Вісник Харківського національного університету
Серія «Математичне моделювання. Інформаційні технології. Автоматизовані системи управління»
№ 780, 2007, с.35-48

УДК 519.713

On development and implementation of a parallel Binary Decision Diagram package

S. M. Bogomolov, G. M. Zholtkevych
V. N. Karazin Kharkiv National University, Ukraine

Efficient manipulation of Boolean functions is an important component of many computer-aided design and verification tasks. An efficient parallel package for manipulating Boolean functions based on the reduced, ordered, binary decision diagram (ROBDD) representation has been developed. In this paper we consider main algorithms and data structures which are used in this package.

Introduction

Binary decision diagrams (BDD) are used in a wide variety of applications such as circuit verification [1], combinatorial problems[2] and other techniques where it is necessary to have an efficient means of representing and manipulating Boolean functions. Many of the current non-parallel BDD packages are based on the depth-first if-then-else (ite) algorithm [3]. Specialized programming techniques, such as attributed edges [4], dynamic variable reordering [5] and garbage collection [3], are often utilized in applications dealing with large BDD structures. However, these techniques are not always sufficient. In many applications of BDDs one should deal with very large data structures and that is why we must develop even more efficient algorithms. One of the ways to do this is to use parallel algorithms. In this work we consider algorithms and data structures for constructing and manipulating Binary Decision Diagrams which uses benefits of multiprocessor systems. Then we will be able to use computational resources of a few processors.

To date, parallel BDD implementations that have been developed include packages for a distributed shared memory (DSM) platform [6] and for vector processors [7]. This thesis describes a different type of parallel BDD library package developed for use in shared memory multi-processor systems.

Often, parallelism may be extracted from a particular algorithm in several different ways. For instance, [5] explores parallelism in breadth-first BDD traversals. In [8], parallelism in operation sequences is examined. This thesis explores parallelism in depth-first BDD traversals. Package library functions utilize multiple processors to perform operations on the BDD. These library functions are similar to the non-parallel versions on which they are based. Each processor executes its own thread. So the package simultaneously executes multiple threads of computation on a BDD. These techniques apply to a wide range of different BDD applications.

In Fig. 1 the typical process of verification is shown. In this work we consider the process of constructing Binary Decision Diagram on the basis of the System Model. In the future we would like to develop a verification environment which would integrate the results of this work.

Other aspects of BDD techniques and applications see in [9-14]. Among last works concerning our topic we mention [15-21].

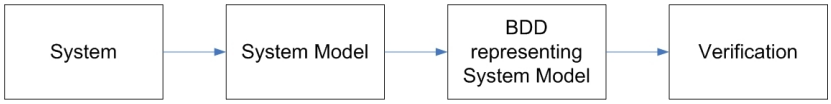

Fig. 1. System Verification Process

1 Binary Decision Diagrams

Let $x \to y_0, y_1$ be if-then-else operator (ITE operator) defined by

$$x \to y_0, y_1 = (x \wedge y_0) \vee (\overline{x} \wedge y_1)$$

hence, $t \to t_0, t_1$ is true if t and t_0 are true or if t is false and t_1 is true. We call t the test expression. All operators can easily be expressed using only the if-then-else operator and the constants 0 and 1. Moreover, this can be done in such a way that all tests are performed only on (un-negated) variables and variables occur in no other place. Hence this operator gives rise to a new kind of normal form. For example, $\neg x$ is $(x \to 0,1)$, $x \Leftrightarrow y$ is $x \to (y \to 1,0),(y \to 0,1)$. Since variables must only occur in tests the Boolean expressions x is represented as $x \to 1,0$.

