A Parallel Approach to Symbolic Traversal based on Set Partitioning

G. Cabodi[1] P. Camurati[2] A. Lioy[1]
M. Poncino[1] S. Quer[1]

[1] Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy,
e-mail {cabodi,a.lioy,poncino,quer}@polito.it
[2] Dipartimento di Matematica e Informatica, Università di Udine, Udine, Italy,
e-main camurati@dimi.uniud.it

Abstract

Binary Decision Diagrams are the state-of-the-art technique for many synthesis, verification and testing problems in CAD for VLSI. Many efforts have been spent to optimize this representation but in many complex applications they still require large amounts of (working) memory and of CPU time. Virtual memory is not a good solution to this problem because, if the working set size for a program is large and memory accesses are random, an extremely large number of page faults significantly modifies software performance.

Within this frame one of the most important and general operations is symbolic generation of the states reachable by a Finite State Machine. This is the starting point for many verification tasks.

This paper presents a new mechanism for alleviating the two previous drawbacks by: 1) Decomposing the global problem in sub-problems by a "*divide-and-conquer*" approach that reduces memory requirements and 2) Using a parallel implementation of the algorithms, as parallel architectures represent a natural environment to overcome these limitations.

Partitioning techniques and granularity of parallel tasks are discussed as a major issue to obtain a viable and efficient solution. Experimental results show the feasibility of the approach.

Keywords

BDDs, FSMs, Reachability Analysis, Symbolic Breadth-First Traversals, Parallel Computing

1 INTRODUCTION

Efficient techniques for Boolean function manipulation are a key point in many areas, such as digital logic design and testing, artificial intelligence, combinatorics. The efficiency of logic function manipulation depends on the data structure used for representing Boolean functions. The state-of-the-art approach to the problem is based on ***Binary Decision Diagrams*** (BDDs) [4]. A number of results have been published on this topic in recent years and BDDs are now used in several academic and commercial tools for digital circuit design, verification and testing. Most of these algorithms involve the exploration of state spaces of ***Finite State Machines*** (FSMs)

FSMs are a popular model for ***control-dominated*** ASICs and are identified by their input/output alphabets, initial state sets and next state and output functions. The ***reachable state space*** of a FSM is identified by a ***forward traversal.*** Intuitively, a state is reachable if a sequence of inputs causes the FSM to evolve from any initial state to that state. The next state function determines evolution along time. The next states are the image, for all inputs, of the current state set according to the next state function. The process terminates as soon as a fixed-point is reached, i.e., no newly reached states are found.

Although quite successful, symbolic methods cannot complete the reachability analysis of large FSMs, because:

- BDDs require too much memory: peak BDD size during image computation and the size of BDDs representing reached states are critical. The conventional BDD manipulation packages work on a "depth–first" basis and result in random access to memory, leading to a large number of page faults. Since a page access is of the order of tens of milliseconds, a large number of page faults increases the amount of time necessary to deal with the problem even though the time spent by the processor doing useful work is quite small.
- Manipulating BDDs is computationally expensive.

Several approaches have been proposed to solve these problems. Dynamic variable reordering [19] overcome many limitations but they can slow down an application by more than an order of magnitude and they are still insufficient for very complex manipulations. Ochi *et al.* [16] proposed a breadth–first approach to regularize memory access, which leads to fewer page faults. Following this trend, Ashar *et al.* [2] presented an improved breadth–first algorithm, which enabled manipulation of BDDs with up to 100 million nodes. Other authors concentrate on different formalisms, e.g., Indexed BDDs (IBDDs) [11], that allow polynomial representations of functions that would otherwise require an exponential space. Attempts have been made also at using special computing hardware configuration: in [12] Kimura *et al.* adopt a shared memory multiprocessors and in [15] authors use a vector processor. Finally a network of workstations is used in [18] to design a distributed BDD data structure and in [1] to allow the use of distributed main memory.

In our approach we do not modify the software package (the BDD package) and we do not rely on special hardware platforms, rather we concentrate on a higher level of abstraction.

Therefore our goal is to:

- Present a *"divide-and-conquer"* approach to the problem; in that direction we follow the method presented in [8], where we:
    - Decompose sets of states when their BDD representation becomes too large or when image computation becomes too expensive.
    - Carry out expensive computations in a decomposed form; this allows us to deal with just one subset at a time, decreasing the peak number of BDD nodes.
- Resort to a parallel algorithm oriented to general purpose Multiple Instruction Multiple Data (MIMD) architectures with distributed main memory but with a shared disk.

Despite the possibility of repeated computations, decomposition decreases overall complexity, with remarkable benefits in terms of running CPU time [8]. The intrinsic characteristic of the method and the size and complexity of functions make the use of parallel computers in this area extremely attractive.

For our experiments we use sets of benchmarks universally known in the field of CAD (ISCAS'89 and ISCAS'89-addendum). These are large synchronous circuits whose characteristics are reported in the literature. Experimental results show that the overall method is particularly effective on large circuits.

