Distributed Binary Decision Diagrams

by

Oluwasola Mary Fasan

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science at the University of Stellenbosch

Department of Mathematical Sciences, Computer Science Division,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Dr. Jaco Geldenhuys

December 2010

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

December 2010

Summary

Binary Decision Diagrams (BDDs) are data structures that have been used to solve various problems in different aspects of computer aided design and formal verification. The large memory and time requirements of BDD applications are the major constraints that usually prevent the use of BDDs since there is a limited amount of memory available on a machine.

One way of overcoming this resource limitation problem is to utilize the memory available on a network of workstations (NOW). This requires the distribution of the computation and memory requirements involved in the manipulation of BDDs over a NOW.

In this thesis, an algorithm for manipulating BDDs on a NOW is presented. The algorithm makes use of the breadth-first technique to manipulate BDDs so that various BDD operations can be started concurrently on the different workstations on the NOW. The design and implementation details of the distributed BDD package are described. The various approaches considered in order to optimize the performance of the algorithm are also discussed. Experimental results demonstrating the performance and capabilities of the distributed package and the benefits of the different optimization approaches are given.

Opsomming

Binêre besluitnemingsbome (BBBs) is data strukture wat gebruik word om probleme in verskillende areas van Rekenaarwetenskap, soos by voorbeeld rekenaargesteunde ontwerp en formele verifikasie, op te los. Die tyd- en spasiekoste van BBB-gebaseerde toepassings is die hoofrede waarom BBBs nie altyd gebruik kan word nie; die geheue van 'n enkele is ongelukkig te beperkend.

Een manier om hierdie hulpbronprobleem te omseil, is om die gedeelde geheue van die werkstasies in 'n netwerk van werkstasies (Engels: "network of workstations", oftewel, 'n NOW) te benut. Dit is dus nodig om die berekening en geheuevoorvereistes van die BBB bewerking oor die NOW te versprei.

Hierdie tesis bied 'n algoritme aan om BBBs op 'n NOW te hanteer. Die algoritme gebruik die breedte-eerste soektegniek, sodat BBB operasies gelyklopend kan uitvoer. Die details van die ontwerp en implementasie van die verspreide BBB bilbioteek word beskryf. Verskeie benaderings om die gedrag van die biblioteek te optimeer word ook aangespreek. Empiriese resultate wat die werkverrigting en kapasiteit van die biblioteek meet, en wat die uitwerking van die onderskeie optimerings aantoon, word verskaf.

Acknowledgements

I give thanks to Him who is holy and faithful, the giver of wisdom and the owner of my life for all He has been to me, and for His mercy and grace upon me through the days, months and years. He makes a way in the wilderness and to them that are of no might He increases strength.

I wish to express my sincere gratitude to my supervisor, Dr. Jaco Geldenhuys for the time he has invested in me throughout the course of my studies. For all the guidance and kind words, I am very grateful. My appreciation also goes to everyone that has assisted me during my studies especially at the Department of Computer Science, Stellenbosch university. I am very grateful for the financial assistance I received from the University of Stellenbosch and the African Institute for Mathematical Sciences. I say a big thank you to all my friends for being there when I needed someone to lean on.

Finally, my appreciation goes to my family and my one and only B. for their faith in me, for all the encouragements and support no one else could give and for always bringing a smile to my face even in the toughest time. I love you all. Thank you.

For my maker and closest friend, I am nothing without HIS grace.

And for Mum and Dad for being ever caring and always believing in me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many areas in Computer Science depend heavily on Boolean algebra. Problems in system design and testing, combinatorics, artificial intelligence and mathematical logic can be expressed as a sequence of Boolean operations. The efficient representation and manipulation of Boolean functions is an important requirement for many algorithms used in the different application areas. Binary Decision Diagrams (BDDs) are data structures that provide such an efficient way of representing and manipulating Boolean functions, and have been used in various applications including circuit verification, combinatorial problems, symbolic model checking, finite state machines traversal and symbolic simulation.

BDDs are directed acyclic graph representations of Boolean functions which were first introduced in 1959 by Lee [27] and later widely popularized in 1986 by Bryant [8] after developing algorithms that can be used to efficiently manipulate them. Since this time, BDDs and their use in various application areas have been extensively studied by several researchers. The canonical representation of Boolean functions that BDDs provide has led to its wide use in several application areas and has also led to major breakthroughs in many of these areas. For example, in symbolic model checking, the use of BDDs has made it possible to verify systems with a very large number of states [12]. However, a major problem often encountered is that the size of the BDD representing a Boolean function may grow so large such that computation involving such BDDs becomes impossible to handle due to limited resources.

Over the years, different BDD packages for manipulating BDDs have been developed from various BDD algorithms with known complexities. Many of these packages use the conventional depth-first technique presented by Brace et al. [6]. Moreover, various techniques of speeding up the computation of BDDs and also reducing the size of BDDs generated during computation in order to combat the problem of arbitrary size which is the major drawback of BDDs have also been implemented. Some of these techniques include dynamic variable ordering, garbage collection, the use of specialized programming techniques for storing BDD nodes and other special higher-level algorithms [23]. However, these techniques may still fail because the manipulation of large BDDs is still often limited by the size of physical memory.

A major problem with the use of conventional depth-first algorithms in BDD manipulation is the random memory access pattern involved which results in a poor locality of reference and bad use of the CPU caches. An alternative way of manipulating BDDs in order to regularize memory accesses was presented by Ochi et al. [33]. Their approach involves a breadth-first manipulation of BDDs and leads to fewer page faults and allows larger BDDs to be handled. However, the major problem with their algorithm is that it is still limited by memory requirements. Moreover, the efficient swapping algorithm presented in their work which makes use of the processor swap space leads to the creation of redundant BDD nodes in the application.

Another approach that can be used to handle the resource limitation problem is to combine the resources available on a Network of Workstations (NOW) and to use distributed programming techniques. Some of the advantages of this approach which can be easily identified include:

1. Network communication is generally faster than disk accesses, so the approach is better than allowing the processor to use the swap space.

2. By making use of the collective resources available on the NOW, BDD applications can take advantage of the availability of a large amount of memory and possibly more processing power.

3. The approach does not require special hardware like a shared memory multiprocessor or a dedicated parallel computer; a NOW is usually easy to set up.

Various work has been done on how to parallelize BDD manipulation algorithms. Some of

the parallel BDD implementations that have been developed include packages for distributed shared memory (DSM) architectures [35, 25], and for vector processors [32]. However, these approaches are still limited by the amount of memory available on either the machine or the distributed shared memory. Other work that has been done on parallelizing BDDs include the work of Stornetta [42], Milvang-Jensen [30] and that of Ranjan et al. [37]. Details of their work are discussed in Section 2.6.

1.1 Thesis Goal

This thesis presents a distributed BDD manipulation package that uses the collective resources available on a NOW. The thesis gives a brief description of a non-parallel BDD manipulation algorithm and an implementation of the algorithm which forms the basis for the distributed BDD package. The major questions that need to be answered are:

1. How do we find an efficient way of distributing BDDs over the workstations on a NOW to use the collective memory available on the NOW?
2. How do we distribute the computation involved in BDD manipulation over the workstations in order to maximize our use of the computing power of each of the workstations on the NOW?
3. How do we make sure that each of the workstations executes different threads of computation simultaneously?
4. What is the effect of caching, the effect of different cache sizes, and the effect of caching different kinds of information during BDD computations?

This thesis gives a detailed description of the design and implementation of a distributed BDD package and the approaches used to resolve these questions. Techniques used to improve the performance of the distributed BDD package are discussed in detail, and the results of experiments conducted to evaluate the performance of the package are also presented.

1.2 Thesis Outline

Chapter 2 provides the basic background information necessary to understand Boolean functions and how BDDs are used for their representation. An overview of the major algorithms used in BDD manipulation and some of the various application areas of BDDs are presented. We give a brief description of the implementation of one of the modern sequential BDD packages available and also look at some of the previous attempts to distribute a BDD package over a NOW.

The core of the thesis is Chapter 3 which includes a detailed discussion of the design and implementation of the distributed BDD package developed. First, an implementation of the non-distributed BDD package which forms the basis for the distributed package is briefly discussed since the distributed BDD package uses similar data structures. The rest of chapter discusses how the major tasks involved in the distribution of a BDD application are handled in the implemented package and how our distributed BDD package compares to other similar packages.

Chapter 4 describes different approaches for improving the performance of the distributed BDD package developed and how they are implemented. We discuss the details of two levels of caching and an alternative way of distributing the memory and computational requirements of a BDD application. A new technique for analyzing the performance of a distributed BDD package for any specific problem is explained. The benefits of the various optimization techniques considered are also examined.

Results of experiments conducted to measure the performance of the distributed BDD package developed are discussed in Chapter 5 of the thesis. The performance of the various optimization techniques considered are measured. These experiments were conducted using the high performance computing (HPC) cluster at the University of Stellenbosch.

Chapter 6 presents the conclusions of the thesis and proposes various ideas for future work.

Chapter 2

Background

Over the last two decades, various application areas of Boolean functions have benefited from the symbolic representation and manipulation of Boolean functions [12, 16, 15, 10, 11, 29]. The efficiency of many of these applications depends on the data structure used to represent the Boolean functions involved. An efficient way of symbolically representing Boolean functions known as Binary Decision Diagrams (BDDs) which has made it possible to solve various complex problems in applications involving Boolean function manipulations was presented by Bryant [8] in 1986 and has been extensively studied by various researchers since then.

This chapter describes the details necessary to understand BDDs. Section 2.1 presents a brief overview of Boolean functions and other approaches that have been used for representing and manipulating Boolean functions. Details about BDDs are presented in Section 2.2 up to Section 2.4. The chapter concludes in Section 2.5 with a brief description of a modern sequential BDD package known as CUDD [40].

2.1 Boolean Functions

A Boolean function is of the form:

$$f : \mathcal{B}^k \rightarrow \mathcal{B}$$

where $\mathcal{B} = \{0, 1\}$ and k is a non-negative integer. The set $\mathcal{B}$ is the set of Boolean values whose elements are sometimes referred to as *false* and *true* instead of 0 and 1, respectively. For any k, there are exactly 2^{2^k} possible Boolean functions.

Boolean functions are used for expressing the relation between different Boolean variables. A Boolean expression is composed of Boolean variables, $x, y, \ldots$, Boolean values, *true* (1) and *false* (0) and also the Boolean operators: conjunction $\wedge$, disjunction $\vee$, negation $\neg$, implication $\Rightarrow$, and bi-implication $\Leftrightarrow$. Formally, Boolean expressions are generated by the grammar:

$$t ::= x \mid 0 \mid 1 \mid \neg t \mid t \wedge t \mid t \vee t \mid t \Rightarrow t \mid t \Leftrightarrow t$$

where x can be any element of a set of Boolean variables or Boolean values. Parentheses and operator priorities are used to resolve ambiguities. Usually, the priorities of the operators (starting from the highest) are: $\neg$, $\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$ [22]. One example of a Boolean expression is:

$$\neg x_1 \Rightarrow x_2 \vee x_3.$$

To make the priorities absolutely clear, the expression can also be written as:

$$((\neg x_1) \Rightarrow (x_2 \vee x_3)).$$

A Boolean expression describes how to determine a Boolean output value based on logical calculations on some Boolean variables and values. The sequence of assignments of values to Boolean variables is referred to as a truth assignment (or interpretation) and is written as:

$$[1/x_1, 0/x_2, 0/x_3]$$

which means that a value 1 is assigned to x_1 and 0 is assigned to x_2 and x_3. A truth assignment for a given expression evaluates to either 0 or 1. For example, the truth assignment $[1/x_1, 0/x_2, 0/x_3]$ evaluates to 1 in the above expression while the truth assignment $[0/x_1, 0/x_2, 0/x_3]$ evaluates to 0 for the same expression.

Two Boolean expressions p and q are said to be equivalent if they yield the same output values for all truth assignments. A *tautology* is a Boolean expression that yields the value 1 for all possible truth assignments whereas a *contradiction* is one that always yields 0. A Boolean expression is said to be *satisfiable* if it yields the value 1 for at least one truth assignment.

In practice, some of the tasks for which Boolean functions are used include testing for satisfiability and checking for equivalence. Many of these tasks require solutions to NP-complete or co-NP-complete problems [8]. Given our present knowledge, the amount of time and memory required to complete them grows exponentially in the size of the problem. Some of the methods that have been used for representing Boolean functions include the use of classical representations like truth table, Karnaugh maps and prime cubes. However, all these approaches have their drawbacks because they yield representations of exponential size for some common functions. Moreover, for a given function, they may give more than one representation or in cases where the representations are not of exponential size, performing a simple operation may lead to a function with exponential representation. Thus, testing for equivalence and satisfiability can be difficult. In addition, for all the different approaches, the time required to perform these operations also grows exponentially with the size of the problem. Thus, there is a need for an efficient way of representing and manipulating Boolean functions so that the size of the representations will be reasonable and the exponential computations will be avoided.

2.2 Binary Decision Diagrams

As mentioned earlier, BDDs were first introduced by Akers [1] and Lee [27] and were later popularized by Bryant [8] when he presented a restricted form of BDD known as the Reduced Ordered Binary Decision Diagram (ROBDD) which can be used to efficiently represent and manipulate Boolean functions. The main idea behind BDDs is the Shannon expansion. For a function f and variable x, the Shannon expansion[1] is:

$$f = x \cdot f_{|x=1} + \overline{x} \cdot f_{|x=0}.$$

The Shannon expansion is a way of expressing a Boolean function as a sum of the positive and negative Shannon cofactors of the function. The positive Shannon cofactor of a function f with respect to a variable x is described as the function f with all values of x set to 1 while the negative Shannon cofactor is the function f with the values of x set to 0. When expressed

[1]The notation "$\cdot$" refers to the logical *and* ($\wedge$) operation, "$\overline{x}$" means *not* x or negation of x and "$+$" means the logical *or* ($\vee$) operation.

over several variables $x_1, x_2, \ldots, x_n$, the expansion can be written as:

$$f(x_1, x_2, x_3, \ldots, x_n) = x_1 \cdot f(1, x_2, x_3, \ldots, x_n) + \overline{x}_1 \cdot f(0, x_2, x_3, \ldots, x_n).$$

BDDs are directed acyclic graphs (DAG) consisting of decision nodes where each node either has two outgoing edges or none. The terminal nodes in the graph are labeled "T" or "F" corresponding to the *true* and *false* values, respectively. The root of the tree has no incoming edges and there is only one root. Each internal node of the DAG is labeled with a variable taken from the set of variables over which the function is defined. The output edges are labeled "1" or "0" and are often referred to as the THEN and ELSE edges or the *true* and *false* edges, respectively. Each edge from a node in the graph leads to another node called the child node of the node.