Table	Name	Expression	Equivalent form
0000	0	0	0
0001	AND (F,G)	$F \cdot G$	$ite(F,G,0)$
0010	F>G	$F \cdot \overline{G}$	$ite(F,\overline{G},0)$
0011	F	F	F
0100	F<G	$\overline{F} \cdot G$	$ite(F,0,G)$
0101	G	G	G
0110	XOR (F,G)	$F \oplus G$	$ite(F,\overline{G},G)$
0111	OR (F,G)	$F + G$	$ite(F,1,G)$
1000	NOR (F,G)	$\overline{F + G}$	$ite(F,0,\overline{G})$
1001	XNOR (F,G)	$\overline{F \oplus G}$	$ite(F,G,\overline{G})$
1010	NOT (G)	$\overline{G}$	$ite(G,0,1)$
1011	F≥G	$F + \overline{G}$	$ite(F,1,\overline{G})$
1100	NOT (G)	$\overline{F}$	$ite(F,0,1)$
1101	F≤G	$\overline{F} + G$	$ite(F,G,1)$
1110	NAND (F,G)	$\overline{F \cdot G}$	$ite(F,\overline{G},1)$
1111	1	1	1

Fig. 2. All two variable functions described using ITE

Definition

An If-then-else Normal Form (INF) is a Boolean expression built entirely from the if-then-else operator and the constants 0 and 1 such that all tests are performed only on variables.

If we by $t[0/x]$ denote the Boolean expression obtained by replacing x with 0 in t then it is not hard to see that the following equivalence holds: $t = x \to t[1/x], t[0/x]$.

This is known as the Shannon expansion of t with respect to x. This simple equation has a lot of useful applications. The first is to generate INF from any expression t. If t contains no variables it is either equivalent to 0 or 1 which is an INF. Otherwise we form the Shannon expansion of t with respect to one of the variables x in t. Thus since $t[0/x]$ and $t[1/x]$ both contain one less variable than t, we can recursively find INFs for both of these; call them t_0 and t_1. An INF for t is now simply $x \to t_1, t_0$.

We have proved: Any Boolean expression is equivalent to an expression in INF.

Example

Consider the Boolean expression $t = (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$. If we find an INF of t by selecting in order the variables x_1, y_1, x_2, y_2 on which to perform Shannon expansions, we get the expression

$t = x_1 \to t_1, t_0$; $t_0 = y_1 \to 0, t_{00}$; $t_1 = y_1 \to t_{11}, 0$; $t_{00} = x_2 \to t_{001}, t_{000}$; $t_{11} = x_2 \to t_{111}, t_{110}$; $t_{000} = y_2 \to 0,1$; $t_{001} = y_2 \to 1,0$; $t_{110} = y_2 \to 0,1$; $t_{111} = y_2 \to 1,0$

Fig. 3 shows the expression as a tree. Such a tree is called a decision tree.

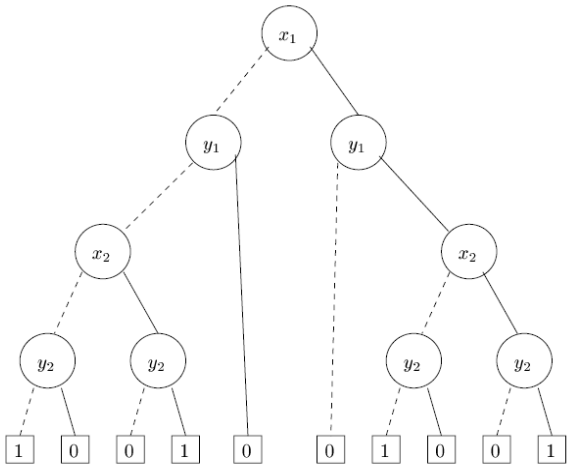

Fig. 3. A decision tree for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$.
Dashed lines denote low-branches, solid lines high-branches.

A lot of expressions are easily seen to be identical, so it is tempting to identify them. For example, instead of t_{110} we can use t_{000} and instead of t_{111} we can use t_{001}.

If we substitute t_{000} for t_{110} in the right-hand side of t_{11} and also t_{001} for t_{111}, we in fact see that t_{00} and t_{11} are identical, and in t_1 we can replace t_{11} with t_{00}.

