 5 with a summary and discussion of some directions for future research.

2. Finite Automata and Binary Decision Diagrams

We start with some simple definitions dealing with finite automata and binary decision diagrams. A string is a sequence of symbols over some alphabet \( \Sigma \). In this paper, the alphabet will always be \( \Sigma = \{0, 1\} \), where 0 represents False and 1 represents True. For example, 110 and 111 are strings. The length of a string is the number of symbols in the string. Thus the length of 110 is 3.

A finite automaton \( M \) is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where \( Q \) is a finite set of states, \( \Sigma \) is the alphabet for strings, \( \delta \) is the state transition function from \( Q \times \Sigma \) to \( Q \), \( q_0 \) is the initial state in \( Q \), and \( F \) is a set of final states in \( Q \). \( M \) accepts a string \( a_1a_2...a_n \) where each \( a_i \in \Sigma \) if and only if there exists a sequence of states \( q_0, q_1, ..., q_n \) such that \( q_i = \delta(q_{i-1}, a_i) \) and \( q_n \in F \). The set of strings accepted by \( M \) is called the language of \( M \) and will be denoted by \( L(M) \).

For example, \( M = (\{q_0, q_1, q_2, q_3, \bot\}, \{0, 1\}, \delta, q_0, \{q_3\}) \) accepts \( \{110, 111\} \), where \( \delta \) is defined as \( \delta(q_0, 0) = \bot, \delta(q_0, 1) = q_1, \delta(q_1, 0) = \bot, \delta(q_1, 1) = q_2, \delta(q_2, 0) = q_3, \delta(q_2, 1) = q_3, \delta(q_3, 0) = \bot, \delta(q_3, 1) = \bot, \delta(\bot, 0) = \bot, \) and \( \delta(\bot, 1) = \bot \). \( \bot \) is called a sink state. The representation of \( \delta \) as a directed graph is shown in Figure 1. Note that the graph is acyclic; this will be true for all of the automata that we consider in this paper. The sink state is not shown in the figure for simplicity. In the following, the sink state may not be mentioned explicitly, but its existence is always assumed.

A Boolean function \( f \) with \( n \)-variables is a function from \( \{0, 1\}^n \) to \( \{0, 1\} \). For example,

\[
 f(x_1, x_2, x_3) = \begin{cases} 
 1, & \text{if } (x_1, x_2, x_3) \text{ is } (1, 1, 0) \text{ or } (1, 1, 1); \\
 0, & \text{otherwise};
\end{cases}
\]
is a boolean function with three boolean variables. The value of the function could, of course, also be given by a boolean expression \( f(x_1, x_2, x_3) = x_1 \land x_2 \). Observe that the set of triples in the domain where \( f \) has value 1 (i.e. \( f^{-1}(1) \)) is the same as the language that is accepted by the finite automaton in the previous example.

In general, the set of elements in \( \{0, 1\}^n \) for which \( f \) is 1 can be used to represent \( f \). If we associate the \( n \)-tuple \((a_1, a_2, ..., a_n)\) with the string \( a_1a_2...a_n \), then each set of \( n \)-tuples from \( \{0, 1\}^n \) will correspond to a set of strings over \( \Sigma = \{0, 1\} \) with length \( n \). This correspondence allows us to associate a finite language contained in \( \Sigma^n = \{0, 1\}^n \) with each \( n \) variable Boolean function. Since all finite languages are regular, it follows from the correspondence between regular languages and finite automata, that each such language is accepted by some finite automaton. The minimal finite automaton corresponding to the Boolean function \( f \) provides a canonical form for \( f \): two \( n \)-variable Boolean functions will have the same minimal automaton if and only if they are logically equivalent. Since each node in the state-transition graph for a Boolean function will have at most two successors (one for each value of \( \Sigma \)), we can view this graph as a binary decision diagram for the function.