The remainder of the paper is organized as follows. Section 2 summarizes some useful concepts on BDDs, Boolean functions and sets, image computations, FSMs and symbolic traversals. Section 3 describes the underline approach to decompose and manipulate BDDs, as described in [8]. Section 4 describes the parallel implementation. Section 5 shows experimental results. Section 6 closes the paper with a brief summary and future work.

2 PRELIMINARIES

Binary Decision Diagrams and Elementary Operations

Binary Decision Diagrams (BDDs) are a canonical representation of Boolean functions $f(x_1, x_2, x_3, \ldots, x_n)$ in the form of directed acyclic graphs. The reader should refer to [4] and [5] for a tutorial introduction to BDDs.

We use the standard operations of Boolean algebra: $+$ for OR, $\cdot$ for AND, $\oplus$ for EXCLUSIVE-OR and an overline for NOT. We also use summation $\sum$ and product $\prod$ notation referring to Boolean sums (OR) and products (AND).

The function resulting from assigning variable x_i to a constant value k (either 0 or 1) is a *restriction* or a *cofactor* of $f(x_1, \ldots, x_i, \ldots, x_n)$ with respect to x_i [3]:

$$f(x_1, \ldots, x_i, \ldots, x_n)_{x_i=0} = f(x_1, \ldots, 0, \ldots, x_n) = f_{\overline{x_i}}$$
$$f(x_1, \ldots, x_i, \ldots, x_n)_{x_i=1} = f(x_1, \ldots, 1, \ldots, x_n) = f_{x_i}$$

Given the two cofactors $f_{\overline{x_i}}$, i.e., the *negative* cofactor, and f_{x_i}, i.e., the *positive* one, f is expressed as:

$$f(x_1, \ldots, x_i, \ldots, x_n) = \overline{x_i} \cdot f_{\overline{x_i}} + x_i \cdot f_{x_i}$$

This identity is commonly referred to as *Shannon's expansion* of f with respect to x_i, although it was originally recognized by Boole.

The two following identities define *existential quantification* (or *smoothing*) and *universal quantification* (or *consensus*):

$$\begin{aligned} \exists_{x_i} f &= f_{\overline{x_i}} + f_{x_i} \\ \forall_{x_i} f &= f_{\overline{x_i}} \cdot f_{x_i} \end{aligned}$$

Many operations on Boolean functions can be implemented by simple graph algorithms that work recursively on their BDD representation in a conventional *"depth-first"* fashion. For example if f and g are Boolean functions represented by BDDs, if x_i is one of their variables and if op is a generic binary operator, we express f op g as:

$$f \text{ op } g = \overline{x_i} \cdot (f_{\overline{x_i}} \text{ op } g_{\overline{x_i}}) + x_i \cdot (f_{x_i} \text{ op } g_{x_i})$$

This is possible because algebraic operations and expansion *commute*.

Tasks are then usually expressed as a sequence of steps, each involving an operation on one or more BDDs. Examples include: Determining whether two functions are equivalent, computing their logical conjunction, disjunction or negation, determining the size of the on–set for a function, etc. These algorithms obey to the *closure* property: Given BDDs as arguments with a certain ordering, the result is a BDD obeying the same ordering. Some of these algorithms have time and space complexity polynomial in the size of their operand graphs.

2.1 Supports, Sets and Characteristic Functions

Given a vector of Boolean variables $x = (x_1, x_2, \ldots, x_n)$ and a Boolean function $f(x)$, the *"true support"* of f is the set of x_i variables on which f depends, i.e., such that the positive and negative cofactors of f with respect to x_i differ:

$$supp(f) = \{x_i \text{ such that } f_{x_i} \neq f_{\overline{x_i}}\}$$

Let A be a subset of $\mathcal{B}^n$. The *characteristic function* of A is the function $\chi_A : \mathcal{B}^n \to \mathcal{B}$ defined by:

$$\chi_A(a) = \begin{cases} 1 & if \quad a \in A \\ 0 & if \quad a \notin A \end{cases}$$

Operations on sets can be efficiently implemented by Boolean operations on their characteristic functions. For example, if A and B are two subsets of S and χ_A and χ_B are their characteristic functions, we write:

$$\begin{aligned} \chi_{A\cup B} &= \chi_A + \chi_B \\ \chi_{A\cap B} &= \chi_A \cdot \chi_B \\ \chi_{A-B} &= \chi_A \cdot \overline{\chi_B} \end{aligned}$$

With abuse of notation, in the rest of this paper we make no distinction between the BDD representing a set of states, the characteristic function of the set and the set itself.

2.2 Image of a Set

In many application of BDDs, sets are constructed and manipulated using characteristic functions without ever enumerating explicitly their elements.

