A Parallel Algorithm for Constructing Binary Decision Diagrams

Shinji Kimura
Dept. of Electronics Engineering
Kobe University
Kobe, 657 Japan

Edmund M. Clarke
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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 [5]. When we construct a binary decision graph, our parallel 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* [1] 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 identified with the set of Boolean vectors that make it true. If we associate a Boolean vector as a string, then f can be represented by a finite set of strings. Since all finite languages are regular, there is the minimal finite automaton that accepts the 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 standard constructions from elementary automata theory can be used to build the binary decision diagram for the result of logical operations.

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.

[0]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.

2 Binary Decision Diagrams

We start with some simple definitions on 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*. The length of a string is the number of symbols in the string.

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 $a_1a_2...a_n$ 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, $M = (\{q_0, q_1, q_2, q_3, q_4, q_5, \perp\}, \{0, 1\}, \delta, q_0, \{q_5\})$ accepts {010, 110, 111}, where δ is defined as $\delta(q_0,0) = q_1$, $\delta(q_0,1) = q_2$, $\delta(q_1,0) = \perp$, $\delta(q_1,1) = q_3$, $\delta(q_2,0) = \perp$, $\delta(q_2,1) = q_4$, $\delta(q_3,0) = q_5$, $\delta(q_3,1) = \perp$, $\delta(q_4,0) = q_5$, $\delta(q_4,1) = q_5$, $\delta(q_5,0) = \perp$, $\delta(q_5,1) = \perp$, $\delta(\perp,0) = \perp$, and $\delta(\perp,1) = \perp$. $\perp$ is called a *sink* state. The representation of δ as a directed graph is shown in Figure 1. The sink state is not shown in the figure for simplicity.

A Boolean function f with n-variables is a function from $\{0,1\}^n$ to $\{0,1\}$. 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 f. Since all finite languages are regular, there is the *minimal* finite automaton accepting the language corresponding to f. The minimal automaton 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.

For example, a binary decision diagram in Figure 1 represents $f(x_1,x_2,x_3) = (\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2)$, the one in Figure 2 represents $f_U(x_1,...,x_n) = 1$ for all inputs, and the one in Figure 3 represents $f(x_1,...,x_n) = x_i$.

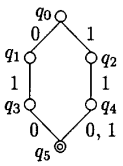

Fig.1 A binary decision diagram accepting {010, 110, 111}.

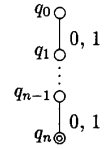

Fig.2 A binary decision diagrams accepting all strings.

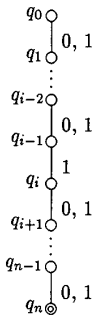

Fig.3 A binary decision diagram corresponding to x_i.

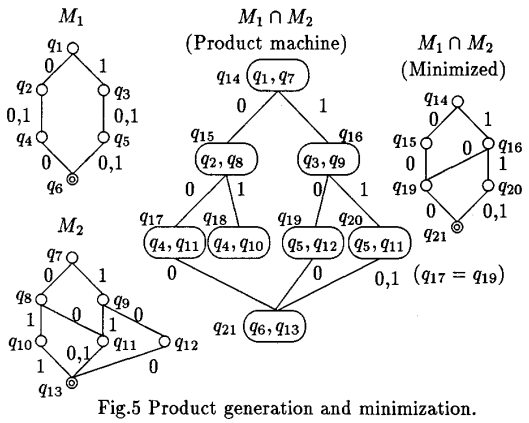

Fig.5 Product generation and minimization.

3 Boolean Operations

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 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 $L(M_1) \cap L(M_2)$ 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 $\delta_\wedge((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ if $\delta_1(q_1, a) \neq \perp_1$ and $\delta_2(q_2, a) \neq \perp_2$, and $\perp$ 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. $M = (Q_1 \times Q_2 \cup \{\perp\}, \{0, 1\}, \delta_\vee, (q_0^1, q_0^2), (F_1 \times Q_2) \cup (Q_1 \times F_2))$, where $\delta_\vee$ is defined as $\delta_\vee((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ if $\delta_1(q_1, a) \neq \perp_1$ or $\delta_2(q_2, a) \neq \perp_1$, and $\perp$ otherwise.

The NOT operation 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 operation $\oplus$ is also similar to the OR operation. The finite automaton for this operation 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 operations involves a product construction $M_1 \times M_2$. 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.

4 Product Automaton Generation

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 that 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.

```
Let the initial pair be (q0^1, q0^2);
Put the pair in the queue S, and allocate a new state for it;
While (S is not empty) Do Begin
  Dequeue a pair (q1, q2) from S;
  For symbol a ∈ {0,1} Do Begin
    Compute (δ1(q1, a), δ2(q2, a));
    If this pair is new, then
      add the pair to S, and allocate a new state.
    Connect the a-edge from a state
      corresponding to (q1, q2) to
      the state corresponding to (δ1(q1, a), δ2(q2, a));
  End;
End;
```

Fig.4 Construction of the product automaton.

Note that there are only two places where we need to take into account the types of the Boolean operation: 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 table with chaining, we can make this test take essentially constant time. The hash function that we use is given by

$$hash(q_1, q_2) = mod(q_1 * (hash\_size/2) + q_2, hash\_size),$$

where q_1 and q_2 are integer values for the state pointers.

An example illustrating this phase is shown in Figure 5, where the intersection of M_1 and M_2 is computed (corresponding to the *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 example, states are generated in the order q_{14}, q_{15}, ..., q_{21}.

