A Parallel Algorithm for Constructing Binary Decision Diagrams

S. Kimura
E. M. Clarke
July 1990
CMU-CS-90-148

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was partially supported by National Science Foundation grant CCR-87-226-33. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or the U.S. Government.

KeyWords: Parallel Algorithms, Binary Decision Graphs and Diagrams, Shared Memory Multiprocessors, Computer Aided Design.

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 [1] 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 fifteen processors our algorithm is almost an order of magnitude faster on some examples than the program described in [2]. 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 [1]. We view the binary decision diagram for an n-argument Boolean function f as the minimal finite state automaton 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.

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 logically 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.

Our approach to binary decision diagrams uses some simple ideas from finite automata theory. An n-argument Boolean function can be 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 a 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. For example, 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.

We believe that there are several 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 operations. 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.

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 implementation of Boolean operations as operations on finite automata. Section 4 describes the algorithm for building the product automaton and minimizing it. Section 5 describes the parallel algorithm and gives performance statistics for a number of examples. Section 6 shows a method to manipulate the construction of BDD's with large number of nodes. The paper concludes in Section 7 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 Σ. 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, Σ is the alphabet for strings, δ 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 $a_1 a_2 ... a_n$ 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, $M = (\{q_0, q_1, q_2, q_3, q_4, q_5, \perp\}, \{0, 1\}, \delta, q_0, \{q_5\})$ accepts $\{010, 110, 111\}$, where δ is defined as $\delta(q_0, 0) = q_1$, $\delta(q_0, 1) = q_2$, $\delta(q_1, 0) = \perp$, $\delta(q_1, 1) = q_3$, $\delta(q_2, 0) = \perp$, $\delta(q_2, 1) = q_4$, $\delta(q_3, 0) = q_5$, $\delta(q_3, 1) = \perp$, $\delta(q_4, 0) = \delta(q_4, 1) = q_5$, $\delta(q_5, 0) = \delta(q_5, 1) = \perp$, and $\delta(\perp, 0) = \delta(\perp, 1) = \perp$. $\perp$ is called a *sink* state. The representation of δ 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,

$$f(x_1, x_2, x_3) = \begin{cases} 1, & \text{if } (x_1, x_2, x_3) \text{ is (0, 1, 0), (1, 1, 0) or (1, 1, 1);} \\ 0, & \text{otherwise;} \end{cases}$$

is a Boolean function with three Boolean variables. The value of the function could, of course, also be given by a Boolean expression $f(x_1, x_2, x_3) = (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2)$. 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 $a_1 a_2 ... a_n$, then each set of

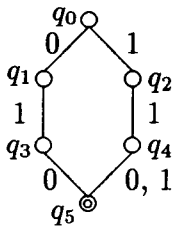


Fig.1 A binary decision diagram accepting {010, 110, 111}.

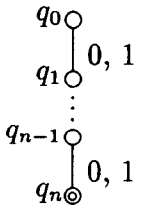


Fig.2 A binary decision diagrams accepting all strings.

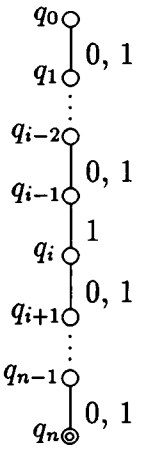


Fig.3 A binary decision diagram corresponding to x_i.

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 Σ), 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. $f_U(x_1, \ldots, x_n) = 1$ for all values of $x_1, \ldots, x_n$. 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 δ_U is defined as $\delta_U(q_i, 0) = \delta_U(q_i, 1) = q_{i+1}$. 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 $x_1, \ldots, x_n$, corresponds to the empty language and is accepted by a finite automaton $M_\emptyset = (\emptyset, \{0, 1\}, \delta_\emptyset, \emptyset, \emptyset)$.

Finally, the function $f_i(x_1, ..., x_n) = x_i$ corresponds to the set $\{0, 1\}^{i-1}1\{0, 1\}^{n-i}$. 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 δ_i is defined as $\delta_i(q_j, 0) = \delta_i(q_j, 1) = q_{j+1}$ 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.

Any Boolean function can be described using the above functions and Boolean operations, and a BDD corresponding to any Boolean function can be constructed from the above BDD's and operations on BDD's corresponding to Boolean operations.

3. Implementing Boolean Operations on Binary Decision Diagrams

Let $M_1 = (Q_1, \{0, 1\}, \delta_1, q_0^1, F_1)$ and $M_2 = (Q_2, \{0, 1\}, \delta_2, q_0^2, F_2)$ be the binary decision diagrams for two n-variable Boolean functions f_1 and f_2, $\perp_1$ be the sink state in Q_1, and $\perp_2$ be the sink state in Q_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($\wedge$), OR($\vee$), NOT($\neg$), and EXOR($\oplus$).