Let $f : \mathcal{B}^i \rightarrow \mathcal{B}^j$ be a Boolean function and $C \subseteq \mathcal{B}^i$ a subset of its domain. The *image* of C according to f is:

$$\textsc{Img}(f,\ C) = \{y \in \mathcal{B}^j \text{ such that } \exists x \in C \ \wedge \ y = f(x)\}$$

Subset C is often called "*constraint*". Whenever $C = \mathcal{B}^i$, the image is often called "*range*".

2.3 The Model

A *Finite State Machine* (FSM) is an abstract model describing the behavior of a sequential circuit. A completely specified FSM M is a 6-tuple

$$M = (I,\ O,\ S,\ \delta,\ \lambda,\ S_0)$$

where I is the input alphabet, O is the output alphabet, S is the state space, $\delta : S \times I \rightarrow S$ is the next state function, $\lambda : S \times I \rightarrow O$ is the output function and $S_0 \subseteq S$ is the initial state set.

In the rest of the paper, we denote with $s = (s_1,\ s_2,\ \ldots,\ s_n)$ present state variables, with $x = (x_1,\ x_2,\ \ldots,\ x_m)$ primary inputs I of the FSM and with $y = (y_1,\ y_2,\ \ldots,\ y_n)$ next state variables. We do not consider output functions λ; we introduced them just for sake of completeness.

2.4 The Transition Relation

Let M be a FSM. The *Transition Relation* T associated to M is defined as:

$$T(x,\ s,\ y) = \prod_{i=1}^{n}(y_i \equiv \delta_i(x, s)) = \prod_{i=1}^{n} t_i(x,\ s,\ y_i)$$

A transition relation is often called *partitioned* if conjunctions are not performed once and forall, but t_is are kept separate until an image computation [6], [7]. The *image* of a set $C(s)$ is defined as:

$$\textsc{Img}(\delta,\ C(s)) = \exists_{x,\ s}\ (T(x,\ s,\ y) \cdot C(s)) = \exists_{x,\ s}\ (\textstyle\prod_{i=1}^{n} t_i(x,\ s,\ y_i) \cdot C(s))$$

Images are computed with a partitioned transition relation by resorting to early quantification during conjunction steps. Suppose that some inputs and current state variables appear just in the first i partitions. Let E_i be sets of such variables. Early quantification eliminates variables belonging to the E_i sets before conjoining the t_{i+1} term:

$$\begin{aligned}\textsc{Img}(\delta,\ C(s)) &= \exists_{x,\ s}(\textstyle\prod_{i=1}^{n} t_i(x,\ s,\ y) \cdot C(s)) \\ &= \exists_{(x,\ s)\in E_n}(t_n \cdot (\exists_{(x,\ s)\in E_{n-1}}(t_{n-1} \cdot \dots \cdot \exists_{(x,\ s)\in E_1}(t_1 \cdot C(s)))))\end{aligned} \tag{1}$$

The atomic operation in image computation is conjunction–quantification, i.e., and–exists operations.

Several heuristics have been presented to sort the partitions. Further improvements are obtained through clustering, that may be used to decrease the number of partitions by performing some products once forall before image computations [17].

2.5 Symbolic Traversal

A *Symbolic Traversal* is a breadth-first search that returns at each iteration the set of states reached from the current state set.

Figure 1 shows the pseudo-code.

```
(1)  TRAVERSAL ($\delta$, $S_0$)
(2)  {
(3)  Reached = From = New = $S_0$;
(4)  while (New $\neq \emptyset$)
(5)      {
(6)      To = IMG ($\delta$, From);
(7)      New = To $\cdot$ $\overline{\text{Reached}}$;
(8)      Reached = Reached + New;
(9)      From = BEST_BDD (New, Reached);
(10)     }
(11) return (Reached);
(12) }
```

Fig. 1. Forward Traversal.

From is the current set of states and To is the set directly reached from From. This is accomplished by means of a symbolic image computation IMG(δ, From), line 6. Set **New** contains the To states that have not yet been visited. Reached states are accumulated in **Reached**. Initially From is set to S_0 and then it is selected by choosing a suitable BDD that represents all newly reached states and possibly some of the already visited ones, as in [9] (procedure BEST_BDD).

The termination condition is to reach a least fixed-point. This condition is equivalent to testing the emptiness of **New** at each step, line 4.

The number of iterations of this algorithm gives the ***sequential depth*** of the machine.

Large amount of states, greater than 10^{120}, have been visited efficiently by means of symbolic traversals.

3 DECOMPOSED TRAVERSAL

Proceeding from one step to the next one in symbolic traversal, all sets, in particular From and **Reached**, become larger and much more complex to represent by means of BDDs. As a consequence, symbolic traversal experiences two bottlenecks:

- A monolithic BDD representing the sets may be too large to fit into main memory.
- It may be impossible to perform an image computation (function IMG), because of the size of the BDDs involved in intermediate computations.