If we in fact identify all equal subexpressions we end up with what is known as a binary decision diagram (a BDD). It is no longer a tree of Boolean expressions bt a directed acyclic graph (DAG).

Applying this idea of sharing, t can now be written as:

$t = x_1 \to t_1, t_0$; $t_0 = y_1 \to 0, t_{00}$; $t_1 = y_1 \to t_{00}, 0$; $t_{00} = x_2 \to t_{001}, t_{000}$; $t_{000} = y_2 \to 0,1$; $t_{001} = y_2 \to 1,0$

Each subexpression can be viewed as the node of a graph. Such a node is either terminal in the case of constants 0 and 1, or non-terminal. A non-terminal node has a low-edge corresponding to the else-part and a high-edge corresponding to the then-part. See Fig. 4. Notice, that the number of nodes has decreased from 9 in the decision tree to 6 in the BDD. It is not hard to imagine that if each of the terminal nodes were other big decision trees the savings would be dramatic. Since we have chosen to consistently select variables in the same order in the recursive calls during the construction of the INF of t, the variables occur in the same orderings on all paths from the root of the BDD. In this situation the binary decision diagram is said to be ordered (an OBDD). Fig. 3 shows a BDD that is also an OBDD.

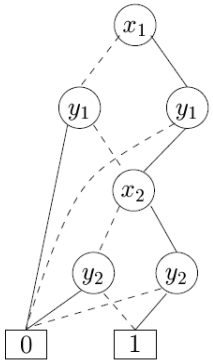

Fig. 4. A BDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ *with ordering* $x_1 < y_1 < x_2 < y_2$. *Low-edges are drawn as dotted lines and high-edges as solid lines.*

A Binary Decision Diagram (BDD) is a rooted, directed acyclic graph with

- one or two terminal nodes of out-degree zero labeled with 0 or 1, and
- a set of variable nodes u of out-degree two. The two outgoing edges are given by two function $low(u)$ and $high(u)$. In pictures, these are shown as dotted and solid lines, respectively. A variable $\mathrm{var}(u)$ is associated with each variable node.

A BDD is Ordered (OBDD) if on all paths through the graph the variables respect a given linear order $x_1 < x_2 < \ldots < x_n$. An (O)BDD is Reduced (R(O)BDD) if (uniqueness) no two distinct nodes u and v have the same variable name and low- and high- successor, i.e.,

- $\mathrm{var}(u) = \mathrm{var}(v)$, $low(u) = low(v)$, $high(u) = high(v)$ implies $u = v$, and
- (non-redundant tests) no variable node u has identical low- and high-successor, i.e., $low(u) \neq high(u)$.

ROBBDs have some interesting properties. They provide compact representations of Boolean expressions, and there are efficient algorithms for performing all kinds of logical operations on ROBBDs. They are based on the crucial fact that for any function $f : \mathrm{B}^n \to \mathrm{B}$ there is exactly one ROBDD representing it. This means, in particular, that there is exactly one ROBDD for the constant true (and constant false) function on B^n: the terminal node 1 (and 0 in case of false). Hence, it is possible to test in constant time whether an ROBBD is constantly true of false. (Recall that for Boolean expressions this problem is NP-complete.)

To make this claim more precise we must say what we mean for an ROBDD to represent a function. First, it is quite easy to see how the nodes u of an ROBDD inductively defines Boolean expressions t^u: A terminal node is a Boolean constant. A non-terminal node marked with x is an if-then-else expression where the condition is x and the two branches are the Boolean expressions given by the low- or high-son, respectively:

$t^0 = 0$; $t^1 = 1$; $t^u = \mathrm{var}(u) \to t^{high(u)}, t^{low(u)}$, if u is a variable node.