An Ordered Binary Decision Diagram (OBDD) has a total ordering of the associated Boolean variables such that along every path of the BDD starting at the root and terminating at a "T" or "F" node, the variables associated with the nodes occur in a given linear order that is the same for all possible paths. An OBDD is reduced (and called a Reduced Ordered Binary Decision Diagram) if each node in the BDD represents a unique function. That is, it contains no duplicate or redundant nodes. Given any BDD, an ROBDD is generated by performing the following operations:

1. Merge all isomorphic subgraphs, that is, similar nodes are shared and not duplicated.

2. Eliminate any node whose two children are identical. That is, if the two edges of a node lead to the same child node, the node is deleted and its incoming edge(s) is directed to its child node.

For example, the DAG representing the Boolean function $(x_1 \vee x_2) \wedge (x_1 \vee x_3)$ shown in Figure 2.1 can be transformed into the ROBDD shown in Figure 2.2 by applying these two rules.

As Bryant shows, an important property of the ROBDD is that for any given ordering of the variables, a Boolean function has a unique representation [8]. This property makes it useful in checking for the equivalence of two Boolean functions by checking if they have the same representations. Another property of BDDs that can be easily seen from the example is that

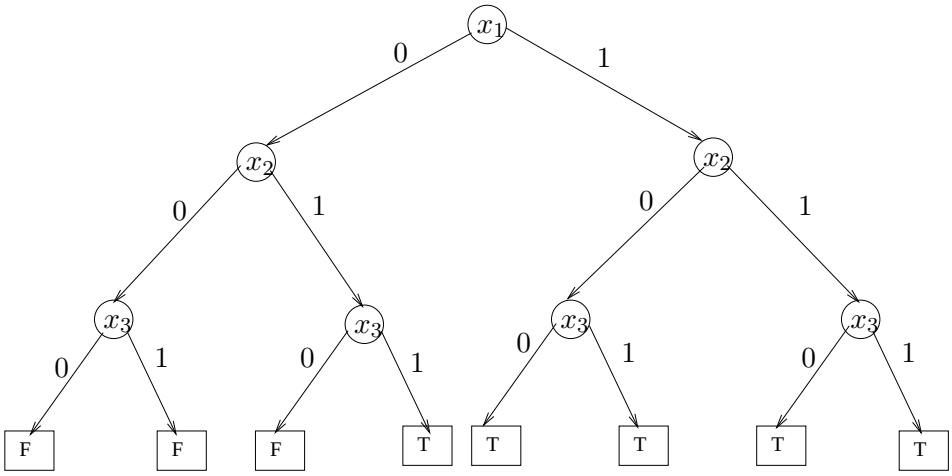


Figure 2.1: A DAG representing the Boolean function $(x_1 \vee x_2) \wedge (x_1 \vee x_3)$

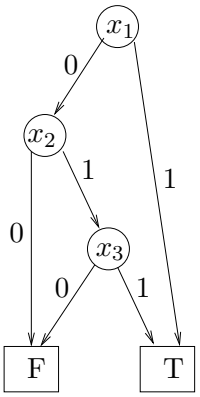


Figure 2.2: ROBDD representation of the Boolean function $(x_1 \vee x_2) \wedge (x_1 \vee x_3)$

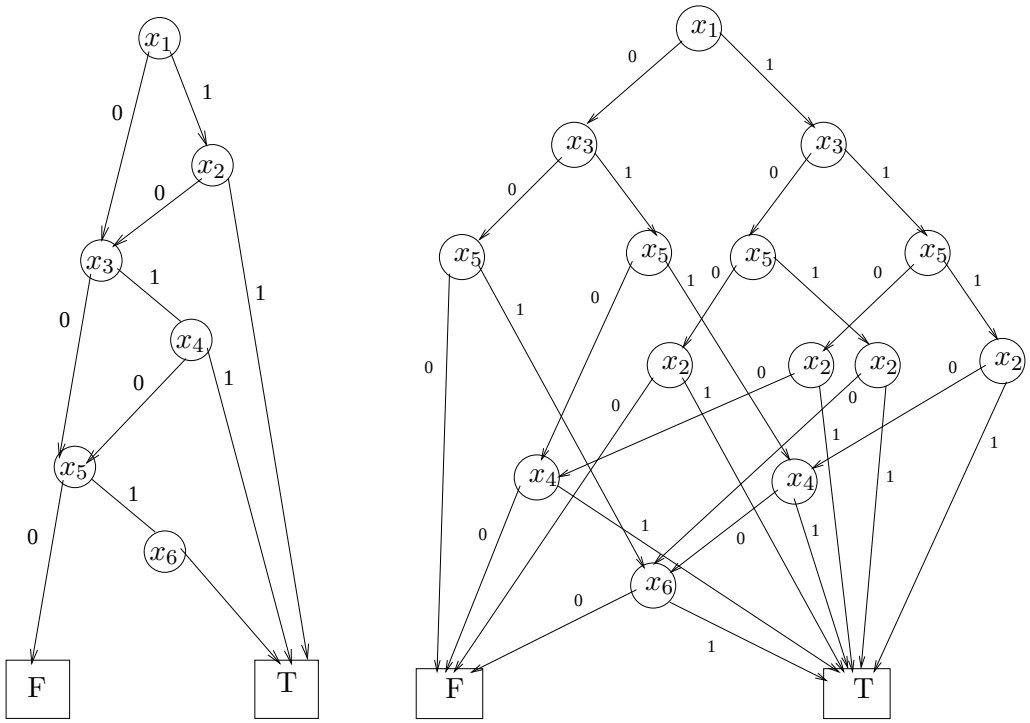
BDD representations are very compact. The difference in the size of the BDD and binary tree representations is an illustration of this fact. In the rest of this thesis, we shall simply refer to Binary Decision Diagrams (BDDs) instead of Reduced Ordered Binary Decision Diagrams (ROBDDs) since all our BDDs will be of this form.

In the BDD representation of the Boolean function $(x_1 \vee x_2) \wedge (x_1 \vee x_3)$ shown in Figure 2.2, the path through the BDD leads to "T" if and only if the vector $\langle x_1 x_2 x_3 \rangle$ which corresponds to the values selected at each of the nodes x_i is an element of the set $\{\langle 011 \rangle, \langle 100 \rangle, \langle 101 \rangle, \langle 110 \rangle, \langle 111 \rangle\}$

(which is also true for the first DAG representation). A function representing a BDD is satisfiable if and only if the BDD contains a terminal vertex labeled "T".

2.2.1 Variable Ordering

In practice, the size of the BDD representation of any function depends on both the function and the chosen ordering of the variables over which the function is defined. The ordering of Boolean variables is very important because it has a crucial effect on the size of BDDs. For example, Figure 2.3 shows two different BDDs (with different variable ordering) for the same function $f = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$.



(a) $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$

(b) $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$

Figure 2.3: BDD representations for different variable orderings

Using the variable ordering $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$, the BDD representing the function f has 6 internal nodes whereas if the variable ordering $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$ is used, the BDD representation of f consists of 14 internal nodes. In some applications, a particular

ordering is usually chosen at the outset and BDDs are then constructed using this ordering. It is difficult to determine how good a particular variable ordering will be and the problem of finding the best variable ordering is NP-hard [5]. However, there are heuristic techniques used to handle this problem, and dynamic variable reordering [38] is also widely used.

```
01: Apply(A, B, op) {
02:      if IsTerminalCase(A) or IsTerminalCase(B)
03:           return (A op B)

04:      else if Apply(A,B,op) is in computed-table
05:           return result

06:      else
07:           T = Apply(Then(a1), Then(b1), op)
08:           E = Apply(Else(a1), Else(b1), op)

09:      if T = E
10:           return T
11       result = Node(minimum(A.var, B.var), T, E)
12:      if InUniqueTable(result)
13:           return result
14:      insert in the computed-table (Apply(A,B,op), result)
15:      return Makenode(result)
16: }
```

Figure 2.4: Simplified implementation of the *Apply* algorithm

2.2.2 BDD Operations

The ability to efficiently perform operations on functions represented by BDDs is one of the most important features of BDDs. As proposed by Bryant [8], there are various algorithms for efficient manipulation of BDDs. The most common operations on BDDs are based on the *Apply* and *If-Then-Else* (ITE) algorithms.

The *Apply* algorithm is the general algorithm for implementing all binary Boolean operations. The algorithm takes three arguments: two BDDs A and B and a Boolean operator $*$. It returns another BDD representing the function,

$$f_C = f_A * f_B$$

which is defined as

$$(f_A * f_B)(x_1, \ldots, x_n) \;=\; (f_A(x_1, \ldots, x_n)) * (f_B(x_1, \ldots, x_n)).$$

The construction of the *Apply* algorithm is based on the Shannon expansion. It is easy to show that for all Boolean operators $*$,

$$f_1 * f_2 = x_i \cdot (f_1 |_{x_i=1} * f_2 |_{x_i=1}) + \overline{x}_i \cdot (f_1 |_{x_i=0} * f_2 |_{x_i=0}).$$

A simplified implementation of the *Apply* algorithm is shown in Figure 2.4. To apply an operator to two Boolean functions represented by BDDs A and B with roots a_1 and b_1, respectively, the different possible cases are considered as shown in the algorithm. The simple cases of the operands are handled first, that is, when a_1 or b_1 or both is a terminal node. Otherwise, recursive calls of the *Apply* algorithm are made using the THEN and ELSE edges of a_1 and b_1 until a terminal node is reached. Lines 9–10 ensure that there are no redundant nodes, while lines 11–13 ensure that there are no duplications. The algorithm has a worst-case time complexity of $O(|A| + |B| + |C|)$ where C is the BDD representing the result of the operation and $|A|$ is the number of BDD nodes in A [8]. As we shall discuss in Section 3.1, the check for node duplication is typically done using a *unique-table* [6] implemented as a hash table; this maintains the canonicity of the BDD. We also make use of memoization by keeping a cache of all operations already done. The cache which is called the *computed-table* helps to improve the efficiency of the *Apply* algorithm.

$$\begin{array}{lcl}
Apply(A, B, \wedge) & = & Apply(x_1, x_3, \wedge) \\
& T_1 = & Apply(x_2, x_3, \wedge) \\
& & T_2 = Apply(true, x_3, \wedge) = x_3 \\
& & E_2 = Apply(false, x_3, \wedge) = false \\
& & return(x_2.var, x_3, false) \\
& E_1 = & Apply(false, x_3, \wedge) = false \\
& return & (x_1.var, T_1, false) \\
Apply(A, B, \wedge) & = & (x_1.var, T_1, false)
\end{array}$$

Figure 2.5: Recursive use of the *Apply* Algorithm

An example of the application of the *Apply* algorithm is shown in Figure 2.5 for computing the conjunction of two BDDs representing the functions, $A = x_1 \wedge x_2$ and $B = x_3 \wedge x_4$. The

algorithm is applied recursively through both BDDs to generate the result. The two BDDs and the result generated from the conjunction are shown in Figure 2.6.

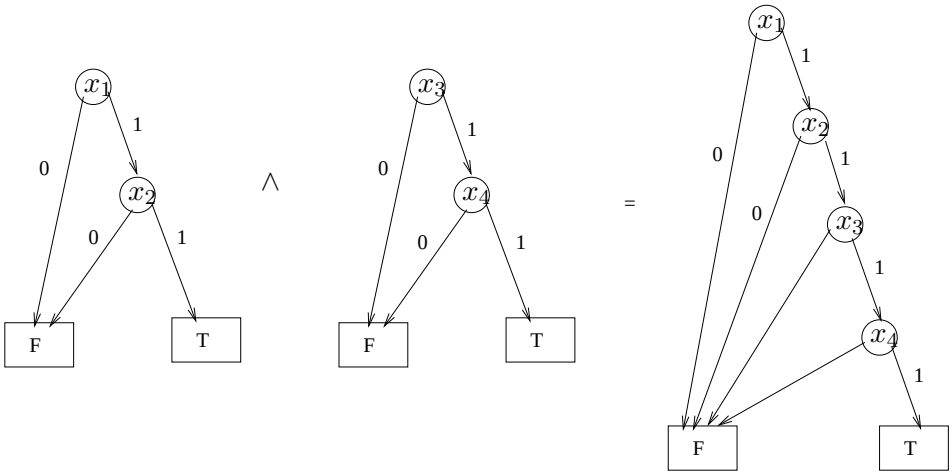


Figure 2.6: Conjunction of two BDDs

```
01: ITE(A, B, C) {
02:      if IsTerminalCase(A) or IsTerminalCase(B)
03:           return result
04:      else if Computed-table has Entry (A, B, C)
05:           return result;
06:      else
07:           let x be the top variable of (A, B, C)
08:           T = ITE(A_x, B_x, C_x)
09:           E = ITE(A_x', B_x', C_x')
10:           if T = E
11:                return T
12:      R = findInUniqueTable_Or_Add(Node(x, T, E))
13:      return R
14: }
```

Figure 2.7: The ITE algorithm

The second algorithm used in many BDD operations is the ITE (If-Then-Else) algorithm which is very similar to the *Apply* algorithm. It takes three arguments A, B and C which are all BDDs and returns a BDD D resulting from the If-Then-Else operation which is defined as: *if*

A then B, else C. The Boolean representation of the ITE algorithm can be expressed as:

$$ITE(A, B, C) = A \cdot B + \overline{A} \cdot C.$$

An outline of the ITE algorithm as presented by Brace et al. [6] is shown in Figure 2.7. The term `A_x` (`A_x'`) refers to the BDD representation of `A`, with the values of $x = 1$ ($x = 0$). Lines 12–13 are equivalent to lines 11–15 of the *Apply* algorithm as both are used to ensure that there are no duplication of nodes or repetition of computation. The algorithm can be used to implement all binary Boolean operations [6] and thus it can also be used to express an *Apply* operation. For example, $Apply(A, B, \vee) = ITE(A, 1, B)$ and $Apply(A, B, \wedge) = ITE(A, B, 0)$. Table 2.1 shows the ITE implementation of various Boolean operations. The time complexity of the ITE algorithm is $O(|A| \cdot |B| \cdot |C|)$. In practice, the number of computation steps in both the *Apply* and *ITE* algorithms is normally close to the size of the resulting BDD.