As mass memory is inexpensive with respect to main memory, very often virtual memory (BDD nodes are automatically swapped to the hard disk by the operating system) is considered a good solution to problems where memory requirements are a key issue. Unfortunately conventional "depth-first" algorithms cause random access of memory. As there have not been significant improvements in the speed of swapping from main memory to disk, in the last few years, this implies that if the working set of memory pages for a program is large, the time required to deal with page faults significantly impacts on the performance of the system.

A lot of approaches essentially modify the BDD package by changing the representation (e.g., *Zero-Suppressed BDDs*, *MTBDDs*, etc.) or the access method (e.g., ***breadth-first*** manipulation, etc.). As all these approaches are quite expensive in term of rewriting code and modifying strategies and have both advantages and disadvantages, we prefer to work at a higher level, leaving the BDD package unchanged and we concentrate on the applications, i.e., reachability analysis.

Our approach [8] consists in decomposing state sets, using a "***divide-and-conquer***" strategy, when, during traversal, they become too large to be represented as a monolithic BDD or when image computation becomes too expensive. Image computation is carried out on decomposed sets. Useless BDDs are then stored in secondary memory and loaded only when needed.

Using a certain threshold we split the problem in sub-problems whose complexity is smaller; splitting can occur between:

- Image computations.
- Conjunction–abstraction operations within image computations.

In the first case, we decompose the current state set $C(s)$ as $v \cdot C_v(s) + \overline{v} \cdot C_{\overline{v}}(s)$ using the Boole's expansion. Its image, according to δ, is then equivalent to the union of the images of $v \cdot C_v(s)$ and $\overline{v} \cdot C_{\overline{v}}(s)$:

$$\begin{aligned}\textsc{Img}(\delta,\ C(s)) &= \textsc{Img}(\delta,\ (v \cdot C_v(s) + \overline{v} \cdot C_{\overline{v}}(s))) \\ &= \textsc{Img}(\delta_v,\ (v \cdot C_v(s))) + \textsc{Img}(\delta_{\overline{v}},\ (\overline{v} \cdot C_{\overline{v}}(s)))\end{aligned}$$

In the second case the approach is similar. Computing the image using the standard partitioned transition relation, see Equation 1, we apply the same decomposition technique to the i–th conjunction step, $\exists_{(x,s) \in E_i}(t_i \cdot \ldots)$.

In general, the first method is better when overall image computation is too expensive whereas the second one optimizes the first one when only a few steps of image computation are particularly expensive. In both cases we can re-compose the resulting set after a decomposed operation, or we can carry on operations on the partitioned form.

Partitioning is a good solution in both cases, because:

- The advantage of working on decomposed sets stems from lowering overall complexity in terms of memory and execution time. In fact recursive splitting is a very common practice with BDDs, as it characterizes almost all BDD operators, but it normally follows a fixed variable selection scheme: Variable ordering. Partitioning the operands by means of splitting variables is equivalent to pushing them onto the top of variable ordering and it is independent from variable ordering as it possibly chooses different splitting variables when recurring in different set partitions.
- Partitions not directly involved in computations can be downloaded to mass storage and this:
  - Lowers the amount of working memory required.
  - Avoids repeated page faults.
  - Allows easily an implementation on parallel machines.

  The method used to download large BDDs relies on binary file manipulation. As a BDD node typically takes 16 bytes of memory for machines with 32 bit pointers, downloaded BDDs take approximately 1/5 of the space that they take on main memory[3].

[3] The global amount of main memory is influenced also by a few tables, like the computed table, that can consume large amount of memory and whose size is typically related to the amount of BDD nodes used.

Figure 2 shows the pseudo-code for decomposed traversal. It is derived from the standard traversal of Figure 1.

```
(1)  PARTITIONED_TRAVERSAL ($\delta$, $S_0$, $th$)
(2)  {
(3)  $\text{Reached}_p = \text{From}_p = \text{New}_p = \{ S_0 \}$;
(4)  while ($\text{New}_p \neq \emptyset$)
(5)      {
(6)      $\text{From}_p$ = SET_PARTITION ($\text{New}_p$, $th$);
(7)      $\text{To}_p = \emptyset$;
(8)      foreach $f \in \text{From}_p$
(9)          $\text{To}_p = \{ \text{To}_p$, IMG ($\delta$, $f$) $\}$;
(10)     $\text{New}_p$ = SET_DIFF ($\text{To}_p$, $\text{Reached}_p$, $th$);
(11)     $\text{Reached}_p$ = SET_UNION ($\text{New}_p$, $\text{Reached}_p$, $th$);
(12)     }
(13) return ($\text{Reached}_p$);
(14) }
```

Fig. 2. Partitioned Forward Traversal.

Reached_p, From_p and New_p represent sets **Reached**, **From** and **New** in monolithic or partitioned form. They are initially set to S_0. At each step we generate set From_p, line 6, in the right decomposed representation according to the size of the BDD representation of set New_p and to parameter th. Parameter th controls size and number of state set partitions as well as the complexity of the image computation procedure. Its value is usually chosen by manually tuning the traversal procedure, keeping into account the complexity of the problem and the power of the host machine.