We illustrate these ideas by giving the finite automata and binary decision diagrams for some simple \( n \)-variable Boolean functions. First, we consider the function \( f_U \) which is identically 1 for all possible values of its arguments, i.e. \( f_U(x_1, ..., x_n) = 1 \) for all values of \( x_1, ..., x_n \). The language corresponding to \( f_U \) consists of all strings of length \( n \) over the alphabet \( \Sigma = \{0, 1\} \), and accepted by a finite automaton \( M_U = (\{q_0U, q_1U, ..., q_nU\}, \{0, 1\}, \delta_U, q_{0U}, \{q_nU\}) \), where \( \delta_U \) is defined as \( \delta_U(q_i, 0) = q_{i+1} \) and \( \delta_U(q_i, 1) = q_{i+1} \). The binary decision diagram is shown in Figure 2.
Similarly, the $n$-variable function that is identically 0 for all values of its arguments, i.e., $f_{\emptyset}(x_1, ..., x_n) = 0$ for all values of $x_1, ..., x_n$, corresponds to the empty language and is accepted by a finite automaton $M_{\emptyset} = (\emptyset, \{0, 1\}, \emptyset, \emptyset, \emptyset, \emptyset)$.

Finally, the function $f_i(x_1, ..., x_n) = x_i$ corresponds to the set $\{0, 1\}^i \{0, 1\}^{n-i}$. This set is accepted by the finite automaton $M_i = (q_{0i}, q_{1i}, ..., q_{ni}, \{0, 1\}, \delta_i, q_{0i}, q_{ni})$, where $\delta_i$ is defined as $\delta_i(q_j, 0) = q_{j+1}$ and $\delta_i(q_j, 1) = q_{j+1}$ for $j$ in $\{0, 1, ..., i-2, i, ..., n-1\}$ and $\delta_i(q_{i-1}, 1) = q_i$. The binary decision graph for this case is shown in Figure 3.

3. Implementing Boolean Operations on Binary Decision Diagrams

Let $M_1 = (Q_1, \Sigma, \delta_1, q_1^0, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2^0, F_2)$ be the binary decision diagrams for two $n$-variable Boolean functions $f_1$ and $f_2$. We will show how simple automata theoretic constructions can be used to find the binary decision diagrams for various combinations of $f_1$ and $f_2$ involving the Boolean operations AND($\land$), OR($\lor$), NOT($\neg$), and EXOR($\oplus$).

We consider the AND operation first. The set of strings over $\{0, 1\}$ that satisfy $f_1 \land f_2$ corresponds to the intersection of sets accepting $M_1$ and $M_2$. The standard construction of a finite automaton $M$ that accepts the intersection of $L(M_1)$ and $L(M_2)$ may be used in this case. $M = (Q_1 \times Q_2 \cup \{\bot\}, \{0, 1\}, \delta_\land, (q_1^0, q_2^0), F_1 \times F_2)$, where $\bot$ denotes the sink state for the product automaton. $\delta_\land$ is defined as $\delta_\land((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$, if both $\delta_1(q_1, a)$ and $\delta_2(q_2, a)$ are not the sink state, and $\bot$ otherwise.

The OR operation is similar. The OR of two Boolean functions represented by $M_1$ and $M_2$ corresponds to the union of sets accepted by $M_1$ and $M_2$. The standard construction for such an $M$ can also be used in this case. $M = ((Q_1 \times Q_2) \cup \{\bot\}), \{0, 1\}, \delta_\lor, (q_1^0, q_2^0), (F_1 \times Q_2) \cup (Q_1 \times F_2))$, where $\delta_\lor$ is defined as $\delta_\lor((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$, if $\delta_1(q_1, a)$ is not the sink state or $\delta_2(q_2, a)$ is not the sink state, and $\bot$ otherwise.

The NOT operator corresponds to the set difference. Let $U$ be the set of all strings with length $n$, then $U - L(M_1)$ corresponds to the negation of the Boolean function represented by $M_1$. A finite automaton accepting $U - L(M_1)$ can be constructed from $M_U = (Q_U, \{0, 1\}, \delta_U, q_0^U, F_U)$ and $M_1$ as $M = ((Q_U \times Q_1) \cup \{\bot\}), \{0, 1\}, \delta_\neg, (q_0^U, q_1^0), F_U \times (Q_1 - F_1))$, where $\delta_\neg$ is defined in the same manner as for the OR operation. The EXOR operator $\oplus$ is also similar to the OR operator. The finite automaton for this operator is given by $M = ((Q_1 \times Q_2) \cup \{\bot\}, \{0, 1\}, \delta_\oplus, (q_1^0, q_2^0), F_1 \times (Q_2 - F_2) \cup (Q_1 - F_1) \times F_2)$, where $\delta_\oplus$ is defined in the same manner as for the OR operation.