Moreover, if $x_1 < x_2 < \ldots < x_n$ is the variable ordering of the ROBDD, we associate with each node u the function f^u that maps $(b_1, b_2, \ldots, b_n) \in \mathrm{B}^n$ to the truth value of $t^u[b_1 / x_1, b_2 / x_2, \ldots, b_n / x_n]$. We can now state the key lemma:

Lemma (Canonicity lemma)

For any function $f : \mathrm{B}^n \to \mathrm{B}$ there is exactly one ROBDD u with variable ordering $x_1 < x_2 < \ldots < x_n$ such that $f^u = f(x_1, \ldots, x_n)$.

2 Constructing and Manipulating ROBDDs

Due to enormous size of systems to be verified efficient algorithms and data structures shall be developed. Let us consider main algorithms and data structures for manipulating Binary Decision Diagrams.

2.1 Data Structures

Hash Table

A hash table associates a value with a key. A hash function applied to the key selects which of N linked lists the key, value pair is stored. The load factor of a hash table is defined $\alpha = n / N$, where n is the number of keys stored in the table.

Computed table

A memory function for the function F is a table of values $(x, F(x))$ that the function has already computed. If F is called with argument x again, $F(x)$ is

returned without any computation. We use a memory function to improve the performance of ite. We call the memory function for ite the computed-table (cache table). The computed-table maps three nodes F, G, H to the result node $ite(F,G,H)$ once this result has been computed. We implement the computed-table using a hash table.

Uncomputed table

The uncomputed table is a hash table used to keep track of ongoing computations. The uncomputed table is necessary because results in an ongoing recursive BDD computation are almost likely not immediately available. For instance, the current thread of computation may be temporally stopped because it requires a result from another thread. Thus, it is necessary to keep a record of having started a computation to prevent it from being started multiple times.

The uncomputed table provides a forward mechanism that allows threads of computation to send the information about the completion of computation to other threads waiting for this event.

Functions also store their returning value in this table.

Hash-based cache

A hash-based cache is a hash table where collision chain is not used to resolve collisions. Instead, at insert time, any existing element at the particular array position is discarded and replaced with the new entry. At lookup time, if the element does not match the stored key, a cache miss occurs and no element is returned.

Unique Table

As was already mentioned each node in the ROBDD represents a Boolean function, and is written using a capital letter, such as F, and can be denoted by the triple (v,G,H), where v is the top variable of F, G is the node connected to the 1 (or then) edge of F, and H is the node connected to the 0 (or else) edge of F.

A hash table imposes a strong canonical form on the nodes in the ROBDD, so that each node in the ROBDD represents a unique logic function. Hence, this hash table is called the unique-table.

The unique-table maps a triple (v,G,H) to an ROBDD node $F=(v,G,H)$. Each node in the ROBDD has an entry in the unique-table. Before a new node is added to the ROBDD, a lookup in the unique table determines if a node for that function already exists. If so, the existing node is used. Otherwise, the new node is added to the ROBDD and a new unique-table entry is made. By assumption, when we create a new node F, the nodes G and H will already obey the strong canonical form. Hence, the function F exists in the ROBDD if, and only if, the triple (v,G,H) is already in the unique table, thus maintaining the strong canonical form.

The unique table allows a single multi-rooted DAG to represent all of the user's formulae simultaneously.

The relationship between a unique table and a cache table is shown in Fig. 5.

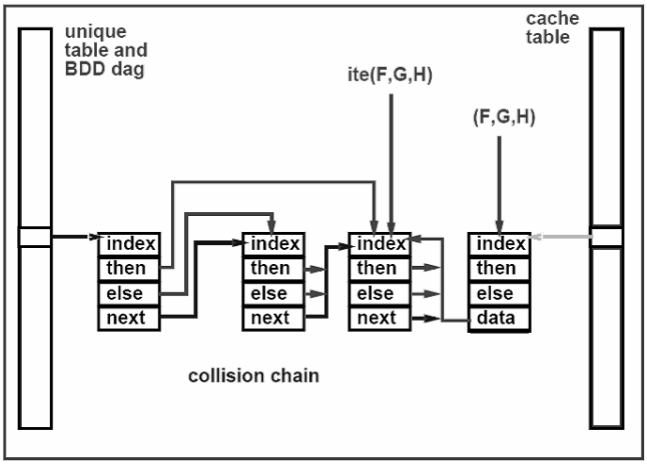

Fig. 5. Unique table and cache table

Work Queue

The work queue facilitates parallel tree traversals and is the mechanism by which simultaneous threads of computation may be executed. It is an unordered, prioritized queue to which any thread may add tasks.