At each step, set To_p is initialized to the empty set, line 7. Instead of computing a single image, line 6 in Figure 1, we call the image computation procedure for each subset f of From_p, line 8. This is done on line 9, IMG (δ, f), and new images are added to To_p as a new set. This allows the image computation procedure to work on just a subset at a time, decreasing peak BDD size. Internally the image function can decompose sets as previously introduced but this doesn't appear in the pseudo-code.

After image computation we call functions SET_DIFF and SET_UNION. These functions are relatively straightforward and compute sets New_p and Reached_p for the next iteration. In particular they evaluate fixed point and new decomposed representations for these sets.

More information on the overall methodology can be found in [8].

4 THE PARALLEL APPROACH

Previous works on BDDs and parallel computing have mainly focused on "pure" functions representation or combinational verification. On the other hand we concentrate on sequential verification.

A wide range of solutions are possible, basically depending on the kind of target parallel environment. Before going into the details of the implemented algorithm, we will focus on the two main possible choices: ***Domain*** and ***co-domain partitioning***. We show how the first possibility, i.e., domain partitioning, is a natural extension to a parallel platform of the approach introduced in Section 3. Then we focus on our approach.

4.1 Domain and co-domain partitioning

As discussed in Section 3 symbolic evaluation of images is the main problem of symbolic traversal. Coudert *et al.* introduce in [10] ***domain*** and ***co-domain partitioning*** to reduce complexity of image computations.

Domain and co-domain splitting perform a sort of ***functional partitioning*** while ***data partitioning*** is introduced to fully exploit the whole memory available in a parallel machine.

Let us introduce again the IMG operator in the form:

$$\mathsf{To} = \textsc{Img}(\delta, \mathsf{From})$$

in which From and To represent the source (domain) and target (co-domain) of the image computation process, respectively. In ***domain partitioning*** the domain set From is decomposed in d disjoint subsets

$$From = \bigcup_{i=1}^{d} From_i$$

and the image To can be expressed as the union of distinct image computations:

$$To = \bigcup_{i=1}^{d} \textsc{Img}(\delta, \mathsf{From}_i)$$

In ***co-domain partitioning*** a single image evaluation task can be divided in sub-tasks through a proper co-domain space partitioning. Let us divide also the image space Y in c disjoint subsets

$$Y = \bigcup_{j=1}^{c} Y_j$$

and create a new procedure IMG* that works on a restricted image space:

$$To_j = \textsc{Img}^*(\delta, \mathsf{From}, Y_j)$$

Then co-domain partitioning can be expressed as:

$$To = \bigcup_{j=1}^{c} To_j = \bigcup_{j=1}^{c} \textsc{Img}^*(\delta, \mathsf{From}, Y_j)$$

4.2 Proposed approach

Neither domain nor co-domain partitioning is a perfect partitioning: In the former case disjoint domain sub-sets can produce overlapping images, in the latter repeated evaluations can occur in different image sub-spaces. Experimental results in [10] show that there is no a priori choice between the two approaches, and that depending on the test case one can perform much better than the other.

The strategy proposed in Section 3 is a particular implementation of domain partitioning and it is well suited to a parallel implementation. In fact "global" operations, like splitting sets, fixed point evaluation and so on, are far less expensive, both in terms of CPU time and of BDD nodes, than pure image computation. On the other hand each single image computation is splitted in several steps as images are computed on decomposed representations. In lines 8 and 9 of Figure 2, the image computation procedure work on just a subset at a time. On the monoprocessor implementation this decreases the peak BDD size and on the multiprocessor one allows the parallel implementation, as different image computations can be easily computed on different CPUs. Our target is to keep sub-problems large enough and well balanced to make computing times overcome communications and idle times due to unbalancing. All these factors can be controlloed tuning the value of the threshold th.

Domain sets are recursively splitted in sub-sets when complexity of their BDD representation exceeds a given threshold. For every subset an image computation is performed. The process described in Figure 2 can be then divided in three separate phases:

1. Domain set partitioning (line 6).
2. Image computation (lines 8 and 9).
3. Set union and closure computation (lines 10 and 11).

As soon as an image task is splitted into sub-tasks they are executed in parallel. The splitting procedure can follow two paradigms:

- *Centrally Controlled Approach*: In this kind of approach phase 1, function SET_PARTITION, and phase 3, functions SET_DIFF and SET_UNION, are executed by a unique main process that splits completely the BDD representing the domain; then the different images (function IMG) are computed in parallel.
- *Fully Distributed Approach*: In this approach all three phases, partition, images computation and recombination, are distributed, i.e., executed in parallel.

The second strategy is supposed to be more efficient but it also has a greater overhead. We will show in the experimental result section that image computation is by far the most expensive phase. This implies that also the first approach can be quite efficient.