We consider the AND operation first. The set of strings over $\{0,1\}$ that satisfy $f_1 \wedge f_2$ corresponds to the intersection of sets accepted by 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. $M = (Q_1 \times Q_2 \cup \{\perp\}, \{0, 1\}, \delta_\wedge, (q_0^1, q_0^2), F_1 \times F_2)$, where $\perp$ denotes the sink state for the product automaton. $\delta_\wedge$ is defined as:

$$\delta_\wedge((q_1, q_2), a) = \begin{cases} (\delta_1(q_1, a), \delta_2(q_2, a)) & \text{if } \delta_1(q_1, a) \neq \perp_1 \text{ and } \delta_2(q_2, a) \neq \perp_2 \\ \perp & \text{otherwise} \end{cases}$$

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. $M = (Q_1 \times Q_2 \cup \{\perp\}, \{0, 1\}, \delta_\vee, (q_0^1, q_0^2), (F_1 \times Q_2) \cup (Q_1 \times F_2))$, where $\delta_\vee$ is defined as:

$$\delta_\vee((q_1, q_2), a) = \begin{cases} (\delta_1(q_1, a), \delta_2(q_2, a)) & \text{if } \delta_1(q_1, a) \neq \perp_1 \text{ or } \delta_2(q_2, a) \neq \perp_2 \\ \perp & \text{otherwise} \end{cases}$$

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 $M = (Q_U \times Q_1 \cup \{\perp\}, \{0, 1\}, \delta_\neg, (q_0^U, q_0^1), F_U \times (Q_1 - F_1))$, 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 $M = (Q_1 \times Q_2 \cup \{\perp\}, \{0, 1\}, \delta_\oplus, (q_0^1, q_0^2), F_1 \times (Q_2 - F_2) \cup (Q_1 - F_1) \times F_2)$, 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 $M = (Q, \{0,1\}, \delta, q_0, F)$ 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: the computation of $(\delta_1(q_1,0), \delta_2(q_2,0))$ and $(\delta_1(q_1,1), \delta_2(q_2,1))$. The most time-consuming part of this procedure is deciding whether a pair is new or not. By using a hash method with chaining we can make this test take essentially constant time. The hash function that we use is given by

$$hash_{prod}(q_1, q_2) = mod(q_1 * (HASH\_SIZE/2) + q_2, HASH\_SIZE),$$

where q_1 and q_2 are integer values for the state pointers. The size of the hash table (the parameter $HASH\_SIZE$) and the hash function are critical factors in determining the execution time of this phase of the algorithm.

```
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 ∈ {0,1} do
        Begin
                Compute the pair of successors (δ_1(q_1,a), δ_2(q_2,a));
                If this pair is new, then
                        add the pair to S, and allocate a new state.
                Connect the a-edge from the state corresponding to (q_1, q_2) to
                        the state corresponding to (δ_1(q_1,a), δ_2(q_2,a));
        End;
End;
```

Fig.4 Construction of the product automaton.

Since we use a hash method with chaining, each state is recorded in a linked data structure with 3 fields: the first field (*edge0*) holds a pointer to the 0-successor of the state, the second field (*edge1*) holds a pointer to the 1-successor of the state and the third field holds a pointer

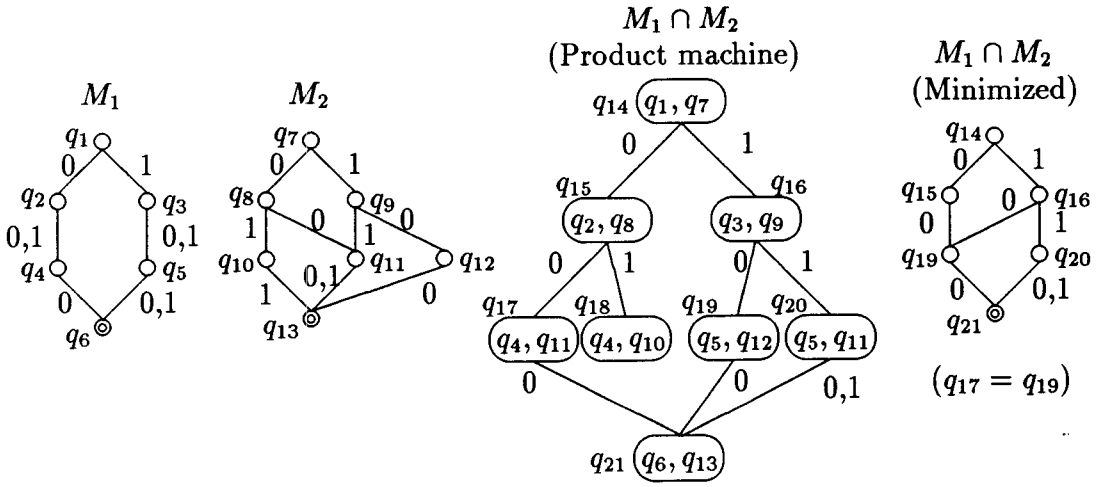


Fig.5 Product generation and minimization.

to a state with the same hash key. It should be mentioned that we need no special memory for a state pair.

Let state q correspond to a state pair (q_1, q_2), state q' correspond to a state pair $(\delta_1(q_1, 0), \delta_2(q_2, 0))$, and state q" correspond to a state pair $(\delta_1(q_1, 1), \delta_2(q_2, 1))$. First, a data cell corresponding to a state q is allocated so that $edge0$ of q holds a pointer to q_1 and $edge1$ holds a pointer to q_2. Then q is registered in a hash table and placed in a queue. If the state pair (q_1, q_2) is generated as a next state of some state, then the hash key is computed and the hash entry is checked. Since the state pair is already registered as q in the hash table, there is no need to register the pair. In this manner, the data cell of q is used for the occurency check of the same state pair.