Note that determining the state set of the finite automaton for each of these four operators involves a product construction $M_1 \times M_2$. We exploit this observation by giving a single procedure for the product construction that is parameterized by the type of Boolean operator involved. Also note, that in each case the resulting automaton $M$ may not be minimal, even if both $M_1$ and $M_2$ are minimal. Consequently, a final minimization stage is needed after the product construction in order to obtain a canonical binary decision diagram.
4. The Basic Algorithm for Constructing Binary Decision Diagrams

4.1. The Product Automaton

Because of our convention regarding final states, a binary decision diagram \( M = (Q, \{0, 1, q_0, F\} \) may be represented by its state-transition graph alone. In particular, two edges emanate from each state \( q \): a 0-edge pointing to \( \delta(q, 0) \) and a 1-edge pointing to \( \delta(q, 1) \). In generating the product automaton for the result of some two-argument Boolean operation applied to \( M_1 \) and \( M_2 \), the initial product state is given by \((q_0^1, q_0^2)\) where \( q_0^1 \) is the initial state of \( M_1 \) and \( q_0^2 \) is the initial state of \( M_2 \). The successors of this state are determined for the inputs 0 and 1, and this process is repeated until no new state pairs are generated. The process is shown in Figure 4.

Note that there are only two places where we need to take into account the types of the Boolean operator: when we compute \((\delta_1(q_1, 0), \delta_2(q_2, 0))\) and when we compute \((\delta_1(q_1, 1), \delta_2(q_2, 1))\). The most time-consuming part of this procedure is deciding whether a pair is new or not. By using a hash table we can make this test take essentially constant time. The hash function that we use is given by

\[
\text{Hash}(q_1, q_2) = \text{mod}(q_1 * (\text{HASH_SIZE}/2) + q_2, \text{HASH_SIZE}),
\]

where \( q_1 \) and \( q_2 \) are integer values for the state pointers. The size of the hash table (the parameter \( \text{HASH_SIZE} \)) and the hash function are critical factors in determining the execution time of this phase of the algorithm. Now, each hash tables holds 8191 entries.
Let the initial pair be \((q_0^1, q_0^2)\);
Put the pair in the queue \(S\), and allocate a new state for it;
While (\(S\) is not empty) do Begin
  Dequeue a pair \((q_1, q_2)\) from \(S\);
  For symbol \(a \in \{0, 1\}\) do
    Begin
      Compute the pair of successors \((\delta_1(q_1, a), \delta_2(q_2, a))\);
      If this pair is new, then
        add the pair to \(S\), and generate a new state.
      Connect the \(a\)-edge from the state corresponding to \((q_1, q_2)\) to
      the state corresponding to \((\delta_1(q_1, 0), \delta_2(q_2, 0))\);
    End;
  End;
End;

Fig. 4 Construction of the product automaton.

States are recorded in a linked data structure with the format

typedef struct { int edge0; int edge1; int next; } state_cell;

The field edge0 (edge1) is a pointer to the next state with respect to input 0 (input 1), and
the field next is a pointer to another state with the same hash key. It should be mentioned
that we need no special memory for a state pair. Let a state \(q\) correspond to a state pair
\((q_1, q_2)\), a state \(q'\) correspond to a state pair \((\delta_1(q_1, 0), \delta_2(q_2, 0))\), and a state \(q''\) correspond to
a state pair \((\delta_1(q_1, 1), \delta_2(q_2, 1))\). First, a pointer to \(q_1\) is entered in edge0 of \(q\), and a pointer
to \(q_2\) is entered in edge1 of \(q\). Then \(q\) is registered in a hash table and in a queue. After
the state \(q\) is dequeued and calculated the next state corresponding to the pair, the edge0
(edge1) of \(q\) is changed to a pointer to \(q'\) (\(q''\)). If we generate pairs of states in a breadth
first manner, the same state pair as \((q_1, q_2)\) will not be generated after the next state of the
pair is computed. Thus, our method for reducing the memory usage works quite well.