	Boolean expression	ITE expression
1	0	0
2	$f \cdot g$	$ITE(f, g, 0)$
3	$f \cdot \overline{g}$	$ITE(f, \overline{g}, 0)$
4	f	f
5	$\overline{f} \cdot g$	$ITE(f, 0, g)$
6	g	g
7	$f \oplus g$	$ITE(f, \overline{g}, g)$
8	$f + g$	$ITE(f, 1, g)$
9	$\overline{f + g}$	$ITE(f, 0, \overline{g})$
10	$\overline{f \oplus g}$	$ITE(f, g, \overline{g})$
11	$\overline{g}$	$ITE(g, 0, 1)$
12	$f + \overline{g}$	$ITE(f, 1, \overline{g})$
13	$\overline{f}$	$ITE(f, 0, 1)$
14	$\overline{f} + g$	$ITE(f, g, 1)$
15	$\overline{f \cdot g}$	$ITE(f, \overline{g}, 1)$
16	1	1

Table 2.1: ITE implementation of all two variable Boolean functions

An example of the ITE algorithm evaluating $ITE(A, B, C)$ is shown below. The BDDs A, B,

C and the resulting BDD D are as shown in Figure 2.8.

$$\begin{aligned} D &= ITE(A, B, C) \\ &= (u, ITE(A|_{u=1}, B|_{u=1}, C|_{u=1}), ITE(A|_{u=0}, B|_{u=0}, C|_{u=0})) \\ &= (u, ITE(true, G, C), ITE(E, false, C)) \\ &= (u, G, (v, ITE(E|_{v=1}, false, C|_{v=1}), ITE(E|_{v=0}, false, C|_{v=0}))) \\ &= (u, G, (v, ITE(true, false, true), ITE(false, false, H))) \\ &= (u, G, (v, false, H)) \end{aligned}$$

The evaluation of the function $ITE(A, B, C)$ is also the same as evaluating the function:

$$D = Apply(Apply(A, B, \wedge), Apply(\overline{A}, C, \wedge), \vee)$$

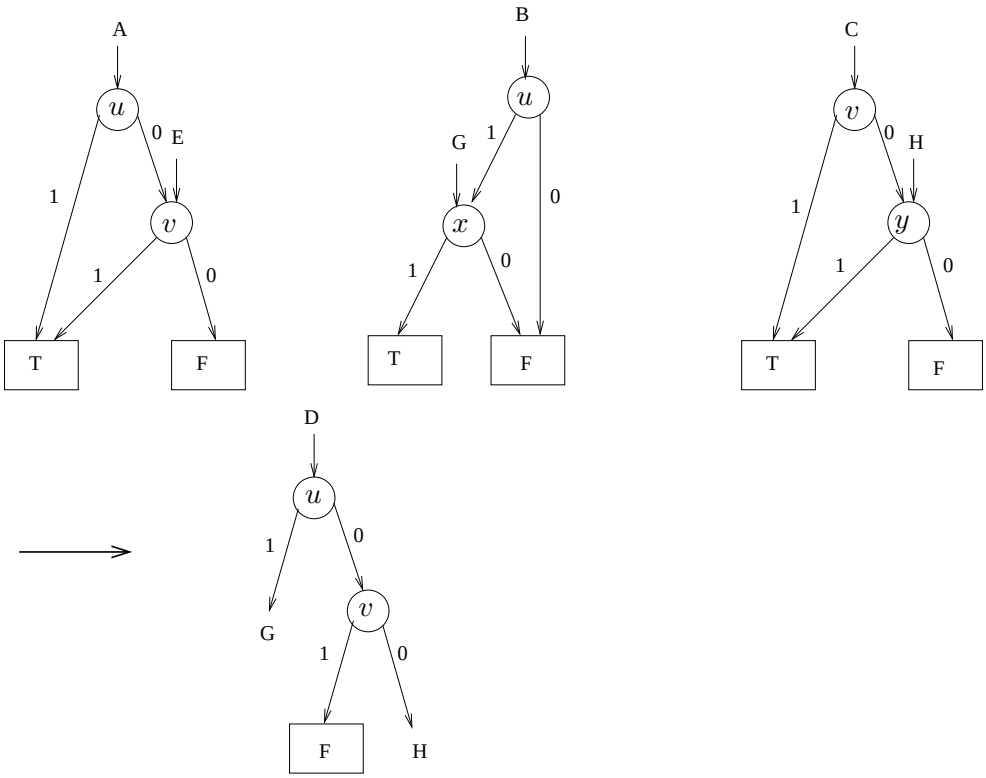


Figure 2.8: An example of ITE algorithm

Some other algorithms [8] used for efficient manipulation of BDDs include:

- *Restrict*: The algorithm is used to construct a restricted form of a BDD. That is, given a truth assignment for a BDD f, the algorithm constructs the corresponding BDD for f under this truth assignment. In other words, the algorithm transforms a BDD f into a BDD representing the function $f\,|_{x_i=c}$ for some variables x_i and Boolean values $c \in \{0, 1\}$.

- *Compose*: Given two formulas g and h, the composition algorithm derives the BDD representing the function f which is a composition of g and h defined as;

$$f = g\,|_{x_i=h} = (h \wedge g\,|_{x_i=1}) \vee (\overline{h} \wedge g\,|_{x_i=0})$$

- *Satisfy-one*: The algorithm is used to decide whether a function f is satisfiable for some input a, that is, if $f(a) = 1$.

- *Satisfy-all*: The algorithm computes the list of all satisfying truth assignments for a Boolean function f. That is, it returns the list of all a such that $f(a) = 1$.

- *Satisfy-count*: The algorithm returns the number of truth assignments satisfying a Boolean function f. That is, it returns the total number of all a such that $f(a) = 1$.

These functions are useful for performing operations like equality testing, satisfiability, existential and universal quantification, and concatenation. Existential and universal quantification of the variables in a function are done in time quadratic in the size of the BDD representing the function. More details about some of the basic operations handled with these algorithms are given in Section 3.4.4.

2.3 Applications of BDDs

The ability to efficiently manipulate BDDs have led to their wide use in various application areas over the last two decades. For any problem domain, in order to apply BDDs, the data to be represented are expressed as Boolean functions. The necessary results are obtained by carrying out a sequence of operations on the BDDs representing the Boolean functions. Some of the various application areas of BDDs include formal verification (especially symbolic model checking), optimization of logic circuits, and testing and optimization of sequential circuits.

2.3.1 Application of BDDs in Verification and Model checking

A detailed description of the sequence of state transitions of a system is often required in order to solve many of the problems in digital system verification. Algorithms that construct an explicit representation of the state graph in order to handle this problem are inefficient because digital systems usually have a very large number of states. BDDs have become a major data structure used in formal verification. The various aspects of formal verification in which BDDs have been applied include verification of combinatorial and sequential circuits, symbolic simulation, and symbolic model checking.

The verification of combinatorial circuits is the problem of proving the equivalence of two circuits usually a verified circuit and an unverified circuit. A formal proof of correctness of the unverified circuit is achieved by computing the BDDs of the functions representing both the verified circuit and the unverified one. The problem is reduced to checking for the equivalence of these two BDDs. A major limitation often encountered in circuit verification is that BDDs representing large circuits are often very large themselves, exhausting the memory on machines handling the BDDs. Some of the numerous studies that have been done in order to simplify the verification of large combinatorial circuits using BDDs include the work of Brand [7], Shin [39], and Lai and Sastry [26].

Unlike combinatorial circuits, sequential circuits are verified by checking the equivalence of a deterministic finite state machine M against a specification M' also given as a finite state machine. This verification, which can be reduced to a reachability problem, requires a compact representation of the finite state machines in order to handle large systems. BDDs are used for these compact representations. The reachability problem is reduced to a number of conjunction, disjunction and existential operations on the BDDs. The reduction to a reachability problem is also a major part of the symbolic model checking technique which is used for automatically verifying finite state systems. As defined by Burch et al. [12], model checking is the process of determining whether a given formula is true in a given model of a system. Since systems to be verified are often very large, an explicit enumeration of the set of states may be impossible. Symbolic model checking uses BDDs to describe the set of states implicitly. The check for satisfiability is done using the BDDs representing the set of states and the propositional formulas.

Systems with very large number of states have been verified using model checking techniques based on the implicit representation of the set of states [12, 29]. More detailed descriptions of the verification of sequential circuits using BDDs are presented by Coudert, Berthet, and Madre [16, 15], Clarke et al. [10] and Burch et al. [11].

The use of BDDs in symbolic model checking has proven to be an efficient technique for combating the *state explosion problem* often encountered in automated verification. This problem arises because for very large systems, the number of states grows exponentially in the number of the components of the system. Although BDDs can be used to handle this problem, a related problem called the *node explosion problem* is encountered when BDDs are used to represent very large systems. This is because intermediary BDDs that arise during the computation are often large even though the final BDD may be small. This results in high memory and computation time requirements. Other approaches to using BDDs for model checking include the work of Burch et al. [13, 12] and Brayton et al. [43].

Some of the other application areas of BDDs include protocol verification, and CAD applications such as functional simulation [2, 28], logic synthesis [44] and test generation [14]. More applications of BDDs are highlighted by Bryant [9].

2.4 Advantages and Disadvantages of BDDs

The popularity of BDDs in various application areas can be attributed to their ability to efficiently represent and manipulate Boolean functions. Apart from the fact that BDDs provide a canonical presentation of Boolean functions, which makes it possible to easily test functional equivalence, procedures involved in performing operations on BDDs are also simple. The number of computational steps usually involved in an operation is always less than the product of the sizes of the operand BDDs (and not more than the size of the resulting BDD). In addition, many interesting functions have compact BDD representations. Thus, most operations on BDDs can be performed relatively fast.

Moreover, a single BDD structure can be used in the representation of several functions thus saving more space and making the manipulation faster and more efficient. The use of BDDs

in the various applications areas has lead to a major breakthrough in most of these areas. For example, the use of BDDs have made it possible to verify very large circuits and systems [12].

However, despite all the advantages of using BDDs, a major problem with the use of BDDs which is often encountered in most of the application areas is that for some large systems and circuits, BDDs constructed during computation often grow extremely large resulting in high memory and time requirements. This often makes it impossible for a single machine to handle their computation and thus hinder the use of BDDs in such application area or problem.

2.5 Sequential BDD Packages

Since the popularization of BDDs by Bryant [8], various BDD packages have been developed by different people. As mentioned earlier, many of these packages share a number of common implementation features which are based on the work of Brace et al. [6] and Rudell [38]. In this section, we give a brief description of one of the modern BDD packages known as CUDD. The description of the CUDD package given below is based on the documentation of the package [40]. Other packages that have been developed include CAL [36], TiGeR [17] and ABCD [4].

The CU decision diagram (CUDD) package is a sequential BDD package based on depth-first traversal of BDDs. The package provides various functions for the manipulation of BDDs, Algebraic Decision Diagrams (ADDs) [3] and Zero-suppressed Binary Decision Diagrams (ZDDs) [31]. In CUDD, a BDD is represented as a pointer to a structure containing several fields including the variable index, the reference count and the node. BDD nodes are stored in a unique-table implemented as a hash list. The hash list is used to guarantee the uniqueness of each of the BDD nodes. In addition, the package also contains several heuristics for dynamic variable reordering which are used to reduce the size of the decision diagrams.

The CUDD package uses a cache which is also implemented as a hash list to store computed results. It typically starts with a small cache which is then increased until it no longer affects the computation involved or until a limit size is reached. The user is allowed to choose both the initial and the limit values for the cache size. The optimal value for the cache usually depends on the specific problem being handled. The cache is always cleared when dynamic variable

reordering takes place.

In addition, the package uses garbage collection for reclaiming memory occupied by nodes that are no longer used. The technique is implemented by keeping reference counts for each node. A node is marked as *dead* when its reference count becomes zero. In order to optimize the performance of the package, garbage collection only takes place when the number of dead nodes reaches a given level which is dynamically determined by the package [40]. All cache entries pointing to a dead node are removed when garbage collection is done. The CUDD package is widely regarded as a very efficient BDD package and is publicly available.

2.6 Distributed BDD Packages

There are many ways to address the resource limitation problem often encountered during BDD computation. Some approaches involve minimizing the size of BDDs, while others involve the use of parallel processing to accelerate BDD operations. Some of the several attempts that have been made in order to minimize the size of BDDs include modification of the BDD structure and alternative representations of the transition relations or system states. However, not all of these attempts have been successful at minimizing large intermediary BDDs [19].

A fair amount of research has been done on using parallel processing to speed-up BDD computation time and provide more memory. Most of them used a parallel distributed memory multi-processing environment. However, not many of these distributed memory architectures actually use a network of workstations. Some of the recent work that have been done on parallelizing BDDs include the work of Stornetta and Brewer [42]. They present a BDD package that is suitable for a distributed memory multi-processor. The package allows depth-first algorithms on BDDs to be performed in parallel. According to their scheme, tasks are distributed to the processors by considering the node with the highest level in a given computation's arguments. Although this technique leads to an excellent distribution of tasks, it results in a very high computation overhead. Moreover, their algorithm exhibits speed-up only when compared to a single machine that is running out of memory.