After the state q is dequeued, and state pairs corresponding to q' and q" are calculated, $edge0$ ($edge1$) of q is overwritten to a pointer to q' (q"). Since the state transition graph is acyclic, if we generate state pairs in a breadth first manner using a queue as shown in Figure 4, the same state pair as (q_1, q_2) will not be generated after the pair is dequeued, and we need not keep the data of the pair for the occurency check after the dequeue operation. Thus, our overwrite 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 (corresponding to an *AND* operation in the original formula). M_1 corresponds to $(\neg x_1 \wedge \neg x_3) \vee x_1$, and M_2 corresponds to $(\neg x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2) \vee (x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3)$. The result of the AND operation is $(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2)$. 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). Next, the successor 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.

The minimization algorithm is summarized in Figure 6. To reduce the memory consumption, we keep a *global* binary decision diagram whose states represent equivalence classes of states of the reduced automaton. The same hash mechanism is used for the occurency check of the new global state as in the product generation phase. The hash key for a state q is defined as $hash_{min}(q) = mod(\delta(q,0) * HASH\_SIZE + \delta(q,1), HASH\_SIZE)$ using the *edge-pair* $(\delta(q,0), \delta(q,1))$ of q.

```
For each state q of the product automaton
      in the reverse order of the generation do Begin
   Reset_Flag;
   For each global state with the same hash key as q do Begin
      If the edge-pair of the global state is the same as that of q then Begin
         Set_Flag;
         Break;
      End;
   End;
   If Flag is not set then Begin
      Allocate a global state cell;
      Copy the edge information from q to the global state;
   End;
   Mark q as registered, and store a pointer to the global state;
End;
```

Fig.6 Minimization algorithm.

For the product automaton in Figure 5, states are processed in the order of q_{21}, q_{20}, ..., q_{14} as shown below. The minimal automaton is also given in Figure 5.

1. q_{21} is processed and is registered as the unique final state.
2. The edge-pair (q_{21}, q_{21}) of q_{20} is new, and q_{20} is registered as unique.

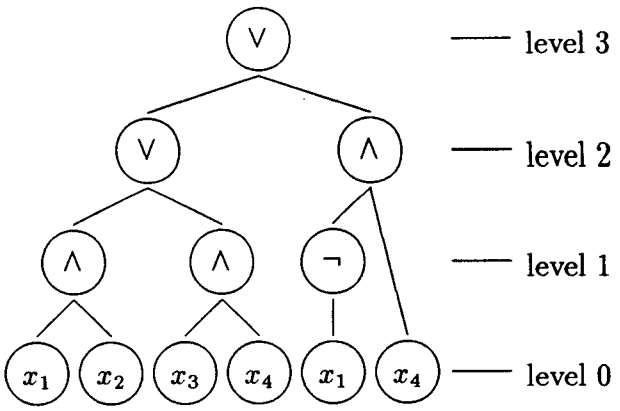


Fig.7 Levels of Boolean operations for
$(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (\neg x_1 \wedge x_4)$.

3. The edge-pair $(q_{21}, \perp)$ of q_{19} is new, and q_{19} is registered as unique.
4. The edge-pair of q_{18} is $(\perp, \perp)$, and it is impossible to reach the final state from q_{18}. Thus q_{18} is removed.
5. The edge-pair $(q_{21}, \perp)$ of q_{17} is the same as that of q_{19}. Thus we set $q_{17} = q_{19}$.
6. The edge-pair (q_{19}, q_{20}) of q_{16} is new, and q_{16} is registered as unique.
7. The edge-pair $(q_{19}, \perp)$ of q_{15} is new, and q_{15} is registered as unique.
8. The edge-pair (q_{15}, q_{16}) of q_{14} is new, and q_{14} is registered as unique.

5. Parallel Implementation

5.1. Implementation

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

$$f(x_1, x_2, x_3, x_4) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (\neg x_1 \wedge x_4)$$

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 $max(l_1, l_2) + 1$, where l_1 and l_2 are levels of its operands.

Since we initially generate binary decision diagrams for input variables, we can process operations at level 1 immediately. After the level 1 operations 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.

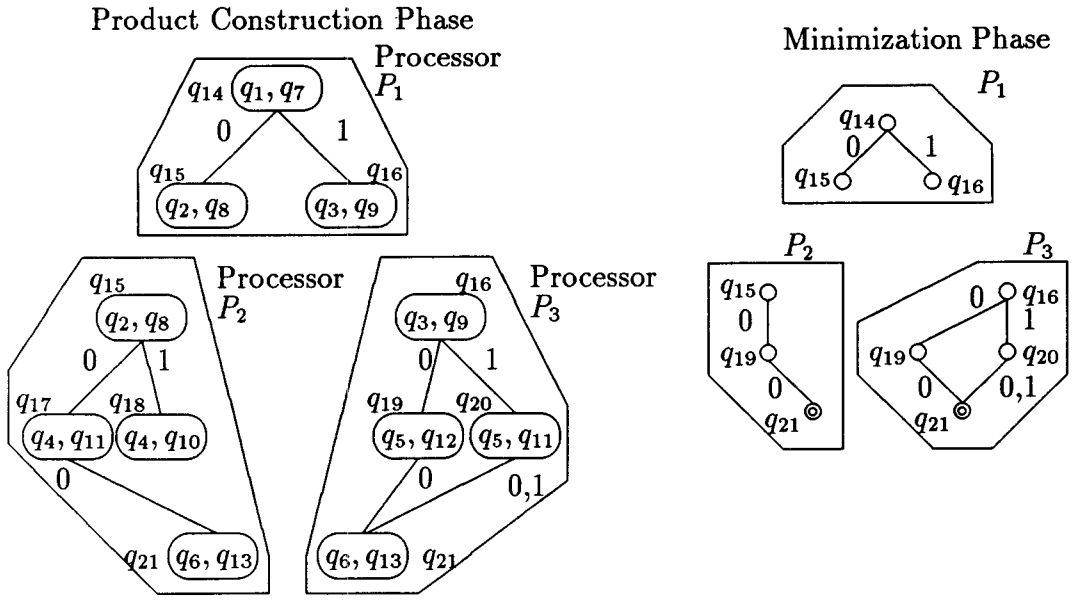


Fig.8 Decomposition of an operation.