An example illustrating this phase is shown in Figure 5, where the intersection of \(M_1\) and
\(M_2\) is computed (because of an AND operation in the original formula). \(M_1\) corresponds
to \((-x_1 \land -x_2) \lor x_1\), and \(M_2\) corresponds to \((-x_1 \land x_2 \land x_3) \lor (-x_1 \land -x_2) \lor (x_1 \land x_2) \lor
(x_1 \land -x_2 \land -x_3)\). The result of the AND operation is \((-x_1 \land -x_2 \land -x_3) \lor (x_1 \land -x_2 \land -x_3) \lor
(x_1 \land x_2)\). In the product construction, the initial pair \((q_1, q_7)\) is entered in the queue \(S\).
Then its next state is computed. At this point the queue \(S\) contains \((q_2, q_8), (q_3, q_9)\). The
successor state of \((q_2, q_8)\) is computed, and the queue becomes \((q_3, q_9), (q_4, q_{10}), (q_4, q_{11})\).
Hence, the states are generated in the order \(q_{14}, q_{15}, ..., q_{21}\).
4.2. Minimization

After the product generation phase, we must minimize the resulting automaton. Since the graphs involved are directed acyclic graphs, we do not need to use the completely general $n \cdot \log(n)$ minimization algorithm described in [1]. Instead, we can use a variant of the linear algorithm for tree isomorphism [1]. In the minimization phase, states are processed starting at bottom level working upward, since the determination of whether two states should be merged into an equivalence class is based on the equivalence of their successor states. First, the final states (the bottom level nodes) are processed. Next, the states which have an edge to the final state are processed, and so on. Thus the order in which the states are processed in this phase is the reverse of the order in which they were generated during the product phase.

For the product machine in Figure 5, the states are processed in the order of $q_{21}, q_{30}, ..., q_{14}$. Although our algorithm does not label states with the names of the Boolean variables they represent, we can identify each state $q$ with the ordered-pair $(\delta(q,0), \delta(q,1))$ which we call edge-pair of $q$. For example, the edge-pair of $q_{20}$ is $(q_{21}, q_{21})$. The sequence of steps in this procedure is shown below. The minimal automaton is given in Figure 5.

1. $q_{21}$ is processed and is registered as the unique final state.
2. The edge-pair for $q_{20}$ is $(q_{21}, q_{21})$, and $q_{20}$ is registered as unique.
3. The edge-pair for $q_{19}$ is $(q_{21}, \bot)$ and is registered as unique.
4. The edge-pair for $q_{18}$ is $(q_{21}, \bot)$. Since this is the same as for $q_{19}$, we set $q_{18} = q_{19}$.
5. The edge-pair for $q_{17}$ is $(\bot, \bot)$, which is deleted.
6. The edge-pair for $q_{16}$ is $(q_{19}, q_{20})$ and is registered as unique.
7. The edge-pair of $q_{15}$ is $(q_{19}, \perp)$ and is registered as unique.

8. The edge-pair of $q_{14}$ is $(q_{15}, q_{16})$ and is registered as unique.

The minimization algorithm is summarized in Figure 6. The same hash function is used as in the product generation phase. To reduce the memory consumption, we keep a global binary decision diagram whose states represent equivalence classes of states of the reduced automaton.

For each state of the product machine,
starting at the bottom and working upward, do Begin

Check whether the state has already been stored as a global state;
If the state is new, then register the state as a global state;
Otherwise, mark the state as previously generated,
and store a pointer to the corresponding global state;

End;

Fig. 6 Minimization algorithm.