Besides generating proper sub-tasks of the image evaluation problem, a second important choice has been done with data structures: State sets are stored on a common disk and are accessible to all processes. This implies that the overall process is expensive from a computational point of view, but not from a communication point of view.

Figure 3 reports the pseudo–code of the *Centrally Controlled Approach.*

```
 (1)  PARALLEL_PARTITIONED_TRAVERSAL ($\delta$, $S_0$, $th$)
 (2)  {
 (3)  $Reached_p$ = $From_p$ = $New_p$ = { $S_0$ };
 (4)  while ($New_p \neq \emptyset$)
 (5)      {
 (7)      $From_p$ = SET_PARTITION ($New_p$, $th$);
 (6)      $To_p = \emptyset$;
 (8)      foreach $f \in From_p$
 (9)          if $\exists$(idle processor) {
(10)              id = fork();
(11)              if (id == 0)
(12)                  exec ($To_p$ = ($To_p$, IMG ($\delta$, $f$))); }
(13)          else {
(14)              wait (&status);
(15)              if ( status > ERROR_CODE )
(16)                  exit (1); }
(17)      foreach child still running
(18)          {
(19)          wait (&status);
(20)          if ( status > ERROR_CODE )
(21)              exit (1);
(22)          }
(23)      $New_p$ = SET_DIFF ($To_p$, $Reached_p$, $th$);
(24)      $Reached_p$ = SET_UNION ($New_p$, $Reached_p$, $th$);
(25)      }
(26)  return ($Reached_p$);
(27)  }
```

Fig. 3. Parallel version of the Partitioned Forward Traversal.

In particular line 9 of Figure 2 is expanded in Figure 3 into lines 9 ÷ 22.

Like in Figure 2 for each subset of $From_p$ we have to compute an image. If there is an idle processor we fork the process: The child process (id==0) computes the new image, line 12, on that processor whereas the parent loops again.

If there are no idle processes the parent waits for the termination of one of the children, line 14, testing for its exit status, lines 15 and 16. The same operation is done when all the children have been started, lines $17 \div 21$.

The pseudo code can easily be expanded to:

- Keep into account further possible decompositions performed inside function IMG, see Section 3.
- Obtain a parallel execution also during phases 1 and 3 (functions SET_PARTITION, SET_DIFF and SET_UNION).
- Get a better balance of the processor charge during various phases of traversal.

Theoretical Evaluation Let us work on a traversal problem P, and define $Size(P)$ as the global number of BDD nodes required to solve it in a mono-processor implementation. In our environment problem P is decomposed in N parallel sub-tasks P_i ($i = 1 \div N$). We express the global size of BDDs involved in this solution as:

$$Size(P_{Part}) = \sum_{i=1}^{N} Size(P_i) = \alpha_s \cdot Size(P)$$

with $\alpha_s \geq 1$. If α_s can be kept low and $Size(P_i)$ ($i = 1 \div N$) is well balanced, the distributed approach can solve problems not manageable with the mono-processor one, due to the large global memory space available on parallel machines.

We can make some similar remarks from the point of view of time speed-up. Global CPU time T_{tot} required for the parallel solution can be expressed as:

$$T_{tot}(P_{Part}) = \max_i(T_{tot}(P_i)) = \max_i(T_{CPU}(P_i) + T_{over}(P_i) + T_{idle}(P_i))$$

where T_{CPU} represents computing time, T_{over} time for spawning and state splitting (including exchange of messages), T_{idle} time lost due to not balanced tasks. T_{CPU} can also be related to $T(P)$ (time of mono-processor execution):

$$\sum_{i=1}^{N} T_{CPU}(P_i) = \alpha_t \cdot T(P)$$

where $\alpha_t \geq 1$ represents the increase in time due to partitioning. If an effectiveness ratio is introduced

$$\rho_t = \frac{\sum_{i=1}^{N} T_{CPU}(P_i)}{N \cdot T_{tot}(P_{Part})}$$

the global speed-up of our parallel solution is

$$SP = \frac{T(P)}{T_{tot}(P_{Part})} = \frac{\rho_t \cdot N}{\alpha_t}$$

where it is easy to notice that the speed-up is directly proportional to the effectiveness ratio (ρ_t) and inversely proportional to the partitioning overhead (α_t).

In the following section we will present experimental data, which quantify some of the entities introduced above and demonstrate that the advantages brought by our method increase with the size of the problems.

5 EXPERIMENTAL RESULTS

A prototype version of the parallel algorithms has been implemented in C language on two systems:

- Three 200 MHz DEC Alpha with a 256 Mbyte main memory sharing the same disk.
- Dual–node 3–way SMP with Pentium Processors at 100 MHz with a 192 Mbyte main memory for each node (i.e., a total memory of 384 Mbytes with, in average, 64 Mbytes for each CPU).