Milvang-Jensen and Hu [30] also introduced a parallel BDD package which is based on the

CUDD package [40]. Their package makes it possible to perform several different BDD operations in parallel using breadth-first algorithms. The package is designed to run on multiple machines using the parallel virtual machine (PVM) library [20]. According to their algorithm, tasks are distributed based on the topmost variable of the operand BDDs. The main problem with their approach is the lack of efficient ways to balance the work [30]. However, for large BDDs, they are the first to report a speed-up of computation on a distributed memory over computation on a single machine provided that the parallel version is running on a certain minimum number of processors.

Another algorithm presented by Ranjan et al. [37] handles memory limitations by manipulating BDDs using a Network of Workstations (NOW). In their approach, BDD variables are distributed among the workstations such that all variables assigned to a workstation are consecutive. Each workstation handles operations involving the BDDs that are assigned to it. Their implementation uses the PVM library to provide the necessary communication between the workstations during BDD manipulation. The study also pointed out the potential impact of distributing BDDs on a network of workstations. The major drawbacks in their implementation include the fact that the performance of the algorithm is hampered by the network overhead resulting from the number of remote requests made to perform the BDD operations. Their approach also results in a duplication of effort due to its inability to recognize requests that have been earlier processed. Moreover, the equal distribution of variables to workstations leads to an uneven distribution of the workload when the number of nodes in certain levels grows very large. However, a better approach of dynamically distributing the variables among the processors in order to balance the load was proposed.

Our approach to parallelizing BDDs also involves the use of a NOW. Some of the features are similar to the work of Ranjan et al. but with additional functionality. We also use the level-by-level distribution of the variables over the NOW. However, we implement two different levels of caching for operations already performed thus reducing the number of network accesses which constitutes a major problem when dealing with a NOW. The problem of uneven distribution of workload is addressed by providing a way for the user to distribute the variables more flexibly. Details regarding our approach of distributing BDDs over a NOW and the differences between our work and that of Ranjan et al. are presented in Section 3.3 through 3.6.

Chapter 3

Design and Implementation

The previous chapter presented background details of BDDs. In this chapter, we describe the design and implementation of our distributed BDD package. Section 3.1 describes the implementation of a non-distributed BDD package. The non-distributed implementation makes it easier to explain the problems encountered in BDD manipulation and how they are handled. It is also useful to compare the performance of the distributed BDD package to the non-distributed version. Our implementation of a distributed BDD package is described in Section 3.2. The distributed BDD application is based on the non-distributed package. Both implementations are done using the C programming language which was chosen to provide more control over the hardware we use.

3.1 Non-distributed BDD Package

Several BDD packages have been developed since the work of Bryant [8] in 1986 to run on single machines. Our implementation of the non-distributed BDD package is similar to the package described by Brace et al. [6]. The *Apply* and *ITE* algorithms form the major part of the implementation of a BDD package since they can be used to express any binary Boolean operation. As shown in Figure 2.4, the *Apply* algorithm is a depth-first recursive algorithm that performs operations by traversing the operand BDDs from top to bottom on a path-by-path basis.

A BDD package is implemented as a library of BDD manipulation routines which are made available to the user. However, it is not necessary for the user to understand the details of the construction of the routines because the implemented Boolean operations can be used without changing the routines.

The two basic BDD nodes in a BDD package are the terminal nodes *true* and *false*. BDD nodes are usually constructed starting from the input variables and then performing desired operations to produce the output. Given that the input variables obey some variable ordering, the implementation constructs BDDs obeying the same ordering. All BDD operations are implemented using a common ordering.

A BDD node is basically a pointer to memory (containing the variable number of the node, and the left and right child nodes). BDD nodes are stored in a table called the unique-table [6] using the `makenode` routine shown in Figure 3.1. The unique-table, which is built as a hash table, maintains the canonical property of BDD nodes and each node is identified by a unique id. A lookup of the unique-table is always done before a BDD node is added to the table. If the node is found, the already stored node is used, otherwise the new BDD node is added to the unique-table. Thus, each node in the unique-table represents a unique Boolean function which is only stored once in the table even if the same function is constructed in different ways. The use of a unique id for representing each node in the unique-table makes it possible to do an equivalence test by simply testing if the two pointers are the same.

```
01: makenode ({varnr, left, right}) {
02:     if (left = right)
03:         return left
04:     else
05:          R = findInUniqueTable_Or_Add({varnr, left, right})
06:     return R
07: }
```

Figure 3.1: The routine for constructing a BDD node

Another table which is implemented in the non-distributed BDD package to improve the performance of the application is the computed-table [6]. Since there are potentially many paths to get to the terminal nodes of a BDD, the computation necessary to perform a recursive operation

is reduced by keeping track of the intermediate computational results. The computed-table is implemented as a hash-based cache with a fixed maximum size. New computational results are stored by computing a hash function and storing the operands and operation leading to the result together with the result. The efficiency of the computed-table is improved by storing an entry only once in the table. If a new operation that has been performed earlier, is to be repeated, and the result of the operation is still present in the computed-table, the result is returned immediately instead of performing the same operation again.

The storing of intermediate results of Boolean operations causes several results to be stored during BDD computation, some of which might not be useful once the desired result is obtained. Thus, it is important to be able to release the memory used by these BDDs. However, a BDD node can be referenced from various locations in the package. For example, apart from the single reference of the BDD node in the unique-table, a node can be referenced many times by other nodes and can possibly also appear in the computed-table. This implies that in order to free a BDD node from memory, one must make sure that no other nodes are pointing to it from anywhere in the package.

A memory management technique called garbage collection is implemented to periodically free unused memory. Garbage collection can be implemented by keeping a reference count for each BDD node in order to know when the node is no longer active. The reference count for a node is incremented when a new BDD node points to it and decremented when a node pointing to it is freed from memory. A node is removed from the memory when its reference count becomes zero. That is, when it is found only in the unique-table. Garbage collection can also be implemented using the “stop-and-copy” or “copying” algorithm [24]. In this case, the available memory on a machine is divided into two, and BDD computation is completed on only one part of the memory. However, when the currently active part of the memory becomes full, BDD computation is paused and all the BDD nodes that are pointed to by another node are copied to the second memory partition. All other nodes left in the active memory partition after copying are deleted since it implies that no other BDD node is pointing to them. The second memory partition then becomes the active memory used for computation. The swap is repeated as each memory partition gets full.

Garbage collection in our non-distributed package is implemented using the “mark-and-sweep”

algorithm [24]. The algorithm involves the marking and unmarking of BDD nodes. To perform garbage collection, all the BDD nodes in the memory are first unmarked. Starting from the root node, each node that is pointed to from another BDD node is then marked. All unmarked BDD nodes are then removed from the memory since it implies that they are no longer needed since no node is pointing to them. Garbage collection is performed at different points in the package to free memory used by nodes that are no longer needed. The use of garbage collection is important because the amount of memory used keeps increasing during BDD manipulation and the memory limit can be reached before the end of the execution if some precaution is not taken. Moreover, even when the memory limit is not yet reached, accessing nodes in the unique-table becomes slower as the table grows fuller.

3.2 Distributed BDD Package

Our distributed BDD package is designed for a network of workstations (NOW). BDDs for small systems are not usually large and they can be easily manipulated on a single machine. However, the number of BDD nodes representing a system can grow exponentially as the system gets larger and it becomes impossible to handle these BDDs on a single machine. Moreover, the time taken to complete BDD manipulation also increases as the system gets larger. The main goal of this project is to avoid memory problems that can arise during BDD manipulation while also speeding up the computation by making use of the collective memory resources available on a NOW. The NOW which is usually an existing infrastructure consists of a number of workstations interconnected via a local area network. Because communication between workstations can be slow, access to the network is avoided as much as possible in the implementation.

The standard BDD manipulation algorithm (non-distributed) shown in Figure 2.4 leads to a large number of network accesses during BDD manipulation since we have to access the memory of another workstation in order to manipulate a BDD node. The effect of this is that workstations on the NOW will at some point have to wait for data from another workstation in order to carry out their own tasks. We need to modify the algorithm so that a workstation can continue other tasks even when some data is needed from other workstations. This cannot

```
01: bfOp(F,G,op) {
02:    if IsTerminalCase(F) or IsTerminalCase(G)
03:    minIndex = minimum variable id of (F,G)
04:    create a REQUEST (F,G) and insert in REQUESTQUEUE[minIndex];
05:    /* Top-down APPLY phase. */
06:    for (i = minIndex; i <= numVars; i++) { bfApply(op,i) }
07:    /* Bottom-up reduce phase */
08:    for (i = numVars; i >= minIndex; i--) { bfReduce(i) }
09:    return REQUEST or the node to which it is forwarded;
10: }

01: bfApply(op,id) {
02:    x = variable with index id
03:    /*Process each request queue*/
04:    while(REQUESTQUEUE[id] not empty) {
05:       REQUEST(F,G) = unprocessed request from REQUESTQUEUE[id]
06:       if (not TerminalCase ((op,F_x,G_x ),result) ) {
07:           nextIndex = minimum index of (F_x,G_x )
08:           result = findOrAdd(F_x,G_x ) in REQUESTQUEUE[nextIndex]
09:       }
10:       REQUEST->THEN = result
11:       if (not TerminalCase((op,F_x',G_x' ),result) ) {
12:           nextIndex = minimum index of (F_x',G_x' )
13:           result = findOrAdd(F_x',G_x' ) in REQUESTQUEUE[nextIndex]
14:       }
15:       REQUEST->ELSE = result
16:    }
17: }

01: bfReduce(id) {
02:   x = variable with index id
03:   while (REQUESTQUEUE[min] not empty) {
04:     REQUEST(F,G) = unprocessed request from REQUESTQUEUE[min]
05:     if (REQUEST->THEN is forwarded to T) { REQUEST->THEN = T }
06:     if (REQUEST->ELSE is forwarded to E) { REQUEST->ELSE = E }
07:     if (REQUEST->THEN == REQUEST->ELSE) { forward REQUEST to REQUEST->THEN }
08:     else if ( (REQUEST->THEN, REQUEST->ELSE) found in UNIQUETABLE[id]) {
09:       forward REQUEST to that node
10:     }
11:     else { insert REQUEST in UNIQUE-TABLE[id] }
12:   }
13: }
```

Figure 3.2: The Breadth-first BDD manipulation algorithm based on [41]

be achieved by using depth-first manipulation.

In order to minimize the number of memory accesses necessary to manipulate BDDs and also allow concurrent execution of threads on the workstations, we use the breadth-first iterative algorithm shown in Figure 3.2. The algorithm performs BDD computations by doing a breadth-first traversal of the operand BDDs rather than the depth-first traversal shown in previous algorithms.

The breadth-first BDD algorithm manipulates BDDs by horizontally grouping the BDD nodes for each input variable together and then manipulating the groups one by one. This technique reduces the random accesses to the memory, thereby improving the performance of the breadth-first technique when compared to the depth-first algorithm.

3.3 Design

In order to achieve our goal of distributing a BDD package, quite a number of decisions have to be made. The major ones include how to distribute BDD nodes among the workstations, how to distribute the computation in order to obtain the best performance of the package, and lastly, how to make sure that no workstation stays idle during BDD manipulation, that is each of the workstations executes some threads of computation concurrently with the others. This is related to the general problem of load balancing [18, 21, 45] in distributed applications. However, in addition to load balancing, another requirement of a distributed BDD application is that the data (BDD nodes stored on each workstation) must also be balanced. Thus, it is necessary to distribute both the BDD nodes that will be stored on the workstations and the operations that will be performed adequately.

3.3.1 Node Distribution

BDD nodes are distributed by assigning each variable to a unique workstation as proposed by Ranjan et al. [37]. The distribution of the variables is done before the construction of BDDs and each workstation is assigned an approximately equal number of BDD variables to prevent overloading any of the workstations.

We note that, performing a large number of network transactions would lead to a poor performance of the distributed BDD package. Thus, a distribution of BDD nodes that requires network transaction when dealing with only one level of a BDD node would be unacceptable. Based on the fact breadth-first technique traverses BDDs on a level-by-level basis, and due to the overhead involved in performing network transactions, the distribution of BDD nodes is done such that all BDD nodes with the same variable number (or a set of consecutive variable numbers) are stored on and handled by the same workstation. This is achieved by distributing BDD nodes to the workstations on a level-by-level basis. A graphical representation of the distribution of the BDD nodes is shown in Figure 3.3. If there are N input variables, the distribution ensures that there are no more than N network accesses in order to reach a terminal node from the root of a BDD.