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.

Some levels have only a few operations that can be performed in parallel. We divide operations on such levels into several sub-operations so that there will not be as many idle processors. The method is as follows.

In the product generation and minimization phase, the 0- and 1-successors of the initial pair (q_0^1, q_0^2) are generated. Then the product generation and minimization are performed for these two successors. After the minimization for these two successors is completed, the minimization of the root state corresponding to the initial pair is begun. Thus the product and minimization phase for each of these two successors (the 0- and 1-successors of (q_0^1, q_0^2)) can be performed in parallel. Note that the minimization phase guarantees the uniqueness of global states.

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 these 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.

Figure 9 shows a program executed in parallel with several processors. The algorithm is intended for a shared memory multiprocessor and would require significant modification for other types of parallel architecture. Data for operations and for global states can be accessed by all processors. *lock* and *unlock* are used to implement mutual exclusion.

```
While (operation exists) do Begin
    lock;
        take one operation if exists;
    unlock;
    If an operation can be taken then Begin
        wait until the operands of the operation have been computed;
        do the operation, i.e.
            construct product automaton;
            minimize the product automaton;
    End;
End;
```

Fig.9 Process structure.

In the parallel minimization algorithm, the following method is used to maintain consistency of global states.

```
Reset_Flag;
For each global state with the same hash key do Begin
    If the edge-pair of the global state is the same as that of q then Begin
        Set_Flag;      Break;
    End;
End;
If Flag is not set then Begin
    lock;
        For each global state with the same hash key
                which is generated until waiting to enter this part do Begin
            If the edge-pair of the global state is the same as that of q then Begin
                Set_Flag;      Break;
            End;
        End;
        If Flag is not set then Begin
            Allocate a new global state cell;
            Copy the edge information of q to the global state;
        End;
    unlock;
End;
```

Table 1 Evaluation of multiplier examples on Multimax.

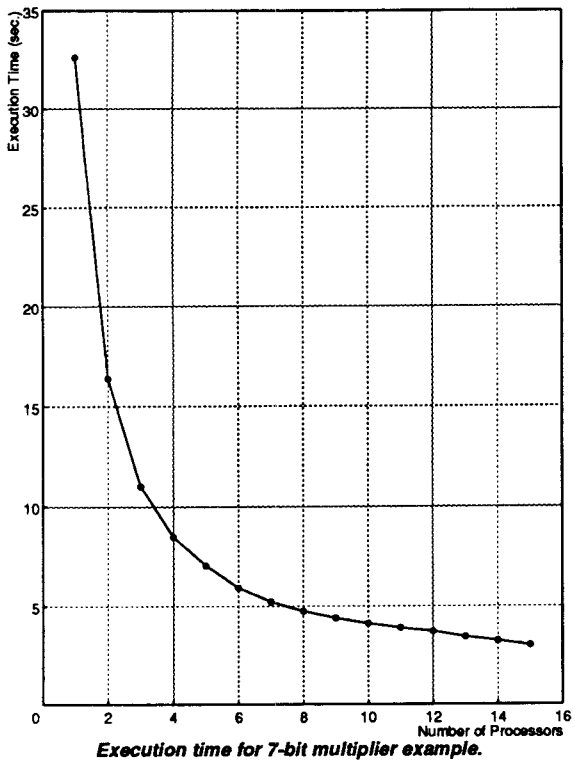
	7-bit	8-bit	9-bit	10-bit
# of variables	14	16	18	20
# of operations	478	620	878	1048
# of levels	32	38	45	51
1 processor	32.6 sec	98.1 sec	339.6 sec	1465.8 sec
2 processors	16.4 sec	50.3 sec	178.0 sec	732.1 sec
3 processors	11.0 sec	34.0 sec	122.3 sec	499.9 sec
4 processors	8.5 sec	25.8 sec	93.8 sec	384.1 sec
5 processors	7.0 sec	21.1 sec	76.7 sec	311.4 sec
6 processors	5.9 sec	18.1 sec	66.6 sec	265.5 sec
7 processors	5.2 sec	15.6 sec	57.9 sec	231.7 sec
8 processors	4.8 sec	14.2 sec	52.3 sec	211.8 sec
9 processors	4.4 sec	12.9 sec	47.4 sec	196.2 sec
10 processors	4.1 sec	12.1 sec	44.0 sec	181.1 sec
11 processors	3.9 sec	11.4 sec	41.0 sec	171.6 sec
12 processors	3.7 sec	10.9 sec	38.4 sec	163.2 sec
13 processors	3.5 sec	10.3 sec	36.1 sec	155.0 sec
14 processors	3.3 sec	9.8 sec	34.0 sec	148.5 sec
15 processors	3.0 sec	9.2 sec	32.4 sec	140.6 sec