5. Parallel Implementation

5.1. Implementation

We now describe how the basic algorithm outlined in the previous section can be implemented on shared memory multiprocessor. To illustrate the procedure we consider the following example.

$$f(x_1, x_2, x_3, x_4) = (x_1 \land x_2) \lor (x_3 \land x_4) \lor (\neg x_1 \land x_4)$$

The first step is to determine the level of each node in the parse tree for the formula (see Figure 7). The leaf nodes of the tree are input variables; the non-leaf nodes correspond to the Boolean operators that occur in the formula. The level of each node is determined by the rule:

1. The level of an input variable is 0.
2. The level of a non-leaf node is $\text{max}(l_1, l_2) + 1$, where $l_1$ and $l_2$ are levels of its operands.

Since the binary decision diagram for each sub-formula has a transition for each Boolean variable in the entire formula, we can process input variables on level 1 immediately. After the level 1 nodes have been completed, we can process Boolean operations at level 2, and so on. In general, we can process level $i$ nodes as soon as the level $i - 1$ nodes have
been completed. Operations at the same level in the tree can be performed in parallel, since they do not conflict. Each such operation is performed on a separate processor since synchronization between processors is very time consuming.

Since some levels have only a few operations that can be performed in parallel, we divide the operations on such levels into several sub-operations that can be performed in parallel so that there will not be as many idle processors. During the product construction phase for such an operation, the 0- and 1-successors of the initial pair \((q_0^1, q_0^2)\) are generated as shown in Figure 5. The product and minimization phase for each of the two parts can be performed in parallel. However, the results of the two parallel computations must be merged to obtain the minimal automaton for the original operation. The merge operation is performed automatically during the minimization phase, so that a special procedure is not needed for this step.

An example of this procedure is shown in Figure 8. First, processor \(P_1\) expands the 0- and 1-successors of the initial pair. Processor \(P_2\) takes the 0-successor \((q_2, q_8)\), generates the product automaton and minimizes this automaton. Processor \(P_3\) takes the 1-successor and does the same thing. After \(P_2\) and \(P_3\) have completed the minimization phase for their product automata, processor \(P_1\) minimizes \(q_{14}\).

If, in the example, we compute the 00-, 01-, 10- and 11-successors of the initial pair, then the original operation can be divided to four parts whose product and minimization phases can be performed in parallel. Three merges are needed to reassemble the parts. In a similar manner we can divide a single operation into 8 parts and 5 merges to obtain an even higher degree of parallelism.

The following program text in Figure 9 illustrates the structure of the parallel processes. The algorithm is intended for a shared memory multiprocessor and would require significant modification for some other type of parallel architecture. The sequence of operations is stored in a array that can be accessed by all of the processors.
Product Construction Phase

Processor $P$

Minimization Phase

Processor $P'$

Processor $P''$

Fig. 8 Decomposition of an operation.

while (op_count < num_op) do
begin
  mutex_lock(op_lock);
  if (op_count < num_op) then
    begin i = op_count; op_count = op_count + 1; end
  mutex_unlock(op_lock);
  wait until (the operands of the i-th operation have been computed);
  do the i-th operation, i.e.
  construct product automaton;
  minimize product automaton;
end;

Fig. 9 Process structure.

5.2. Performance Evaluation

Our program for building binary decision diagrams is implemented in C and uses the C-threads package [5] for parallel programming under the Mach operating system. Interlocks are used for process synchronization instead of general semaphores in order to avoid the expense associated with system calls. The program is organized so that locks are only needed for the hash table and the global tree nodes. Consequently contention for shared memory is light. The performance statistics that we describe below were obtained for an Encore Multimax with 16 processors and 96 megabytes of shared memory. Each processor is a National Semiconductor 32332 and is rated at roughly 2 Mips. We hope to run the program
Table 1 Evaluation of Multiplier Examples.