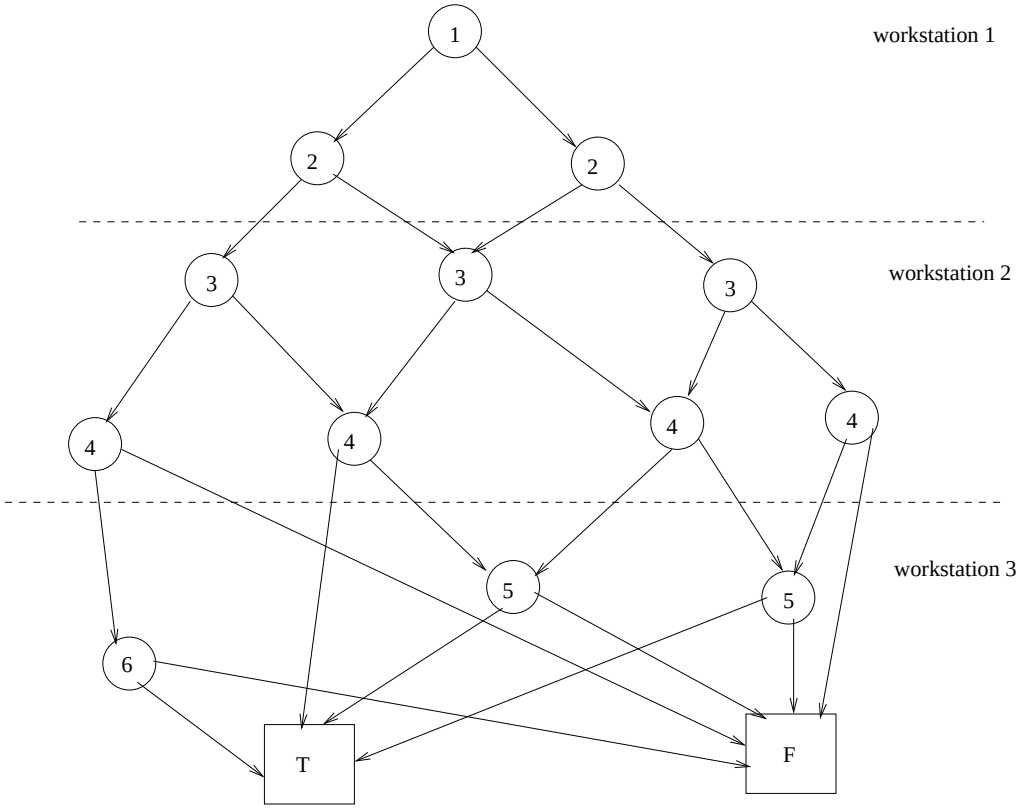


Figure 3.3: Level-by-level distribution of BDD nodes to workstations

In addition, the terminal nodes are stored on all the workstations as constants. Thus, accessing a terminal node (which happens a number of times during BDD manipulation) requires no

network transaction. They are retrieved by accessing the local memory of the workstation on which the manipulation is performed. Also, since there are only two terminal nodes, no significant memory usage is involved.

3.3.2 Generalized Address

In the non-distributed BDD package, each BDD node is uniquely identified by a pointer to memory address. Pointers cannot be used to identify BDD nodes in our distributed package since BDD nodes now reside on different workstations that cannot directly access each other's memory spaces. Moreover, two distinct nodes may reside on two different workstations but may nevertheless be stored at exactly the same address (on different workstations). However, each workstation on the network needs to be able to identify any BDD node regardless of whether the BDD node resides in its local memory or on any other workstation on the network.

We can determine the workstation on which a BDD node resides from its variable number. Thus, we need to be able to retrieve the variable number of a BDD node without actually accessing it since this would involve another network transaction. We therefore form a new address format for BDD nodes. This addressing format called *generalized address* by Ranjan et al. [37] is a tuple `(var_nr,mem_ptr)` consisting of the variable number and the memory address of the BDD node on the workstation where it resides. This address format uniquely identifies each BDD node on the NOW. Given any generalized address, we can determine the workstation on which it is stored by checking the variable number and also access it by checking the memory address pointer associated with it on the workstation on which the node is stored.

3.3.3 Garbage Collection

As discussed in Section 3.1, garbage collection is a memory management technique used to free unused memory in a BDD package. Some of the algorithms that are often used to perform garbage collection include the use of reference counts, the mark-and-sweep algorithm, and the stop-and-copy algorithm [24]. In our non-distributed BDD package, garbage collection is quite easy to implement using the mark-and-sweep algorithm. However, even though the non-distributed package forms the basis for our distributed package, the algorithm will be very

expensive to perform in the distributed BDD package and may also lead to a bad performance of the package. This is because the algorithm will result in a large number of network transactions in order to mark all BDD nodes that are still active and to consequently remove unmarked BDD nodes since a node can be referenced from any of the workstations on which the package is distributed. For similar reasons, other forms of garbage collection are also impractical. Thus, the garbage collection technique is currently not implemented in our distributed BDD package.

3.3.4 BDD Manipulation

As mentioned earlier, BDDs are computed by doing a breadth-first traversal of the operand BDDs. The use of the breadth-first algorithm in Figure 3.2 (adapted to work on a NOW) implies that new processes involving the child nodes of a BDD can be started at the same time. That is, BDD manipulation is done simultaneously on the two different paths of a BDD node. Thus, different processes can be started on the different workstations at the same time. For example, if the BDD node in Figure 3.4 is to be manipulated, a process involving node 1 will be completed by starting two new processes involving nodes 2 and 3 which can either belong to the same or different workstations depending on which workstations they were assigned to.

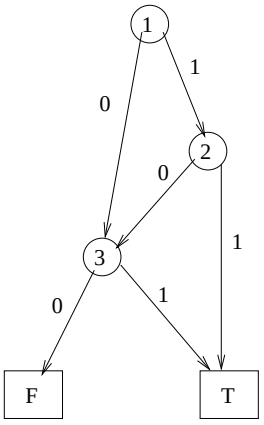


Figure 3.4: BDD Manipulation

For any recursive operation involving two operand BDDs, the process involved is given to the workstation handling the lower variable BDD and the result is stored only on the workstation to which the root variable of the resulting BDD was assigned. Moreover, since BDD nodes are

distributed on a level-by-level basis and nodes with lower variable numbers are closer to the root of the BDD. This implies that from any workstation on the NOW, requests for manipulation of BDDs are always sent to workstations handling higher variable numbers (compared to the variables assigned to the workstation) while the results of requests processed on a workstation are always sent to workstations handling lower variable numbers (compared to the variables assigned to the workstation).

3.4 Implementation

This section describes the implementation details of our distributed BDD package. To implement the transfer of messages between workstations, the distributed package uses the message passing interface (MPI) library. A detailed description of the various messages which can be transfered between the workstations is given in Section 3.4.1. Distributed versions of the various data structures and techniques used to increase the efficiency of the non-distributed BDD package are also implemented in the distributed BDD package. These include implementations of the unique-table and the computed-table. Other data structures implemented include the request queue which does not exist in the non-distributed BDD package.

3.4.1 Data Structures

A description of the various data structures used to aid the implementation of our distributed BDD package and also to improve the efficiency are given below.

Generalized Address Structure

As discussed in Section 3.3.2, a generalized address is a tuple `(var_nr,mem_ptr)` containing the variable number and a pointer to the address in which the BDD node is stored in memory. Given any generalized address v, the memory pointer in v points to a BDD node. The BDD node contains two generalized addresses corresponding to its left and right child nodes, and a pointer called `link` which links the BDD nodes in a hash list. A representation of the generalized address structure is shown in Figure 3.5. The use of generalized addresses provides

a unique identification of each BDD node on a NOW.

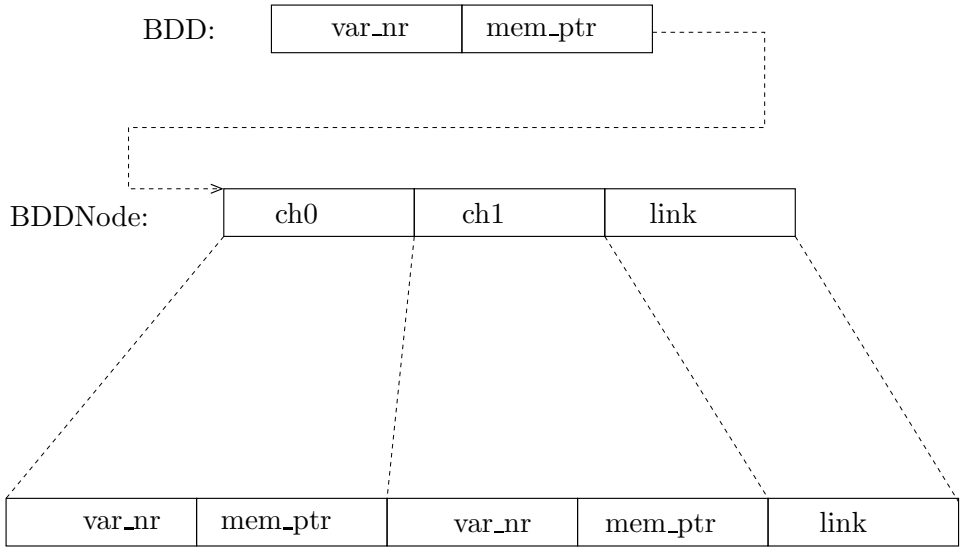


Figure 3.5: A BDD generalized address structure

Unique-Table

The unique-table in our distributed BDD package is similar to the one in the non-distributed package in that it has an entry for each node in the BDD. However, in the distributed package, the unique-table is distributed across the workstations on the NOW. Moreover, instead of having one big hash table for all the BDD nodes, each variable has its own unique-table which resides on the workstation to which the variable was assigned.

An important requirement for the unique-table is that BDD nodes must not be duplicated in the table. This property is maintained by giving each of the workstations the responsibility of adding new BDD nodes with variables that were assigned to the workstation. Before a new node is added, the workstation first confirms that the variable actually belongs to it. BDD nodes with variable numbers that do not belong to the workstation are not stored on such a workstation. This situation should never arise – it would mean that the data structures have been corrupted, forcing the application to terminate immediately. Before insertion, the node is checked to see if it already exists in the hash table for the variable. If the node is found during this lookup operation, it is just returned and not stored again. Otherwise, the new node is

inserted in the hash table. The use of hash lists for the unique-table of each variable helps to maintain a strong canonical representation of BDDs.

Hash tables for the variables assigned to each of the workstations are set up on the workstations during initialization. Section 3.5 gives a full description of the processes involved during a BDD manipulation.

Computed-Table

The distributed computed-table is implemented as a distributed hash-based cache and is used to store intermediate computational results. The use of the computed-table, in the same way as that in the non-distributed BDD package, helps to avoid the repetition of an already completed operation.

The computed-table is set up by specifying different caches for each of the major BDD operations. The user can specify the cache size, which refers to the total number of cache entries that will be stored in all the different caches altogether. During BDD manipulation, new cache entries are added at the beginning of a list (since they are usually more likely to be reused in the next computation than previous entries) provided that the maximum cache size is not yet reached. Once the maximum cache size has been reached, new cache entries are added to the corresponding list by replacing the last entry (which has stayed longest) in the list and the newly added entry becomes the first entry of that list. If the list to which a new cache entry is to be added is empty, we replace the last entry of the first non-empty list found in the computed-table and the newly added entry also becomes the first entry of that list.

The computed-table is implemented on two levels. The first is the local caching in which a workstation only caches operations that it computed itself. That is, the variable number of the lower variable BDD operand belongs to it. The second level of caching is called global caching, and it involves workstations storing intermediate results that were computed and stored on other workstations. That is, it allows a workstation to cache intermediate results with variable numbers that are not necessarily assigned to the workstation. Details about the two levels of caching implemented in the package are discussed in Section 4.1.

Message Structure

The transfer of data between the workstations is implemented using the Message Passing Interface (MPI). There are three types of messages that can be transferred within the distributed BDD package. They are:

1. `BDD_REQUEST` messages,
2. `BDD_ANSWER` messages, and
3. `BDD_QUIT` messages.

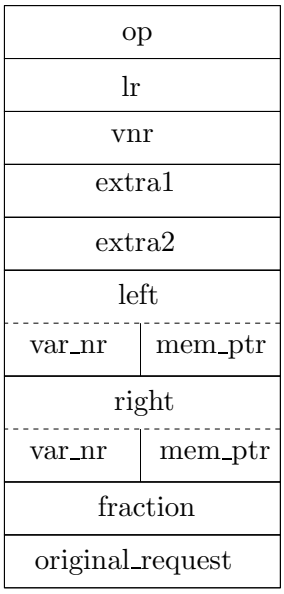


Figure 3.6: Structure for storing requests to be transmitted (OperationData)

A `BDD_REQUEST` message is used to send requests to another workstation on the NOW to carry out an operation on nodes that belong to it. A workstation sends responses to previously received requests to the source of the request with a `BDD_ANSWER` message. Both the `BDD_REQUEST` and the `BDD_ANSWER` messages are sent using the same data structure called an `OperationData`. The third message type (`BDD_QUIT` message) is only sent to all the workstations after all requests have been completed. The message is used to inform the workstations of the successful completion of all BDD manipulations and give them permission to exit the application.

Information about requests sent from one workstation to another are stored using two structures. The first structure, which is shown in Figure 3.6, is an `OperationData` structure and it is part of the request message that is actually transmitted to another workstation. The `op` field in the `OperationData` structure contains the operation to be performed in the request while the `lr` field is used to determine whether a request is a left or right subrequest. The purpose of the `lr` field will become clear shortly. The `vnr` field contains a variable number depending on the operation to be performed. The `extra1`, `extra2`, and `fraction` fields are used to store additional information for specific operations. The `left` and `right` fields contain the operand BDD(s). The `original_request` field contains a pointer to the original request for which this request was generated. The second structure used to store information about requests is called the `RequestData` structure (shown in Figure 3.7). The structure consists of an `opdata` field, which is an `OperationData` structure. In addition, it also contains the `worker` field, which contains the identity of the workstation on which a request is to be completed, the `ans` field which is used to store intermediate result of the request in `opdata`, the `lr` field, which is used to determine whether an intermediate result is an answer to a left or right subrequest and lastly, a `next` field which contains a pointer to the request on the request queue. A `RequestData` structure is kept by the requester during the transfer of requests and referenced by the `OperationData` structure that was actually sent, while the requester waits for the answer to return. This is necessary so that when a workstation sends a request to other workstations, it can continue with other operations. The workstation however needs to store some information about the sent request in order to recognize the result when it returns.

Request Queue

The request queue is used to keep track of work assigned to each workstation. In a non-distributed BDD package using depth-first traversal, a work queue is not explicitly defined, since it uses the call stack. However, in a distributed package, a call may not be local. That is, it might have been sent from another workstation. Moreover, a call on a workstation may also not be completed on the same workstation, thus other work may not be able to start until some have finished.