5.2. Performance Evaluation

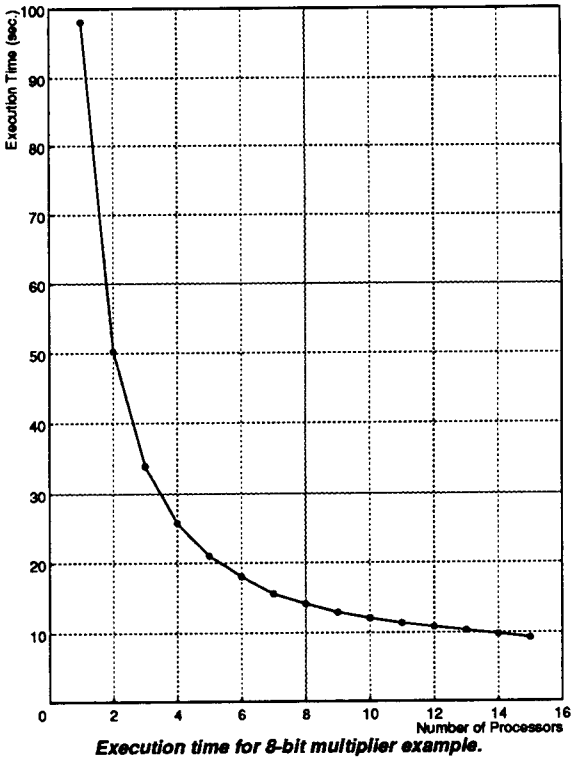
Our program for building binary decision diagrams is implemented in C and uses the *C-threads* package [5] for parallel programming under the Mach operating system. Interlocks are used for process synchronization instead of general semaphores in order to avoid the expense associated with system calls. The program is organized so that locks are only needed for the hash table for global states and for taking a Boolean operation to be executed. 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.

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 7 to 10 bits (14 to 20 Boolean variables). In the evaluation, a hash table with 1023 entries is used for the product generation, and a hash table with 32767 entries for the minimization.

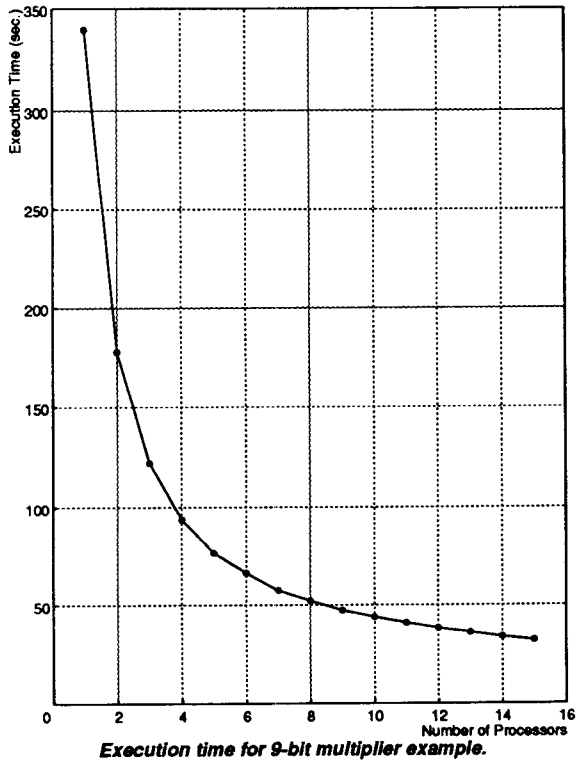
Table 1 shows that the minimum execution time on the Multimax with several processors is about 10-times smaller than the execution time with a single processor. The time for



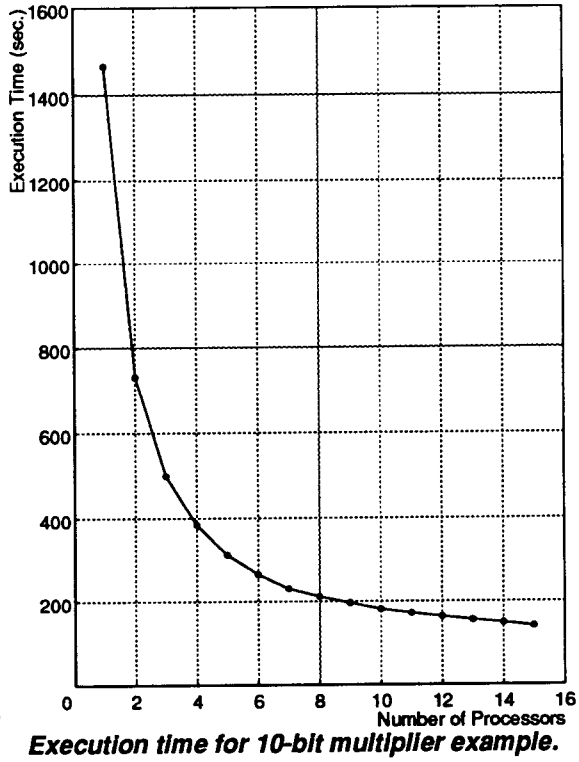
Execution time for 7-bit multiplier example.



Execution time for 8-bit multiplier example.