We impose a working memory limit of 200 Mbytes on the first set of platforms. The techniques are implemented on top of the Colorado University Decision Diagram (CUDD) package [20]. The code added to the CUDD package is written in C and it amounts in about 12500 lines. We refer our result to the ones presented in [8] on a mono–processor implementation to evaluate the improvement attained with the technique.

The original method presented in [8] heavily uses mass memory to store BDDs and separate different phases of the overall process. The traversal procedure could be seen as a set of sub-processes that communicate and synchronize through files exchange. In this environment the parallel implementation has been quite straightforward to realize.

Table 1 reports, for comparison purposes, data from [8] on the mono–processor implementation.

Table 1. Traversal Results on some ISCAS'89 and ISCAS'89-addendum circuits. * indicates that we use a simplified version of the original circuit

Circuit	# FF	# Level	Reached		Disk	Mem.	Time
			# Nodes	# States			
s1269	37	10	612	$1.1313 \cdot 10^{9}$	0.0	28	1424
s3271	116	17	383521	$1.3177 \cdot 10^{31}$	11.8	149	137^{h}
s3330	132	8	28748	$7.2778 \cdot 10^{17}$	9.7	107	4155
s1423	74	14	13738871	$1.7945 \cdot 10^{11}$	125.9	106	8.4^{h}
s6669	239	3	2494135*	*	22.7	97	530

Column Circuit gives the name of the circuit and # FF the number of flip-flops. # Level indicates the number of traversal iterations (partial or total, i.e., up to the fixed point), # Nodes is the number of BDD nodes of the final reachable state set and # States is its number of states. Disk indicates the maximum mass memory (in Mbyte) used to download BDDs. Due to the compression technique this amounts to about 1/5 of the space the same BDDs occupy in main memory. Mem. is the maximum amount of main memory used (in Mbyte). Time indicates the total execution time (in seconds, unless otherwise stated, or in hours).

In Table 2 we report the CPU time required by the various phases of the traversal procedure to complete traversal on the same set of circuit of Table 1.

Table 2. Time Data on the various phases of traversal

Circuit	Time			
	Set_Partition [%]	Img [%]	Set_Diff + Set_Union [%]	Total
s1269	11 [0.8]	1409 [99.0]	4 [0.2]	1424
s3271	4^h [3.0]	132^h [96.5]	0.48^h [0.5]	137^h
s3330	103 [2.5]	3990 [96.0]	62 [1.5]	4155
s1423	3320 [11.0]	24198 [80.0]	2722 [9.0]	8.4^h
s6669	32 [6.0]	478 [90.2]	20 [3.8]	530

Square brackets report percentage values to respect to the global traversal time (column Total). The traversal phase is the most expensive one, whereas decomposition and closure phases require just little percentage of total time. This table shows that it is really important to execute in parallel the traversal phase and that the Centrally Controlled Approach and the Fully Distributed Approach cannot differ too much because they differ just for phases 1 and 3.

Problem decomposition reduces complexity and memory requirements and the granularity of the processes is really important as analyze in Section 4.2. We must reduce the complexity enough to be able to run sub-problems singularly. The more partitions are produced, the easier the overall process can be divided among N processors but beyond a certain point the decomposition strategy becomes inefficient as the degree of overlapping increases to much.

Following Section 4.2, Table 3 reports data concerning N, column # Part, Size $(_i)$, Size (P_{Part}) and α_s.

We consider the final reachable state set of circuit s3271 and we split it in 2, 4 and 8 subsets. More we split more we increase overlap, α_s, but, on the other hand, we could deal better with parallelism. Finding the right balance is one of the major topic of the overall procedure: BDD dimension, BDD structure and computing limit have to be considered.

Table 4 reports data obtained on the first parallel hardware platform. In this case we could run experiments on three machines quite similar to the one used in [8]. Column # Part_{max} indicates the maximum number of partition created to compute an image computation, i.e., the maximum number of parallel processes N. Following Section 4.2, time T (P) and T_{tot} (P_{Part}) are reported. Speed-ups, SP, from a factor of 2 to a factor of 3 are obtained.

Table 3. Split Results on the final reachable state set of circuit s3271

# Part	Size ($_i$)	Size (P_{Part})	α_s
1	383521	383521	1.00
2	204896, 257876	462772	1.20
4	146945, 130854, 194413, 179924	652136	1.70
8	99800, 99697, 86433, 94622, 121819, 160763, 107330, 146274	915738	2.39

Table 4. Parallel Traversal Results on some ISCAS'89 and ISCAS'89-addendum circuits

Circuit	# $Part_{max}$	Time Mono–Processor T (P)	Time Multi–Processor T_{tot} (P_{Part})	SP
s1269	18	71424	35707	2.00
s3271	43	137^h	49^h	2.79
s3330	23	4155	1750	2.37
s1423	85	8.4^h	3.2^h	2.89
s6669	33	530	220	2.41

On the second hardware platform the major problem is the limited amount of working memory. This amount is quite low to obtain good performance on large circuits. We are currently experimenting on this set on machines.