Stack Frames

Some mechanism is required to temporarily store a portion of the recursive computation until it can be continued at a later time. In other words, threads continually process their work queue and do not wait for results from other processors. To meet these needs, a persistent data structure, the stack frame, was constructed so that threads could obtain the required information and perform steps of the calculations at their own convenience.

Each stack frame keeps state information necessary to complete that step of the computation. It points to the parent stack frame to which it can send its result.

2.2 ITE Algorithm

The algorithm is based on Shannon's decomposition theorem which states that $F = v \cdot F_v + \overline{v} \cdot F_{\overline{v}}$

Where F_v and $F_{\overline{v}}$ are F evaluated at $v = 1$ and $v = 0$ respectively. Let $F = (w, T, E)$ and assume $v \le w$. Finding the cofactors of F with respect to v is trivial: $F_v = F$ (if $v < w$) or T (if $v = w$), and $F_{\overline{v}} = F$ (if $v < w$) or E (if $v = w$).

The following recursive formulation is the key to computing $ite(F,G,H)$ for functions F, G, H represented in ROBDD form. Let $Z = ite(F,G,H)$ and let v be the top variable of F, G, H. Then,

$Z = v \cdot Z_v + \overline{v} \cdot Z_{\overline{v}} = v \cdot \left(F \cdot G + \overline{F} \cdot H\right)_v + \overline{v} \cdot \left(F \cdot G + \overline{F} \cdot H\right)_{\overline{v}} =$

$$= v \cdot \left(F_v \cdot G_v + \overline{F}_v \cdot H_v \right) + \overline{v} \cdot \left(F_{\overline{v}} \cdot G_{\overline{v}} + \overline{F}_{\overline{v}} \cdot H_{\overline{v}} \right) = ite\left(v, ite\left(F_v, G_v, H_v \right), ite\left(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}} \right) \right) =$$

$$= \left(v, ite\left(F_v, G_v, H_v \right), ite\left(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}} \right) \right)$$

The terminal cases for this recursion are: $ite(1,F,G) = ite(0,G,H) = ite(F,1,0) = F$.

With the assumption of constant lookup and insert in the tables, all operations in *ite* take constant time. Observe that *ite* can be called at most once for each combination of nodes in F, G, H, i.e., $O\left(|F| \cdot |G| \cdot |H|\right)$ times. So the time complexity is $O\left(|F| \cdot |G| \cdot |H|\right)$ [3]. In practice, the typical performance is closer to the size of the resulting function.

Finally we present the *ite* algorithm:

```
Fill terminal cases in the computed table

ite(F,G,H)
{
while (work queue for this thread is not empty)
  {
    Extract from queue F, G, H
    Find a stack frame S corresponding to F, G, H
    If (terminal case)
      Write the result to the parent stack frame;
    else if (computed-table has entry {F,G,H})
      Write the result to the parent stack frame
    else
    {
      Let v be the top variable of {F,G,H}
      Create stack frames H for (F_v,G_v,H_v) and L for
      (F_v̄,G_v̄,H_v̄). Point H and L to their parent stack
      frame S
      if (S has computed results for low- and high-nodes)
      {
        T = high-node result from S
        E = low-node result from S
        if (T equals E)
          Write T as the result to the parent stack
          frame;
        R = find_or_add_unique_table (v,T,E);
        Insert_computed_table ({F,G,H}, R);
        Delete (F,G,H) from the uncomputed table
        Write R as the result to the parent stack frame
        and send notification to other pending stack
        frames (in other words, add all the pending stack
        frames to the work queue)
      }
      else
```

```
      {
         if (S does not have a computed result for
         the high-node)
         {
            if (uncomputed table does not have an entry
            (F_v,G_v,H_v) )
            {
              Choose thread with the shortest work
              queue. Add (F_v,G_v,H_v) to this work queue
            }
            else
            {
               Add (F,G,H) to the pending list of
               (F_v,G_v,H_v)
            }
         }
         if (S does not have a computed result for
         the low-node)
         {
            if (uncomputed table does not have an entry
            (F_v̄,G_v̄,H_v̄) )
            {
              Choose thread with the shortest work
              queue. Add (F_v̄,G_v̄,H_v̄) to this work queue
            }
            else
            {
               Add (F,G,H) to the pending list of
               (F_v̄,G_v̄,H_v̄)
            }
         }
      }
    }
  }
}
```