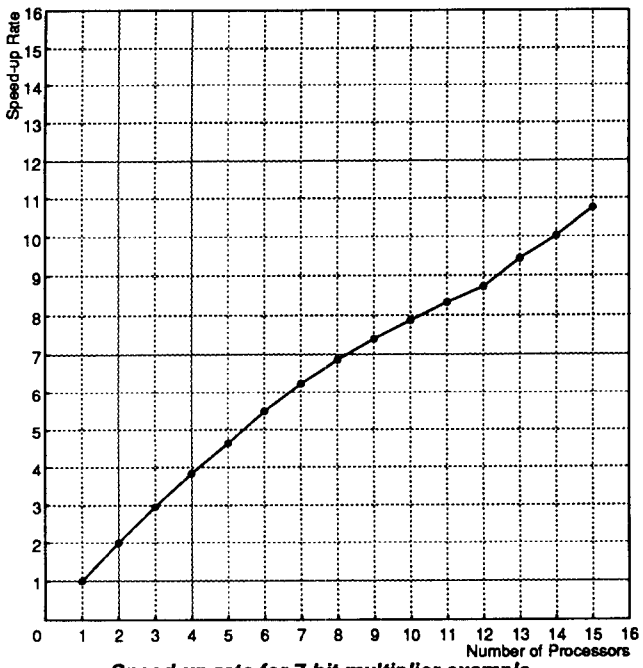


Execution time for 9-bit multiplier example.

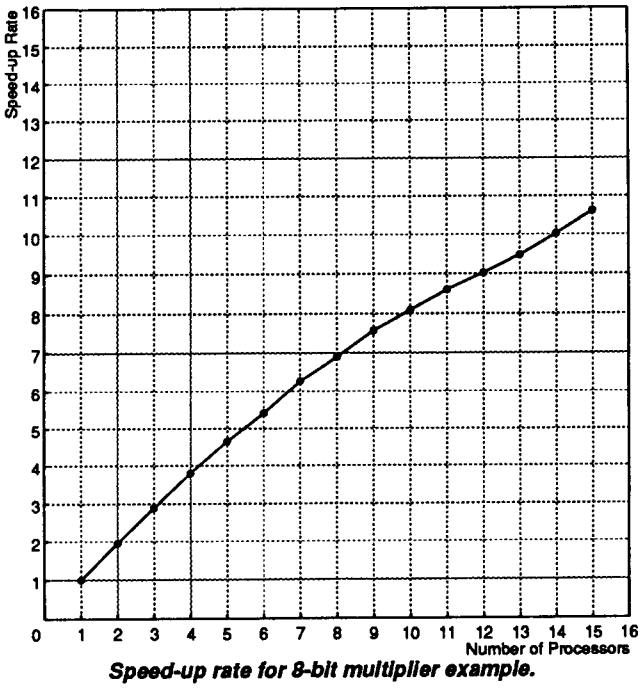


Execution time for 10-bit multiplier example.

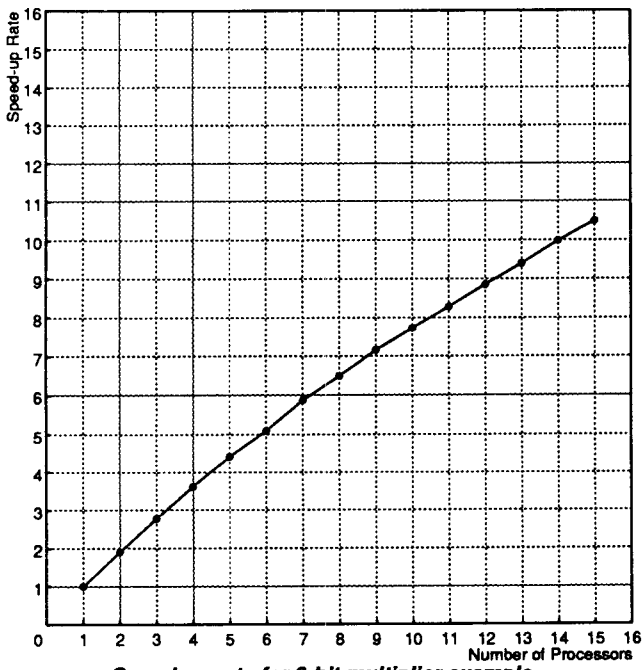
Fig.10 Execution time for multiplier examples on Multimax.



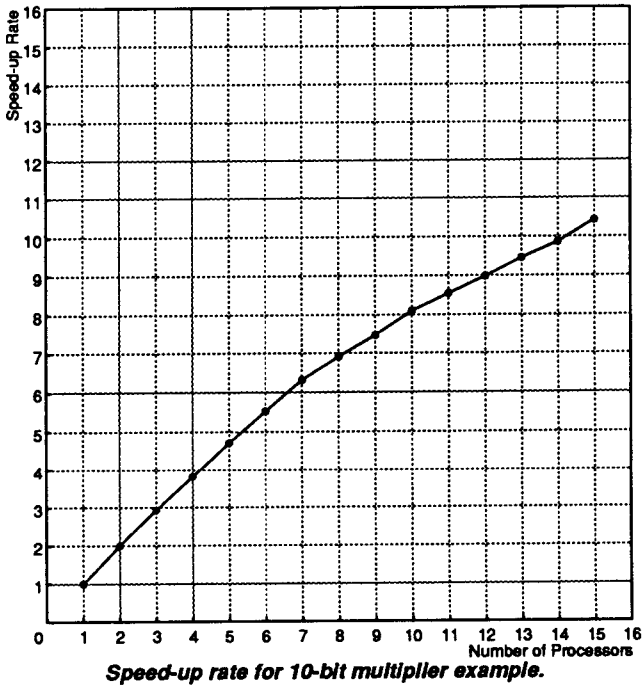
Speed-up rate for 7-bit multiplier example.



Speed-up rate for 8-bit multiplier example.



Speed-up rate for 9-bit multiplier example.



Speed-up rate for 10-bit multiplier example.