The request queue facilitates the scheduling of work on each workstation. Each workstation

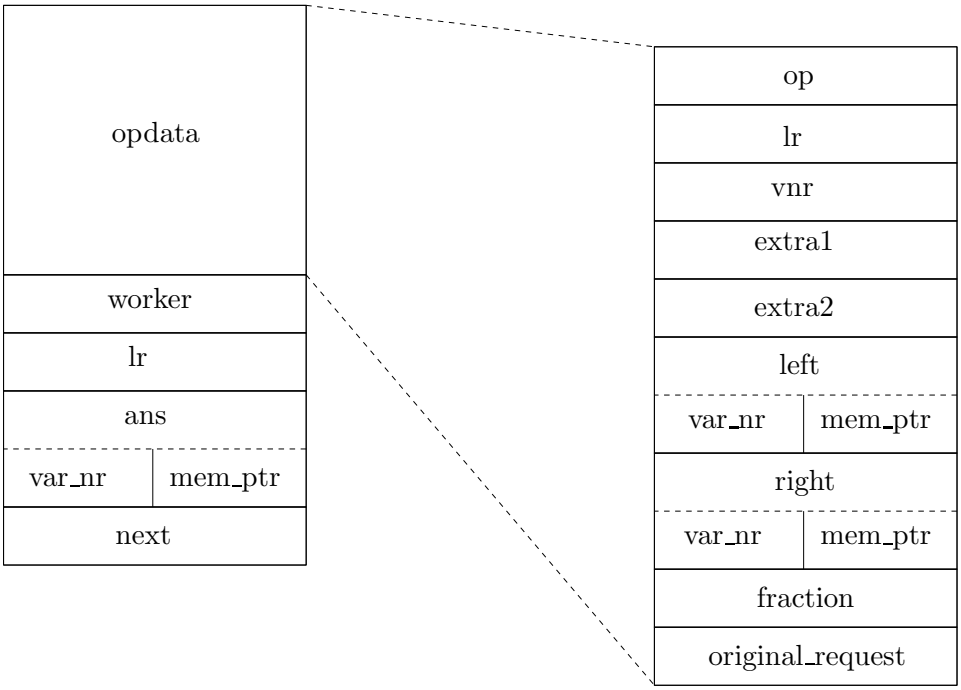


Figure 3.7: Structure for storing sent requests (RequestData)

manages its own queue and jobs are added to the queue as they are received. Both jobs generated locally on a workstation and those received from other workstations are added to the request queue of each workstation. A workstation examines its own queue and completes the work assigned to it in the order of arrival. If a job generates other requests, the requests are added to the appropriate queue whether locally or on other workstations depending on the workstation responsible for the variable number of the original operands.

The purpose of the request queue is to allow a number of different computations to completed at the same time. Each workstation is always busy generating results for one or more computations. The processes involved in the transfer of a request from one workstation to another and how the requests are handled are discussed in Section 3.4.2.

3.4.2 Distributed Computation

Given a BDD node on a particular workstation w_i, it is possible to have its two child nodes on the same workstation or both children on another workstation w_j or even to have one child

node on workstation w_j and another on w_k depending on the variable number of each of the children. This makes it easy to distribute BDD computation by allowing each workstation to manipulate only the BDD nodes with the variable numbers that were assigned to it. In our implementation, computations involving two BDD nodes are processed by the workstation responsible for the node closest to the root in the distribution of BDD variables. Thus, a wide spread of BDD nodes across the workstations will cause the children to frequently point to nodes on other workstations such that all the processors will always have some work to do.

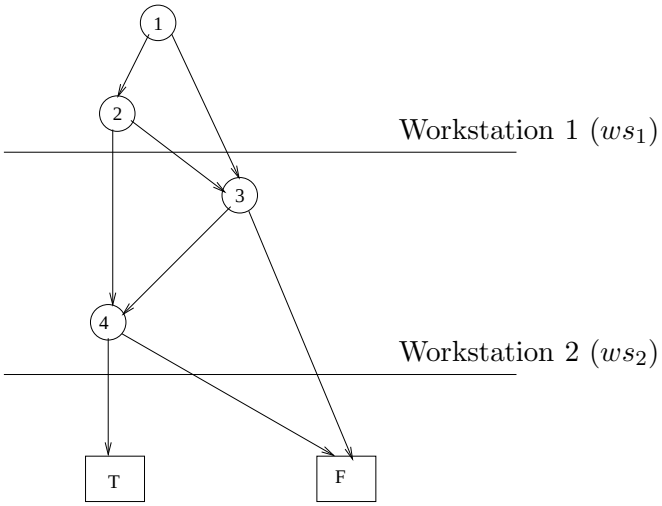


Figure 3.8: BDD to be manipulated

For each operation, a workstation determines whether the work can be done locally (that is, the variable number belongs to it), or if it has to transfer the request to another workstation. If the operation is non-local, the workstation sends a request message with all the necessary data to the appropriate workstation and carries on with the next job on its queue. Some information about the sent request is however stored by the workstation so that the original request can be completed when the results from both child nodes are available. The information stored about the sent request is the `RequestData` structure mentioned earlier (in Section 3.4.1). The structure makes it possible to recognize to which operation an answer applies when that answer is received on a workstation.

During BDD computation, each workstation keeps track of which operations have been started but not yet completed. It also keeps track of whether the answer to a subrequest previously sent

corresponds to the left or the right child node. This avoids starting any operation more than once during the computation. When one of two expected results is obtained, the workstation waits for the second result while carrying on with other jobs on its queue. Once the second result is available, it is combined with the first result and the answer to the original request is sent to the workstation from which the request was received.

For example, Figure 3.8 shows a BDD distributed over two workstations ws_1 and ws_2. The list of variables assigned to each workstation, nodes added to the queue for processing, and the requests that are forwarded are shown in Table 3.1. In this example, in order to process node 1, one request (to process node 2) is added locally to the queue of ws_1 while another request (to process node 3) is transfered to ws_2. The request to process node 2 generates two new requests (for nodes 3 and 4) which are added to the queue of ws_2. The manipulation of node 3 on ws_2 also generates a request which is added to its queue. Each of the workstations completes its tasks and sends the results back to the workstations from which the requests were received. The answers received are combined appropriately since the necessary information to identify the answers had been stored before sending each request away.

Workstation	Variable Assigned	Request to Queue	Request Forwarded
1	$1, 2$	Node $1, 2$ from ws_1	Node $3, 4$ to ws 2
2	$3, 4$	Node $3, 4$ from ws_1 Node 4 from ws_2	none

Table 3.1: The Queue and forwarded requests involved in BDD manipulation

3.4.3 Communication

The distributed BDD package is implemented using the C programming language and the message passing interface (MPI) library. One of the reasons for using MPI is its facility for creating derived data types [34] that describes data as a single entity. The different kinds of messages that can be transfered from one workstation to another are discussed in Section 3.4.1.

During BDD manipulation, MPI ranks the workstations to be used for manipulation from 0 to $n-1$ where n is the number of workstations. The workstation with rank 0 is designated as the "master" while all other workstation are "workers". The master does not handle any BDD

nodes and no variables are assigned to it. However, it causes operation routines to be invoked and the chain of requests generated are completed on the workers. In addition, the master also sends a quit message to all the workers after all operation routines have been successfully completed. The quit message causes the workers to exit and terminate successfully.

A worker continuously waits for all kinds of messages that might be sent to it to arrive. Each message is dealt with according to which type of message it is. When a message to process a request is received, the workstation reads the content of the message and stores the work in its queue for processing. Due to the manner in which BDD nodes are distributed and manipulated on the NOW, messages which are requests are always sent to workstations handling higher BDD variables while messages which are answers are always sent to workstations handling lower variables. Every worker sends a message for each job that does not belong to it. The global cache discussed in Section 4.1.2 is used to reduce the number of jobs sent to other workstations by allowing the worker to make an attempt to get the answer to the request before sending the request. Messages which are answers to previously sent requests are stored. If the answer is one of two expected results, a final result is computed when the second result is received. We do not worry about workers trying to send an answer message after a message to quit has been received. This is because in such a case, a worker would have been stuck waiting for the completion of a process and the master would have been unable to send the quit message to the workers. Thus, if the `BDD_QUIT` message was sent, it implies that all processes have been successfully completed.

3.4.4 Implemented BDD Operations

The distributed BDD package is limited by the fact that the user needs to specify in advance how many variables are to be manipulated. However, for most applications, this is not a serious constraint. The number of variables to be manipulated has to be initialized in order to initialize the execution of the distributed BDD package and also to initialize the message passing interface (MPI). The initialization process for MPI involves getting the number of workstations to be used for the BDD manipulation process and ranking these workstation in order to identify the master and the workers. The initialization process of the package involves setting up the necessary data structures on each of the workstations. The process is done by a

routine called `bdd_init`.

Another important but simple routine in the package, called `bdd_shutdown` is used to clear the message passing interface and to instruct the workers to terminate by sending a quit message from the master. Both the routines to perform the initialization process and the routine to terminate the workstations together with all the main routines used to manipulate BDDs are also called by the user from "outside" the package. The routines to perform BDD manipulations are only used by the master who receives all the calls from the "outside" and invokes a chain of requests on the workers. The routines that are presently handled by the implementation together with a description of which BDD operation the routines are used for, are listed below.

- `bdd_t`: Constructs a BDD for the Boolean function,

  $$f(v) \doteq (v = True)^1$$

  where the parameter v is a Boolean variable. In other words, it constructs the BDD node in Figure 3.9.

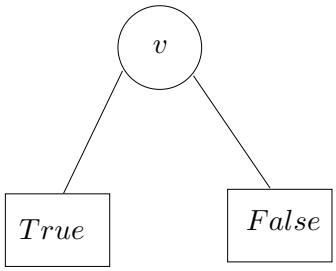


Figure 3.9: BDD node constructed by the bdd_t function

- `bdd_f`: Similar to `bdd_t` except that it constructs a BDD for the Boolean function,

  $$f(v) \doteq (v = False)$$

  The BDD constructed is shown in Figure 3.10.

- `bdd_equal`: Constructs a BDD representing the Boolean function,

  $$f(v_1, v_2) \doteq (v_1 = v_2).$$

[1] $f(v) \doteq (v = True)$ means that the computation, (v = *True*) is done and the result is equal to (or becomes) the function represented as $f(v)$.

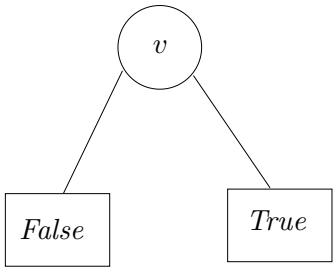


Figure 3.10: BDD node constructed by the bdd_f function

- `bdd_not_equal`: Constructs a BDD for the Boolean function

$$f(v_1, v_2) \doteq (v_1 \neq v_2).$$

- `bdd_neg`: Given a BDD b that represents the Boolean function $f(X)$, this routine computes the Boolean function,

$$g(X) \doteq (f(X) = \mathit{False})$$

  which is the negation of b and returns a BDD representation of $g(X)$. The parameter X ($X = (x_1, x_2, x_3, \ldots, x_n)$) is a vector of Boolean variables.

- `bdd_and`: Constructs the conjunction of two BDD nodes. That is, given a BDD b_1 that represents the Boolean function $f(X)$, and another BDD b_2 that represents the Boolean function $g(X)$, it computes the Boolean function

$$h(X) \doteq (f(X) \wedge g(X)).$$

- `bdd_or`: Given a BDD b_1 that represents the Boolean function $f(X)$, and another BDD b_2 that represents the Boolean function $g(X)$ this routine computes the disjunction of b_1 and b_2. The BDD representing the Boolean function,

$$h(X) \doteq (f(X) \vee g(X))$$

  is returned.

- `bdd_fraction`: This is the only routine that does not return a BDD and does not consider which of the left or right child node has the lower variable number. Given a BDD

representing a Boolean function $f(X)$, it computes the fraction of the possible number of variable assignments that satisfy $f(X)$. In other words, the routine calculates the number of minterms (or satisfying assignments) for a BDD as a fraction of the potential number of minterms.

- `bdd_exists`: Given a BDD b that represents a Boolean function $f(X)$, it computes the existential quantification of f with respect to x, which is given by:

$$g(X) \doteq f(X)|_{x=0} \vee f(X)|_{x=1}.$$

- `bdd_forall`: Given a BDD b that represents a Boolean function $f(X)$, it computes the universal quantification of f with respect to x, which is given by:

$$g(X) \doteq f(X)|_{x=0} \wedge f(X)|_{x=1}.$$

- `bdd_shift`: Given a BDD, it constructs a new BDD with all the variable numbers in the nodes shifted by a given offset. The shifting is done by adding the offset to all the variable numbers in the given BDD.

3.5 Program Execution

This section gives a detailed description of all the processes involved in BDD manipulation using the distributed BDD package implemented.

The command line arguments necessary to use the BDD package includes the number of workstations that the user wishes to use and the name of the file containing the BDD operations required by the user. In reality, a copy of the user program is executed on each of the workstations even though each of them only processes those commands that relate to it. The `bdd_init` function which has to be called first in the user program causes the message passing interface to be initialized and the number of the workstations specified by the user to be stored by MPI. The message passing interface also ranks the workstations by giving them numbers ranging from 0 to $n-1$ where n is the number of workstations specified. The two major BDD nodes *True* and *False* are initialized on all the workers so that each of them can perform operations

on these two nodes. The workstation with rank 0 is set as the master while the others are set as the workers.

The variables are distributed among the workers by assigning an approximately equal number of variables to each worker. No variable is assigned to the master, thus the only BDD nodes it manipulates are *True* and *False*. The number of variables specified by the user when calling `bdd_init` is divided between the $n-1$ workers. In situations where the workers cannot handle an exactly equal number of variables (that is, $(number\_of\_worker \bmod number\_of\_variables) \neq 0$), the workers with ranks 1 to $(number\_of\_worker \bmod number\_of\_variables)$ handle one extra variable compared to the others. Hash tables are allocated on each worker according to the number of variables designated to the worker. The function to process requests `bdd_process_requests` is then called.