2.3 Garbage Collection Process

Computers have limited memory and they run out of it quickly. To avoid this problem, nodes that are unnecessary should be removed. We named the procedure which performs this function garbage collection (GC).

Even if memory is not completely full, GC is very useful. A smaller number of nodes means faster operations on BDDs. However, some extra time is spent on GC. For an effective GC additional information about each node is needed. Unfortunately, this increases the usage of memory. Therefore, a more frequent GC, which is faster, is preferable.

GC deletes all nodes which are not part of any formula. If the user wants, a complete formula can be removed, too. Internal nodes of other formulae must not be deleted by this operation.

Note that all records in the computed-table which contain bad nodes must be removed before deleting bad nodes from the unique-table.

Each node has a count of the number of other nodes and the number of user formulae that refer to it. A node with a reference count of 0 is called dead node.

When a formula is deleted, the reference count of the corresponding top node is decremented by 1. If the new reference count is 0, then the reference counts of successor nodes are recursively decremented. If any of them becomes a dead node, recursion continues there.

It can happen that an existing node which is already dead should be included into the formula. In this case the reference counts of this node and of all dead descendants are incremented by 1. In recursion it should be considered that all successors of a non-dead node are non-dead nodes.

The procedure for GC removes all dead nodes.

4 Experimental Results

Many problems in Electronic CAD (especially in the synthesis and test area) require the computation of the maximum clique in a graph. The BDD approach to this problem is based on using characteristic functions to represent sets of subsets.

Given a set of n elements Q, and being $B=\{0,1\}$, any subset $S \subseteq Q$ can be easily mapped to a point in B^n: each element $q_i \in Q$ is represented by a Boolean variable x_i getting values in B; each combination of values for the variables $x_1,\ldots,x_n$ identifies the subset of Q composed by just the elements corresponding to the variables set to 1. Any set T of subsets Q can now be represented by a characteristic function $f:\mathrm{B}^n \to \mathrm{B}$ which returns the value 1 iff its inputs $x_1,\ldots,x_n$ represent an element of T.

A Boolean expression can be defined for the characteristic function f_{CCC} of the set of all the Completely Connected Components (CCC) of a graph V. Denoting by n the number of vertices in V, and by $x_1,\ldots,x_n$ the Boolean variables associated with the vertices, the function $f_{CCC}:\mathrm{B}^n \to \mathrm{B}$ is defined as follows:

$$f_{CCC}(x_1,\ldots,x_n) = \prod \overline{(x_i \cdot x_j)},$$

where Π denotes the logical and operator extended to all the missing edges e_{ij} between any two nodes v_i, v_j in the graph. The function returns the value 1 iff the input is a CCC.

Once f_{CCC} has been computed, determining the maximum clique in the graph means finding the maximum-cost satisfying assignment for f_{CCC}, where the cost of each assignment is the number of variables set to 1 in it. Denoting by s the number of

nodes of the BDD representing f_{CCC}, this operation can be done with complexity $O(s)$. As s is always much smaller than the worst case 2^n, the total cost of finding the MC with this method is the one for building the function f_{CCC}.