Fig.11 Speed-up rate for multiplier examples on Multimax.

a single processor is roughly the same as the (sequential) program for constructing binary decision diagrams described in [6]. The graphs in Figure 10 show how the execution time varies with the number of processors. The execution time is in reverse ratio with the number of processors. The graphs in Figure 11 show the rate of speed-up for these multipliers. The rate of speed-up is defined as (the execution time using 1 processor) / (the execution time using n processors). The rate is almost linear with the number of processors.

6. A Method to Manipulate Large Binary Decision Diagrams

6.1. Uniform Splitting Method

We have shown a parallel algorithm to construct BDD's. In several cases, the number of nodes exceeds the memory limitation of a computer and the construction of BDD's fails. To overcome the problem, we have devised a *divide-and-conquer* method. Since the parallel algorithm guarantees the high-speed execution, each part can be processed in reasonable time and the total execution time is also reasonable.

Let f and g be Boolean functions with n variables: $f(x_1, x_2, ..., x_n)$ and $g(x_1, x_2, ..., x_n)$. f and g can be described as follows.

$$f(x_1, x_2, ..., x_n) = (\neg x_1 \wedge f(0, x_2, ..., x_n)) \vee (x_1 \wedge f(1, x_2, ..., x_n))$$
$$g(x_1, x_2, ..., x_n) = (\neg x_1 \wedge g(0, x_2, ..., x_n)) \vee (x_1 \wedge g(1, x_2, ..., x_n))$$

In the following, $f(0, x_2, ..., x_n)$ is described as f_0, $f(1, x_2, ..., x_n)$ is described as f_1, $g(0, x_2, ..., x_n)$ is described as g_0, and $g(1, x_2, ..., x_n)$ is described as g_1.

The basic idea of the divide-and-conquer method is that a Boolean function f can be represented as a pair (f_0, f_1), and that Boolean operations can be done independently for each part of the pair.

It is easy to show that

$$f \wedge g = (\neg x_1 \wedge f_0 \wedge g_0) \vee (x_1 \wedge f_1 \wedge g_1)$$

and

$$f \vee g = (\neg x_1 \wedge (f_0 \vee g_0)) \vee (x_1 \wedge (f_1 \vee g_1))$$

We can also show that

$$\neg f = (\neg x_1 \wedge \neg f_0) \vee (x_1 \wedge \neg f_1)$$

Since

$$(\neg f) = \neg((\neg x_1 \wedge f_0) \vee (x_1 \wedge f_1)) = (x_1 \vee \neg f_0) \wedge (\neg x_1 \vee \neg f_1)$$

thus

$$(\neg f) = (\neg x_1 \wedge \neg f_0) \vee (x_1 \wedge \neg f_1) \vee (\neg f_0 \wedge \neg f_1)$$

On the other hand,

$$(\neg f)_{x_1=0} = \neg f_0 \vee (\neg f_0 \wedge \neg f_1) = \neg f_0$$

and

$$(\neg f)_{x_1=1} = \neg f_1 \vee (\neg f_0 \wedge \neg f_1) = \neg f_1$$

thus

$$(\neg f) = (\neg x_1 \wedge \neg f_0) \vee (x_1 \wedge \neg f_1)$$

The construction of BDD's can be divided to two parts using the above expansion method. For example, the construction of BDD's corresponding to

$$(f(x_1, ..., x_n) \wedge \neg g(x_1, ..., x_n)) \vee h(x_1, ..., x_n)$$

can be divided to two parts in the following manner. At first, we construct BDD's for the $(x_1 = 1)$ part. This corresponds to

$$(f(0, x_2, ..., x_n) \wedge \neg g(0, x_2, ..., x_n)) \vee h(0, x_2, ..., x_n)$$

Then we construct BDD's for the $(x_1 = 1)$ part. This corresponds to

$$(f(1, x_2, ..., x_n) \wedge \neg g(1, x_2, ..., x_n)) \vee h(1, x_2, ..., x_n)$$

Note that $f(0, x_2, ..., x_n)$ and $f(1, x_2, ..., x_n)$ are $n-1$ variable Boolean functions, and the number of nodes for describing BDD's for these functions may be smaller than that for the original functions f and g. Thus if we manipulate these parts (parts for $x_1 = 0$ and one for $x_1 = 1$) independently, the number of nodes for BDD's might be reduced.

In the same manner, if we consider the following expansion

$$\begin{aligned} f(x_1, x_2, x_3, ..., x_n) = &(\neg x_1 \wedge \neg x_2 \wedge f(0, 0, x_3, ..., x_n)) \vee (\neg x_1 \wedge x_2 \wedge f(0, 1, x_3, ..., x_n)) \vee \\ &(x_1 \wedge \neg x_2 \wedge f(1, 0, x_3, ..., x_n)) \vee (x_1 \wedge x_2 \wedge f(1, 1, x_3, ..., x_n)), \end{aligned}$$

then we can divide the original problem to 4 parts. We can also divide the problem to 8 parts, 16 parts, 32 parts, and so on.

For the number of nodes of the BDD's in each divided parts, we can show the following proposition.

Proposition 1 *If we divide a BDD for a Boolean function with n-variables into 2^d parts, then the number of nodes in each part is at most 2^{n-d}. Thus if the original BDD has 2^n nodes, then each part has 2^{n-d} nodes and the number of nodes is reduced by the factor 2^{-d}.*

We will show some experimental result on the number of nodes of each part.