Although no variable is assigned to the master, calls to the main BDD routines are handled by the master which like all other workstations also has a copy of the user program running on it. The master generates chains of requests and sends them to the appropriate workers. On the other hand, the workers continuously wait for requests which they have to process. When a BDD manipulation routine is called from the user program, the parameters are stored inside a `OperationData` request structure and sent to the appropriate worker. Each request is sent to the owner with a tag `BDD_QUESTION`. Answers to requests are sent with the tag `BDD_ANSWER`. The tags enable each worker to know which kind of data they are dealing with since both requests and answer are sent using the same message structure (`OperationData`).

When a worker receives a request, if the request has a `BDD_QUESTION` tag, it allocates a new request record either by taking it from the list of free requests (if there are any), or by allocating new memory for it. The list of free requests contains all the previous allocations for requests that have already been successfully processed. This list ensures that the memory used for previous requests is not lost by reusing it and it also reduces memory fragmentation in the memory of the workstations.

Requests from a worker's request queue are serviced depending on which operation the user wishes to perform. In cases where the child nodes of the operand BDD(s) have to be manipulated first, the worker sends a subrequest to the workstation responsible for the variable

numbers involved. If these subrequests still belong to the same worker, they are added to its request queue and the process continues since the worker is constantly waiting for new requests. When sending subrequests, the `lr` field of the request (`OperationData`) structure is modified to reflect whether the request is for the left (0) or right (1) child node. In cases where the answer can be computed without sending subrequests, the answer is stored in an `OperationData` structure and sent with the tag `BDD_ANSWER`. The `lr` field of the message is updated to the `lr` field from the subrequest being processed. This enables a worker to know whether an answer it receives correspond to the left or right subrequest so that they can be combined appropriately. The `original_request` field in an answer message is also updated to the `original_request` field from the question being answered so that a worker receiving the message knows the request to which an answer corresponds.

When a worker gets an `OperationData` structure with a `BDD_ANSWER` tag, the worker checks if an answer has already been received for the same `original_request` field by checking if the variable number of the `answer` in the `original_request` field is -1. If the variable number of the `answer` field is -1, it means that the answer just received is the first answer to the request in question. The `answer` field of the `original_request` structure is then set to the answer just received and the worker continues receiving different data until it receives the second answer to the same `original_request` structure. The two answers are combined depending on the operation that is being performed and sent to the worker from which the original request was received. Once the request has been answered, the request structure containing the request that has just been serviced is added to the free request list so that it can be reused for another request. The final answer to requests is sent to the master. After all requests have been processed, the `bdd_shutdown` function which is the last routine called in the user program is executed by the master. The routine causes the master to send a `BDD_QUIT` message to all the workers, instructing them to terminate. A typical example of a user program using the distributed BDD package is given in Appendix A.

As mentioned earlier, a hash table (the unique-table) is initialized for each variable number at the beginning of BDD manipulation and it is used when making a new node with this variable number, that is when the `bdd_makenode` function is called. Since each worker has certain variable numbers it is responsible for, when a call is made to the `bdd_makenode` function on a

worker, the worker first checks if the variable number indeed belongs to it. Only the owner of a variable number is allowed to make a BDD node with that variable number and each BDD node is stored in the hash table for its variable number. When a new BDD node is created, the hash value of the new BDD is calculated and the hash table of the associated variable number is checked to confirm whether the node has already been created. If it has already been created, the BDD node is returned. Otherwise memory is allocated for the new BDD node and it is created and returned to the caller of the `bdd_makenode` function.

3.5.1 Program Flow Example

To motivate the use of some of the BDD operations implemented in the package and the program flow description above, we discuss a simple typical example of BDD manipulation using the distributed BDD package. An example of a user program that uses the distributed BDD package is included in Appendix A for better understanding of how to use the package. The program flow details of a typical use of the distributed BDD application is shown in Figure 3.11. The chart shows the steps taken in order to compute the conjunction of two BDD nodes A and B where $A =$ `bdd_t`(0) and $B =$ `bdd_t`(1). There are three workstations and the user program is initialized with two variables. Thus the two workers handle one variable each. BDD node A is handled by the first worker and B by the second worker. The labeled arrows on the diagram show the sequence of the operations to execute the user program. The details of each step of this sequence are as follows:

1. The function `bdd_init` is executed on all the workers since they all have to initialize the base BDD nodes ($True$ and $False$), the message passing interface and their internal structures.

2. The master handles the `bdd_and` routine issued from the user, by filling an `OperationData` request structure which is to be sent to the appropriate worker.

3. Both worker 1 and worker 2 are already waiting for requests to process.

4. The master sends the filled `OperationData` request structure to worker 1 which is responsible for the lower variable BDD, that is A.

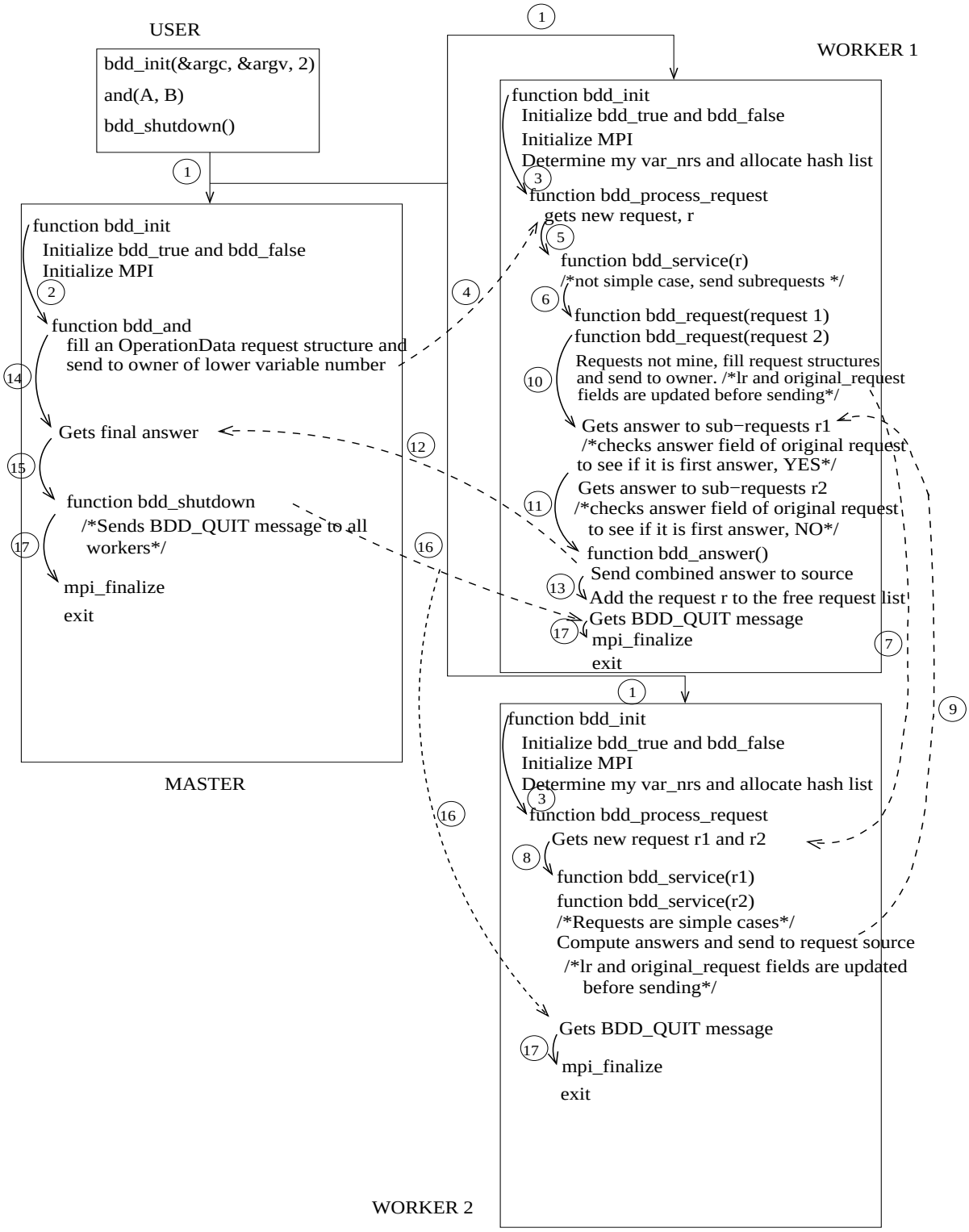


Figure 3.11: Program flow for a typical use of the BDD application

5. Once worker 1 gets the request, it adds the request to its request queue and processes the request by calling the function `bdd_service`. However, worker 1 is unable to compute the final answer because the BDD nodes are not simple cases and the second operand of the `bdd_and` operation (BDD B) belongs to worker 2.

6. Worker 1 generates two subrequests corresponding to the left and right subrequests by calling the function `bdd_request`. The function checks whether the requests belong to worker 1; in this case it does not since the second operand of the subrequests is B. Thus, worker 1 fills two `OperationData` request structures and sends them to worker 2 for processing. The `original_request` field of the subrequests are set as the original request that was sent to worker 1 from the master. The `lr` field of the subrequests are also set to reflect which of them is the left and right request.

7. Worker 2 which is already waiting for requests gets the two subrequests from worker 1 and adds them to its request queue.

8. Worker 2 processes the two new requests on its queue (by calling `bdd_service`). Since the requests contain the simple cases of the `bdd_and` operation, they can be handled by worker 2 independently.

9. Worker 2 computes the answers to the requests and sends them back to the source of the request, that is, worker 1. The `lr` and `original_request` fields of the answer (`OperationData` structure) sent is updated to those obtained from the requests that are being answered.

10. Worker 1 gets the first answer from worker 2 and checks the `answer` field of the answer's `original_request` field to see if it is the first answer (that is, the variable number of the `answer` field is -1). Worker 1 stores the first answer in the `answer` field of the `original_request` field.

11. After getting the second answer of the subrequests from worker 2, worker 1 combines it with the first answer to get the final answer of the `bdd_and` operation by calling the function, `bdd_answer`.

12. Worker 1 sends the final answer of the `bdd_and` operation to the master.

13. The `OperationData` request structure of the request that has just been processed by worker 1 is added to the free request list on worker 1 so that it can be later reused.

14. The master gets the final answer to the `bdd_and` operation on BDD A and B and returns the answer.

15. The master executes the `bdd_shutdown` routine by sending `BDD_QUIT` messages to all the workers.

16. The workers get the `BDD_QUIT` message and stop waiting for new requests to come in.

17. The message passing interface MPI is finalized on all the workstations including the master and they all exit the program.

3.6 Comparison with Previous Work

The distributed BDD package implemented in this thesis is most similar to the work of Ranjan et al. [37]. However, the procedure implemented is not the same as theirs. Some of the major similarities include:

1. The level-by-level distribution of BDD nodes to the workstations on a NOW.

2. The use of generalized addresses to uniquely identify each BDD node represented.

3. Both algorithms use a breadth-first approach for traversing BDD nodes.

There are also some major differences which makes our distributed BDD package unique. Some of these differences include the following:

1. Our distributed BDD package uses the request queues on each workstation to keep track of jobs that need to be completed on each of the workstations instead of the notion of *node forwarding* [37] used by Ranjan et al. This allows us to overcome the problem of shadow request duplication encountered by Ranjan since a request will always be forwarded to the same worker no matter how many times the same request was generated from anywhere in the NOW. Such requests are however not recomputed every time they are to be processed

since the workstation would have cached the result the first time it was computed and will subsequently be returned from the operation cache.

2. We use the message passing interface (MPI) for the transfer of messages between the different workstations on the NOW during BDD manipulation while the parallel virtual machine (PVM) is used in the work of Ranjan.

3. We also introduce two levels of caching in the computed-table which we call local and global caching. Details about these two levels of caching and how they are used to improve the performance of our distributed BDD package are explained in Section 4.1.

4. Finally, we introduce the use of profile shifting to determine how well a particular distribution of variables works on the NOW with regard to the number of messages that are sent and received by each of the workstations. A detailed description of the use of profile shifting and the evaluation of the performance of the BDD package is discussed in Section 4.3.

Chapter 4

Optimization

The previous chapter discussed the implementation details of our distributed BDD package. The package is designed to execute on a network of workstations such that both the BDD nodes and the computations involved in their manipulation are distributed across the set of workstations. The workstations communicate by passing messages to one another using the MPI. Some of the drawbacks that can easily be identified in the implementation include:

1. Each workstation sends a message anytime it has to process a node that does not belong to it and this implies a large number of network transactions.

2. Variables are distributed approximately equally across all the workstations on the NOW without taking into account the number of BDD nodes associated with each variable or any other form of distribution that could make the execution more efficient.

This chapter describes how these drawbacks are handled in the implementation so as to improve the performance of the distributed BDD package.

4.1 Caching

The original attempt made to optimize the performance of the distributed BDD package was the distribution of the BDD nodes and the computation across the workstations. The distribution of

computation however requires communication among the workstations which involves sending messages through the network. The communication overhead associated with the sending of messages can make the package less optimal. To ensure that the performance is improved, the amount of communication required to complete a computation must be reduced as much as possible.

There are different approaches that can be used to reduce the number of requests sent during BDD computation. One simple strategy is to test for base cases before generating subrequests when an operation is to be performed. The test for base cases does not require accessing the contents of the nodes involved thus no network transaction is required. Testing for the base cases is done by comparing the variable numbers. Testing for the equivalence of any other BDD nodes is done by comparing the variable numbers and the memory pointers in the generalized address of the two BDDs involved. During any BDD manipulation process, the test for base cases is done before an operation is recursively performed on the child nodes of the BDD and this prevents unnecessary communication between workstations since the results can always be returned without accessing the other nodes involved.

Another approach to optimizing the implementation by reducing the number of network transactions due to sending of requests is the use of the computed-table. As discussed in Section 3.4.1, the computed-table is a hash-based cache used for storing intermediate computation results and each workstation on the NOW manages its own computed-table.