<table>
<thead>
<tr>
<th></th>
<th>8-bit</th>
<th>9-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td># of variables</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td># of operations</td>
<td>620</td>
<td>878</td>
</tr>
<tr>
<td># of levels</td>
<td>38</td>
<td>45</td>
</tr>
<tr>
<td>1 processor</td>
<td>98.1 sec</td>
<td>339.6 sec</td>
</tr>
<tr>
<td>2 processors</td>
<td>50.3 sec</td>
<td>178.0 sec</td>
</tr>
<tr>
<td>3 processors</td>
<td>34.0 sec</td>
<td>122.3 sec</td>
</tr>
<tr>
<td>4 processors</td>
<td>25.8 sec</td>
<td>93.8 sec</td>
</tr>
<tr>
<td>5 processors</td>
<td>21.1 sec</td>
<td>76.7 sec</td>
</tr>
<tr>
<td>6 processors</td>
<td>18.1 sec</td>
<td>66.6 sec</td>
</tr>
<tr>
<td>7 processors</td>
<td>15.6 sec</td>
<td>57.9 sec</td>
</tr>
<tr>
<td>8 processors</td>
<td>14.2 sec</td>
<td>52.3 sec</td>
</tr>
<tr>
<td>9 processors</td>
<td>13.0 sec</td>
<td>47.4 sec</td>
</tr>
<tr>
<td>10 processors</td>
<td>12.1 sec</td>
<td>44.0 sec</td>
</tr>
<tr>
<td>11 processors</td>
<td>11.4 sec</td>
<td>41.0 sec</td>
</tr>
<tr>
<td>12 processors</td>
<td>10.9 sec</td>
<td>38.4 sec</td>
</tr>
<tr>
<td>13 processors</td>
<td>10.3 sec</td>
<td>36.1 sec</td>
</tr>
<tr>
<td>14 processors</td>
<td>9.8 sec</td>
<td>34.0 sec</td>
</tr>
<tr>
<td>15 processors</td>
<td>9.2 sec</td>
<td>32.4 sec</td>
</tr>
</tbody>
</table>

on larger multiprocessors in the near future. We have permission to run the program on the 64 processor RP3 at IBM Yorktown Heights Research Center. We should have statistics for this machine by the time of the conference.

Multipliers were used to evaluate the program since the binary decision diagrams for these circuits are known to grow quite rapidly (exponentially in the size of the operands, in fact). Table 1 shows the execution time to construct binary decision diagrams for multipliers with 8 (16 boolean variables) and with 9 bits (18 boolean variables).

The table shows that the minimum execution time on the Multimax with several processors is about 10-times smaller than the execution time with a single processor. The time for a single processor is roughly the same as the (sequential) program for constructing binary decision diagrams that is described in [6]. The graphs in Figures 10 and 12 show how the execution time varies with the number of processors, and the graphs in Figures 11 and 13 shows the rate of speed-up (the execution time for 1 processor / the execution time for \( n \) processor).
Summary and Directions for Future Research

This paper describes a parallel algorithm for constructing binary decision diagrams. The algorithm treats binary decision graphs as minimal finite automata. The automaton for a Boolean function with OR as its main operator (AND operator) is obtained by forming the union (intersection) of the regular sets associated with its operands. The union and intersection operations are implemented by a product construction on the minimal automata for the regular sets. After each product construction step the automaton must be re-minimized.

The parallel algorithm is designed so that it is possible to find the minimal representations for several Boolean operations in parallel. The level of each operator is determined. Operations on the same level can be performed in parallel without any communication between processors. If there are relatively few operations in one level, then we divide product construction step into several sub-operations and merge the results. This method works well in practice because it minimizes the amount of locking that is required.

Preliminary experiments show that our parallel algorithm is roughly 6 times faster than with a single processor. The algorithm has a fairly simple structure so it ought to be possible to adapt it to other shared memory architectures as well. Moreover, it should be possible to obtain even greater speedups by using more sophisticated data structures and coding techniques.

We plan to use this algorithm as part of a verification system for finite state concurrent systems (hardware controllers, communications protocols, etc.) that uses a technique called Symbolic Model Checking [3,4]. When synchronization or communication is possible among several finite state processes, the number of system states can be quite large. By using a sequential implementation of binary decision graphs to provide a concise representation for large global state-transition graphs, we have already been able to verify a pipelined circuit with as many as $10^{20}$ states. Since constructing binary decision diagrams is the most time consuming part of the verification procedure, we should be able to handle even larger finite state systems in the future.

References


Checking: $10^{20}$ States and Beyond. In Proceedings of Logic in Co.
To appear.

sity, Pittsburgh, PA 15213, June 1988.