# of parts	8-bit multiplier	16-bit adder (bad order)	16-bit adder (good order)
1	91220	852941	2313
2	54000	747000	2175
4	34000	574000	2038
8	20500	426000	1904
16	11000	268000	1776

The algorithm is summarized as follows, where d is the logarithm of the size of the division, i.e. the number of parts is 2^d. Since the construction of BDD's is based on BDD's for input variables and on Boolean operations, we only create BDD's for input variables correctly in each repetition. Let $d = 3$, and $f(x_1, ..., x_n)$ be the original Boolean formula. In the following loop, BDD's for $f(0, 0, 0, x_4, ..., x_n)$, $f(0, 0, 1, x_4, ..., x_n)$, $f(0, 1, 0, x_4, ..., x_n)$, etc. are constructed with respect to $i = 0, 1, 2, ...$

```
For i = 0 to 2^d − 1 do Begin
    Initialize data with respect to i:
        Set all operations to be undone;
        Create i-th part of BDD's for input variables:
            The depth d successor of the initial state of the original BDD
                with respect to the binary representation of i;
    While (operation exists) do Begin
        do the operation;
    End;
    remove all data generated in the construction;
End;
```

Note that if we need a BDD corresponding to the global outputs, we should keep the data corresponding to these outputs in each repetitions. Also note that if we only want to know whether two functions are equivalent or not, we need not to keep any data, since two Boolean functions are equivalent if and only if each part is equivalent. The check can be done in each repetitions. The parallel execution algorithm can be used in doing the sequence of Boolean operations in the above algorithm.

Applying the method to the multiplier examples, the result is as follows. In these examples, 10 processors are used. At first, an 8-bit multiplier example is shown to compare the data without splitting. The construction of BDD's for 13 to 16-bit multipliers cannot be done without splitting.

1. 8-bit multiplier
   The problem is divided into 4 parts. The number of nodes of BDD's for each part is about 34,000 (0.4 MB), and the execution time for the construction of each part is about 4.0 seconds. Total execution time is about 16.27 seconds.

2. 13-bit multiplier
   The problem is divided into 8 parts. The number of nodes of BDD's for each part is

about 3,600,000 (43 MB), and the execution time for the construction of each part is about 3,000 seconds. Total execution time is about 6.5 hours.

3. 14-bit multiplier
   The problem is divided into 32 parts. The number of nodes of BDD's for each part is about 2,600,000 (31.2 MB), and the execution time for the construction of each part is about 1,500 seconds. Total execution time is about 12.5 hours.

4. 15-bit multiplier
   The problem is divided into 128 parts. The number of nodes of BDD's for each part is about 2,500,000 (30 MB), and the execution time for the construction of each part is about 1,300 seconds. Total execution time is about 40 hours.

5. 16-bit multiplier
   The problem is divided into 2048 parts. The number of nodes of BDD's for each part is about 800,000 (9.6 MB), and the execution time for the construction of each part is about 220 seconds. Total execution time is about 100 hours.

6.2. Non-uniform Splitting Method

This section shows a non-uniform splitting method to manipulate BDD's with large number of nodes. The method is implemented as in the following procedure. In the procedure, *depth* is the logarithm of the size of the division, thus if $depth = d$ then the original construction is divided to 2^d parts. *pattern* denotes the identifier of the divided part. If the pattern is 011, then the procedure constructs a BDD for $f(0, 1, 1, x_4, ..., x_n)$. "$\|$" denotes a concatenation of strings.

```
Procedure Non_uniform_bdd(depth, pattern, operation_sequence);
    Reset_Flag;
    Create BDD's for input variables with respect to the depth and the pattern;
    While (operation exists) do Begin
        do the operation;
        If the number of nodes of BDD's exceeds the limit then Begin
            Set_Flag;
            Break;
        End;
    End;
    If Flag is set then Begin
        Non_uniform_bdd(depth + 1, pattern||0, operation_sequence);
        Non_uniform_bdd(depth + 1, pattern||1, operation_sequence);
    End;
End;
```

At first, this procedure is invoked with $(depth, pattern) = (0, \epsilon)$ (ϵ denotes the null string).

Non_uniform_bdd(0, ϵ, operation_sequence)

If the number of nodes of BDD's exceeds the limit, then that invokes itself twice with $(depth, pattern) = (1, 0)$ and $(depth, pattern) = (1, 1)$.

Non_uniform_bdd(1, 0, operation_sequence)
Non_uniform_bdd(1, 1, operation_sequence)

If the number of nodes of BDD's exceeds the limit in the latter case, then the latter one invokes itself twice with $(depth, pattern) = (2, 10)$ and $(depth, pattern) = (2, 11)$. In this point, the procedure is invoked like as follows.

Non_uniform_bdd(1, 0, operation_sequence)
Non_uniform_bdd(2, 10, operation_sequence)
Non_uniform_bdd(2, 11, operation_sequence)

In the above manner, splitting is done only when it is needed.

7. 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* (*AND*) as its main 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 at 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 the product generation 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 10 times faster than with a single processor. The execution time with a single processor is almost the same as that of a sequential algorithm ([6]). 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

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

[2] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of Design Automation Conf.*, 1990. To appear.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of Logic in Computer Science*, 1990. To appear.

[5] E. C. Cooper. C threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, Pittsburgh, PA 15213, June 1988.

[6] Allan L. Fisher and Randal E. Bryant. Performance of COSMOS on The IFIF Workshop Benchmarkes. In *Proceedings of IMEC Conference*, 1989.