5 Minimization

After the product generation phase, we must minimize the resulting automaton. In the minimization phase, states are processed

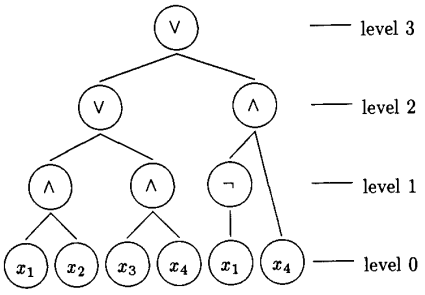

Fig.7 Levels of Boolean operations for $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (\neg x_1 \wedge x_4)$.

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 automaton in Figure 5, the states are processed in the order of q_{21}, q_{20}, ..., q_{14}. In the following, the *edge-pair* of q denotes the ordered-pair $(\delta(q,0), \delta(q,1))$. At first, q_{21} is processed and is registered as a unique final state. Next, q_{20} is processed and is registered as a unique state, since the edge-pair (q_{21}, q_{21}) of q_{20} is unique. q_{19} is also registered as unique. It is impossible to reach the final state from q_{18}, thus q_{18} is deleted. q_{17} is marked as the same as q_{19}, since the edge-pair of these states are the same. q_{16}, q_{15} and q_{14} are also processed, and a minimal finite automaton as shown in Figure 5 is obtained.

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 automaton,
starting at the bottom and working upward, do Begin
  Check whether the state has already been registered
    as a global state;
  If the state is new, then register the state
    as a global state;
  Otherwise, mark the state as *previously registered*,
  and store a pointer to the corresponding global state;
End;

Fig.6 Minimization algorithm.

6 Parallel 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

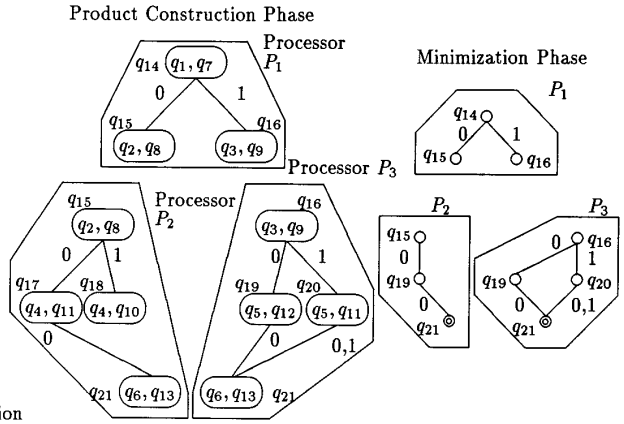

Fig.8 Decomposition of an operation.

operations 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. Operations at the same level in the tree can be performed in parallel, since they do not conflict.

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 corresponding to a Boolean operation, the 0- and 1-successors of the initial pair (q_0^1, q_0^2) are generated. Then the product generation and minimization are done for these two successors. After the minimization for these two successors is performed, the minimization of the root state corresponding to the initial pair is done. 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 with three merges. In a similar manner we can divide a single operation into 8 parts, 16 parts, etc.

In the implementation, all processors execute the same program:

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
6 processors	5.9 sec	18.1 sec	66.6 sec	265.5 sec
9 processors	4.4 sec	12.9 sec	47.4 sec	196.2 sec
12 processors	3.7 sec	10.9 sec	38.4 sec	163.2 sec
15 processors	3.0 sec	9.2 sec	32.4 sec	140.6 sec

take one operation;
wait until the operands have been calculated;
do the operation: product generation & minimization;

Operations (including *divide* and *merge* operations) are ordered from smaller levels to larger levels.

7 Performance Evaluation

Our program for constructing binary decision diagrams is implemented in C and uses the *C-threads* package [4] 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 global 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.

Two dimensional adder array 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 8191 entries is used for the product generation, and a hash table with 32727 entries is used for the minimization.

The table shows that the minimum execution time on the Multimax with 15 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 [5]. A graph in Figure 9 shows how the execution time varies with the number of processors for 10 bit multiplier. The execution time is in reverse ratio with the number of processors. Figure 9 also shows the rate of speed-up (= (the execution time using 1 processor) / (the execution time using n processors)). The rate is almost linear with the number of processors. Other examples show almost the same graphs.

8 Summary and 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 *AND* as its main operation (*OR* operation) is obtained by forming the *intersection* (*union*) 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 operation 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. Preliminary experiments show that our parallel algorithm is roughly 10 times faster than with a single processor.

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* [2, 3]. 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] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[2] 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.

[3] 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.

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

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

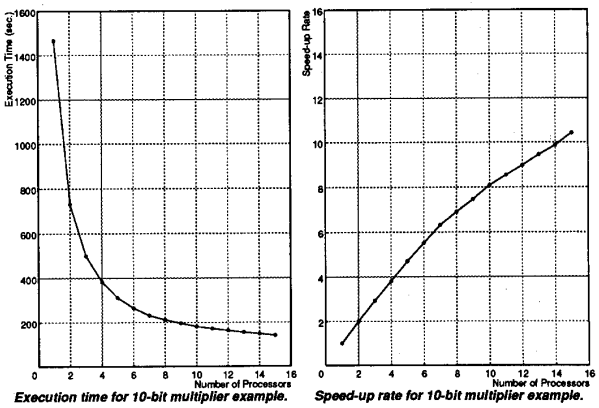

Fig.9 Execution time and speed-up rate for 10 bit multiplier.