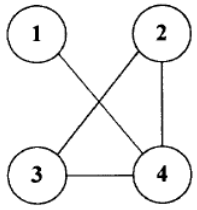

Fig. 6. A graph example

As an example, let us consider the graph of Fig. 6. The function f_{CCC} is:

$f_{CCC} = \overline{x_1 x_2} \cdot \overline{x_1 x_3} = \overline{x}_1 + \overline{x}_1 \cdot \overline{x}_3 + \overline{x}_1 \cdot \overline{x}_2 + + \overline{x}_2 \cdot \overline{x}_3$

The maximum-cost satisfying assignment for such a function is $x_2 \cdot x_3 \cdot x_4$; the MC in the graph is thus composed by the vertices v_2, v_3 and v_4.

Monoprocessor system had Intel Pentium 4 2.4 GHz processor; 2-core system had Intel Core 2 Duo E6600 2.4 GHz processor and 4-core system had Intel Core 2 Quad Q6600 2.4 GHz processor. All the systems were equipped with 2 GB RAM. The experimental results are shown in Fig. 7 and Fig. 8.

Graph size	Monoprocessor system (s)	2-core system (s)	4-core system (s)
10	2	1	<1
50	58	37	25
80	5401	3270	2026
100	7430	4600	2801
150	8501	5304	3257
200	9275	5819	3516
250	12285	8537	5212
400	25341	15509	9410

Fig. 7. Experimental results

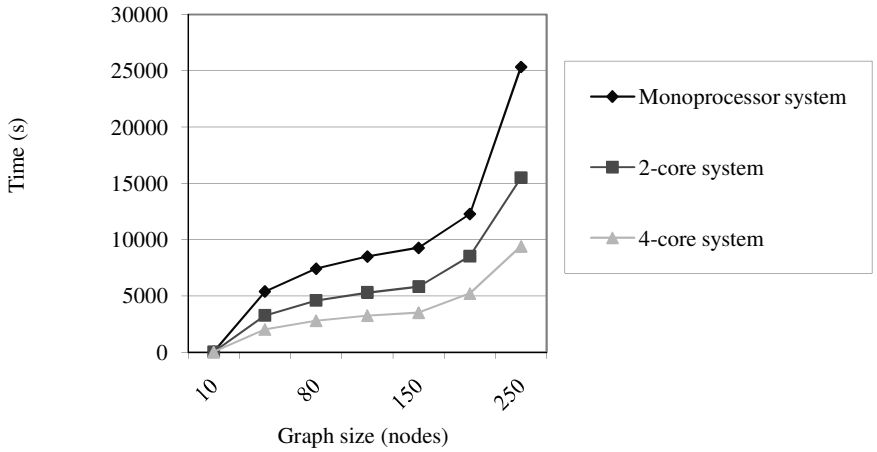

Fig. 8. Graphical representation of the experimental results

Summary

Efficient solutions on as to how to represent Boolean functions with ROBDD are shown. It is shown in detail how to parallelize the main algorithms for manipulating ROBDDs. Our representation uses If-Then-Else operator, computed and uncomputed tables, unique table and work queue. Data structures and algorithms are described in detail. The used algorithms are based on Shannon's decomposition theorem.

A practical result of this research is a ROBDD programming package written in C. The package can be used as a foundation for various tasks: proof of correctness of combinatorial circuits, system verification, protocol validation, etc.

Summing up, this BDD package is an efficient and portable BDD package that demonstrates speed-up over optimized sequential code. It presents an excellent platform for further research.