We are also trying to experiment with a larger number of processors.

6 CONCLUSIONS AND FUTURE WORK

BDDs and symbolic techniques have undergone major improvements in the last decade in different fields of CAD and symbolic FSM state space exploration techniques represent one of the major recent results of formal verification.

The current limit of such techniques resides in the inability to represent, and compute during traversal, very large functions, relations or sets.

In this paper we propose to apply an efficient set decomposition strategy in the field of parallel computing. The original approach has been shown to be effective with large problems, involving large sets and higher computational complexity. The parallel version has been completely described. The technique uses the intrinsic characteristic of the underlined approach in a new parallel environments. Preliminary experimental results seem to show that further and relevant improvements can be obtained, both in terms of speed-ups and in supporting experiments not manageable by mono-processor architectures.

In this sense, our experimental results are not complete, due to the relatively small number of processors and the limited amount of memory available on our second configuration. These limitations could undoubtedly be overcome with larger hardware configurations.

The ease of application of our mechanism is a very important factor for reducing the turnaround time of the implementation.

In the current implementation, a single BDD variable order is used for all functions and sets represented, making it easy to combine and compare different functions and sets. We are planning to extend the work using different orderings; as analyzed in [14] and [13] this could drastically reduce problem size.

References

1. P. Arunachalam, C. Chase, and D. Moundanos. Distributred Binary Decision Diagrams for Verification of Large Circuits. In *Proc. IEEE ICCD'96*, pages 365–370, Austin, Texas, USA, October 1996.
2. P. Ashar and M. Cheong. Efficient Breadth-First Manipulation of Binary Decision Diagrams. In *Proc. IEEE/ACM ICCAD'94*, pages 622–627, San Jose, CA, USA, November 1994.
3. F. M. Brown. *Boolean Reasoning: the Logic of Boolean Equations*. Kluwer Academic Publishers, Boston, MA, USA, 1990.
4. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
6. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic Model Checking with Partitioned Transition Relations. In *Proc. VLSI'91*, pages 49–58, Edimburgh, Scotland, August 1991.
7. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on CAD*, 13(4):401–424, April 1994.
8. G. Cabodi, P. Camurati, and S. Quer. Improved Reachability Analysis of Large Finite State Machine. In *Proc. IEEE/ACM ICCAD'96*, pages 354–360, San Jose, CA, USA, November 1996.
9. H. Cho, G. Hachtel, S. W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. ATPG Aspects of FSM Verification. In *Proc. IEEE ICCAD'90*, pages 134–137, San Jose, CA, USA, November 1990.
10. O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Using Boolean Function Vectors. In *Proc. IFIP Int'l Workshop on Applied Formal Methods for Correct VLSI Design*, volume 1, pages 111–128, November 1989.

11. J. Jain, M. Abadir, J. Bitner, D. S. Fussel, and J. A. Abraham. IBDDs: An Efficient Functional Representation for Digital Circuit. In *Proc. IEEE EDAC'92*, pages 440–446, March 1992.
12. S. Kimura and E. M. Clarke. A Parallel Algorithm for Constructing Binary Decision Diagrams. In *Proc. IEEE ICCD'90*, pages 220–223, November 1990.
13. A. Narayan, A. J. Isle, J. Jain, R. K. Brayton, and A. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned-ROBDDs. In *Proc. IEEE/ACM ICCAD'97*, San Jose, CA, USA, November 1997.
14. A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned ROBDDs - A Compact, Canonical and Efficient Manipulable Representation of Boolean Functions. In *Proc. IEEE/ACM ICCAD'96*, pages 547–554, San Jose, CA, USA, November 1996.
15. H. Ochi, N. Ishiura, and S. Yajima. Breadth-First Manipulation of SBDD of Boolean Functions for Vector Processing. In *Proc. IEEE DAC'91*, pages 413–416, June 1991.
16. H. Ochi, K. Yasuoka, and S. Yajima. Breadth-First Manipulation of Very Large Binary Decision Diagrams. In *Proc. IEEE ICCAD'93*, pages 48–55, San Jose, CA, USA, November 1993.
17. R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley. Efficient BDD Algorithms for FSM Synthesis and Verification. In *IWLS'95: IEEE International Workshop on Logic Synthesis*, Lake Tahoe, CA, USA, May 1995.
18. R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli. Binary Decision Diagram on Network of Workstations. In *Proc. IEEE ICCD'96*, pages 358–364, Austin, Texas, USA, October 1996.
19. R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Proc. IEEE/ACM ICCAD'93*, pages 42–47, San Jose, CA, USA, November 1993.
20. F. Somenzi. CUDD: CU Decision Diagram Package – Release 1.0.4. In *Technical Report, Dept. of Electrical and Computer Engineering, of California*, Boulder, November 1995.