BIBLIOGRAPHY

1. S. Kimura, Residue BDD and Its Application to the Verification of Arithmetic Circuits, Proc. 32th ACM/IEEE Design Automation Conference, pp. 542-545, 1995
2. S.-I. Minato, Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems, Proc. 30th ACM/IEEE Design Automation Conference, pp. 272-277, 1993
3. K. S. Brace, R. L. Rudell, R. E. Bryant, Efficient Implementation of a BDD Package, 27^{th} ACM/IEEE Design Automation Conference, 1990, pp. 40-45
4. S. Minato, N. Ishuira and S.Yajima, Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation, In

Proceedings of 27th ACM/IEEE Design Automation Conference, pp. 52-57, June 1990
5. Y. Parasuram, E. Stabler and Shiu-Kai Chin, Parallel Implementation of BDD Algorithms Using a Distributed Shared Memory, In Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences Vol I: Architecture, pp. 16-25, January 1994
6. H. Ochi, S. Yajima, N. Ishiura, A Vector Algorithm for Manipulating Boolean Functions Based on Shared Binary Decision Diagrams, Supercomputer, Vol. 8, No. 6, November 1991
7. G. Cabodi, S. Gai, M. Rebaudengo, M. Sonza Reorda, A Data-Parallel Approach to Boolean Function Manipulation using BDDs, IEEE/Euromicro Conf. on Massively Parallel Comp. System, 1994, pp. 163-175
8. S. Kimura, T. Igaki, H. Haneda, Parallel Binary Decision Diagram Manipulation, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science, Vol. E75-A, No. 10, pp. 1255-62, October 1992
9. P. Arunnacham, C. Chase, D. Moundanos, Distributed Binary Decision Diagrams fro Verification of Large Circuits, IEEE Int. Conf. on Comp. Design, 1996, pp. 365-370
10. G. Cabodi, S. Gai, M. Rebaudengo, M. Sonza Reorda, A Data-Parallel Approach to Boolean Function Manipulation using BDDs, IEEE/Euromicro Conf. on Massively Parallel Comp. System, 1994, pp. 163-175
11. S. Kimura, E.M. Clarke, A Parallel Algorithm for Constructing Binary Decision Diagrams, IEEE Int. Conf. on Comp. Design, 1990, pp. 220-223
12. S. Kimura, Residue BDD and Its Application to the Verification of Arithmetic Circuits, Proc. 32th ACM/IEEE Design Automation Conference, pp. 542-545, 1995
13. S.-I. Minato, Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems, Proc. 30th ACM/IEEE Design Automation Conference, pp. 272-277, 1993
14. R. Rudell, Dynamic Variable Ordering for Ordered Binary Decision Diagrams, In Proceedings of the IEEE International Conference on Computer-Aided Design, pp. 42-47, Santa Clara, CA, November 1993
15. Tsutomu Sasao, Munehiro Matsuura, Methods and representations for logic synthesis: BDD representation for incompletely specified multiple-output logic functions and its applications to functional decomposition, Proc. 42nd annual conference on Design automation DAC '05, pp. 373 – 378, 2005
16. Lei Cheng, Deming Chen, Martin D. F. Wong, FPGA tools and methodologies: DDBDD: delay-driven BDD synthesis for FPGAs, Proc. 44th annual conference on Design automation DAC '07, pp. 910–915, 2007
17. S. Minato, S. Ishihara, Streaming BDD manipulation for large-scale combinatorial problems, Proceedings of the conference on Design, automation and test in Europe DATE '01, pp. 702–707, 2001
18. Ziv Nevo, Monica Farkash, Distributed dynamic BDD reordering, Proc. 43rd annual conference on Design automation, pp. 223–228, 2006
19. Riccardo Forth, Paul Molitor, An efficient heuristic for state encoding minimizing the BDD representations of the transistion relations of finite state

machines, Proceedings of the 2000 conference on Asia South Pacific design automation, pp. 61–66, 2000
20. Rüdiger Ebendt, Rolf Drechsler, Lower bounds for dynamic BDD reordering, Proceedings of the 2005 conference on Asia South Pacific design automation, pp. 579-582, 2005
21. Ralf Wimmer, Marc Herbstritt, Bernd Becker, Optimization techniques for BDD-based bisimulation computation, Proc. 17th great lakes symposium on Great lakes symposium on VLSI, pp. 405-410, 2007