For any computation thread, there are three situations that can occur with regards to subrequests generated:

1. The two subrequests may be continued on the same workstation; in this case, the same workstation does the next computation step. The subrequests generated will be added to its queue for processing.

2. A subrequest may be processed locally (that is, added to the queue of the workstation) while the second subrequest is processed by another workstation.

3. Both requests may be sent to one or two other workstations; in this case a workstation sends the subrequests and continues with other work on its queue.

As discussed in Section 3.4.2, each workstation takes note of what operation was started on it but is yet to be completed. Intermediate results of BDD computations are stored on two different levels. We refer to these different levels of caching as local caching and global caching.

4.1.1 Local Caching

Local caching is one of the two ways in which intermediate results from BDD computation can be added to the computed-table on a workstation. Each workstation on the NOW is designed to maintain its own computed-table. A workstation records each operation it has performed before by including it in its computed-table. The results of all computation processed from a workstation's queue are added to its computed-table.

Local caching is implemented such that a workstation stores all the results of computations that were started on it. The caching is done regardless of whether the whole computation was handled by the workstation or part of it were handled by other workstations. Once the final result of a computation is available, the operation leading to the result is cached. Answers to subrequests generated from other workstations which were also added to its queue are also cached in the computed-table.

One advantage of local caching is that if a previously performed computation required sending a number of messages to different nodes in order to be performed, these messages are only sent once from a particular workstation, even if the workstation has to perform the same operation again. This is achieved by always checking the computed-table before an operation is recursively performed on the child nodes. This step comes immediately after the test for base cases (in which case the operation can be easily performed without generating subrequest or going through the cache). Since a workstation has a cache of operations that have been started on it before, previously completed operations are not restarted and this helps to reduce the amount of computation and also the amount of communication required during BDD manipulation.

4.1.2 Global Caching

As explained in the last section, local caching is used to cache results for the operations that were started on a particular workstation. However, during BDD manipulation, a subrequest involving BDD nodes which are not assigned to the same workstation may be performed a number of times for different major operations. All these subrequests requires messages to be sent to another workstation. Even though the workstation handling such subrequest would have cached the operation on its computed-table locally, we can reduce the amount of communication required by avoiding the network transaction involved with sending the request in the first place. This is achieved by allowing a workstation to store the results generated from subrequests that were sent to other workstations. This strategy is what we refer to as global caching and it causes workstations to store BDD nodes that are not actually allocated to them in their caches. A typical example that shows some of the advantages of this technique is shown in Figure 4.1.

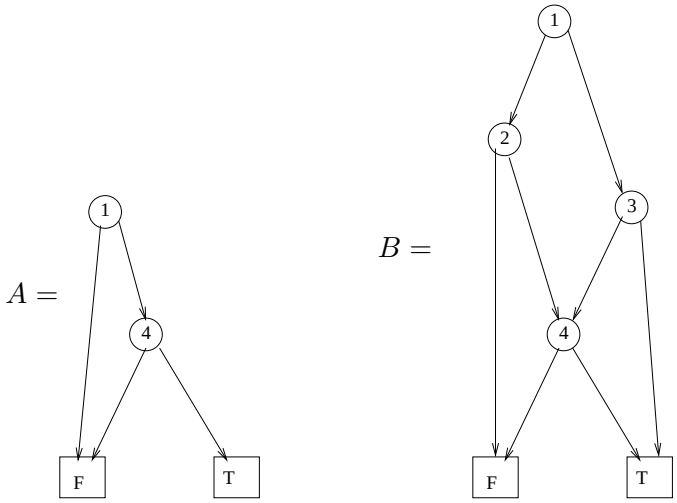


Figure 4.1: Computing the negation of BDD nodes A and B

Assume that variables $1, 2$, and 3 are assigned to workstation 1 and variable 4 is assigned to workstation 2. In order to compute the negation of BDD nodes A and B which are stored on workstation 1, a request to perform the negation of the BDD node C (in Figure 4.2) which is a child node somewhere in both A and B is sent to workstation 2 for each of the negations. The use of local caching will cause the BDD node representing the negation of C to be stored on workstation 2 after it has been done the first time and this result can be returned each time

workstation 1 sends a subrequest to perform the same operation. However, the communication required to send this subrequest to workstation 2 can be avoided by allowing workstation 1 to cache the result of the negation of C.

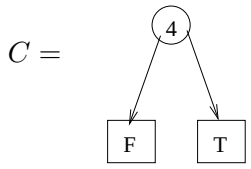


Figure 4.2: BDD node to be negated for each of the computations

Caching of results generated from subrequests are done before the computation of final results for original requests. Since a workstation can easily recognize the subrequest to which an answer it receives corresponds before the answers are actually composed, caching these result does not require much overhead.

Even though we talk of both local and global caching, these two cache levels use the same cache to avoid the overhead of going through two different caches to see if an operation has been earlier performed. The cache is always checked before requests for a recursive operation are generated. Subrequests that have already been cached are simply returned and no communication is required between the workstation on which the subrequests were generated and the workstation that is supposed to process them. This significantly reduces the number of network transactions necessary in the implementation and does not increases the actual computation done on a workstation except for the checking of the cache for results of subrequests generated.

The size of the cache is specified by the user when the distributed BDD package is initialized. The size, which is fixed throughout an execution, is expressed as the number of data entries that may be stored in the cache during BDD manipulation. During the computation of BDDs, if the number of entries specified for the cache is reached, new intermediate results are stored in the computed-table by replacing the first item added to the list or by replacing another entry if the current list is empty. Deleted entries have to be computed one more time in order to be stored again. This is not a major concern since the cache is implemented just as an optimization for the package and it does not affect the completion (or termination) of BDD operations. It is important to note that for different problems, the size of the cache has a

different effect on the performance of the distributed BDD package. But as a general rule, the size of the cache should be big enough to store enough intermediate results and small enough to avoid making use of additional memory that could have been used for computation and storing of BDD nodes.

The caching of results, particularly for global caching, leads to a significant improvement in the performance of the distributed BDD package. Even though the local caching of intermediate results reduces the number of network transaction necessary in BDD computation, the number was reduced to approximately one-tenth when global caching is used. Fewer network transactions imply that the workstations have fewer operations to complete and the computation of BDDs becomes considerably faster, making the application more efficient. Details about experiments conducted to evaluate the use of the global cache are presented in Chapter 5.

4.2 Alternative Distribution of Variables

We originally implemented the distribution of BDD nodes to workstations by assigning an approximately equal number of variables to each workstation for them to handle as explained in Section 3.3.1. However, we noted that a small change in the way variables are distributed across the workstations can cause a significant change in the performance of the application.

Using an equal distribution of variables among seven workstations, Figure 4.3 shows a plot of the total number of network transactions required by each of the workstations to complete BDD manipulation (for the `DP7` problem discussed in Section 5.3). As seen in the histogram, even though the variables are equally distributed among the workstations, the number of network transactions carried out by each of the workstations increases as we traverse the BDD from top to bottom. The increase in the number of network transactions can be attributed to the structure of the BDD involved in the manipulation. For many BDDs, the level-by-level distribution of variables with equal variables assigned to each workstation will cause more BDD nodes to be assigned to some workstations than others because the number of nodes corresponding to each variable is different. In many cases, the number of BDD nodes corresponding to top variables is usually smaller than the number of nodes corresponding to variables midway (or at some lower part) of the BDD. Thus, using an equal distribution of variables may not result

in a good performance of the package for some problems due to the unbalanced distribution of loads on the workstations. We therefore implemented a technique which allows the user to specify how the variables should be distributed to the workstations. This technique is however an optional way of distribution and is most useful when the user has an idea of how the BDD to be manipulated or generated looks.

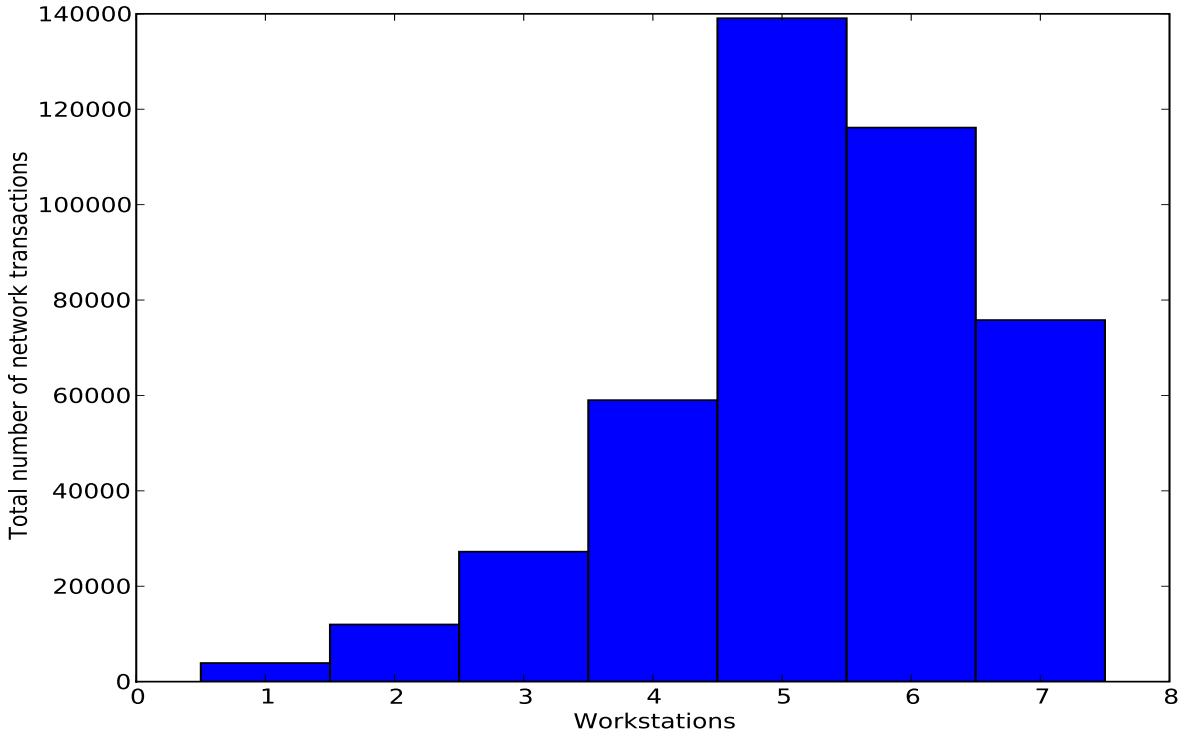


Figure 4.3: Total network number transactions by each workstation

To use this technique, the user has to choose the option to specify the distribution by themselves. The distribution is done by specifying what percentage of the variables are handled by each workstation. For example, if there are n workers to be used, the percentage of the variables assigned to each of the first $n-1$ workers is specified. The remaining variables are then assigned to the last worker. During execution, BDD variables are assigned to the different workstations as the percentages are read from the list provided by the user. For example, to distribute a number of variables over seven workstations using the percentage list $(10, 20, 20, 20, 10, 10)$ specified by the user, the first 10% of the variables (according to the ordering) are assigned to workstation 1, the next 20% of the variables are assigned to workstation 2, then the next 20%

are assigned to workstation 3, and so on.

There are three interesting ways in which a user may choose to distribute variables. The first option is to assign fewer variables at the top of the BDD to a workstation and more variables at the lower part of the BDD to other workstations. An example of this is the percentage distribution $(5, 5, 10, 10, 20, 20)$. The second option is the converse of the first, that is, more variables at the top are assigned to a workstation while fewer variables at the lower part of the BDD are assigned to other workstations; for example using the percentage distribution, $(30, 20, 20, 10, 10, 5)$. A third option is to distribute the variables based on the skewness of the graph obtained when the number of transactions carried out on each workstations for the equal distribution of variables is plotted. For example, if we consider Figure 4.3, we may choose the distribution $(25, 20, 15, 10, 5, 10)$ so that workstations with a large number of network transactions are assigned fewer variables than before and those with little network transactions are assigned more variables in order to make the amount of network transactions on all the workstations within the same range. These different distribution of variables will lead to different behaviors of the distributed BDD package. As mentioned earlier, if we consider the structure of many BDDs, the number of nodes associated with variables closer to the root node are usually smaller than those of nodes associated to variables more closer to the terminal nodes. Thus, we expect that the first option will not yield a good distribution of computation because of an unbalanced load (since fewer nodes are assigned to some workstations while others handle a large number of BDD nodes). The last two options are expected to yield better performance of the package depending on the problem being handled since more BDD variables at the top of the BDD and fewer BDD variables at the lower part of BDD are assigned to different workstations thus creating a level of load balancing on the workstations since the number of BDD nodes handled by each workstation is much closer. These predictions are checked in Section 5.3.

If the distribution that yields the optimal performance of the package can be found for simple cases of a problem, then the optimal distribution of the variables for larger cases of the same problem can generally be predicted. Although the user has no control over the number of network transactions, we expect the number of network transactions carried out on all the workstations to be within a close range for the optimal distribution of variables.

4.3 Measurement of Performance with Profile Shifts

As explained in the previous section, users are allowed to specify how variables are to be distributed among the workstations. An improper distribution of the variables on the NOW will generally cause some of the workstations to be underutilized, thus losing potential computation power while others will be overloaded. In addition, it will also increase the amount of communication necessary between workstations. To optimize our BDD package, we want to avoid these two situations as much as possible. This requires that we find some way of evaluating how well the package performs for any particular distribution of the variables across the workstations. Thus, we introduce the concept of profile shifting. For each execution completed using the distributed BDD package, a profile of the number of messages sent and received by each of the workstations to and from other workstations is obtained. The measurement of the number of network transactions is a better way of comparing the execution of two different problems other than the measurement of time. This is because time taken to complete an execution is usually dependent on several factors including the system architecture of the workstations, the network, and the order in which messages are received from the network among other things. On the other hand, the number of network transactions will always remain almost the same at any time, even when different systems or platforms are used. Thus, the use of profiles is a good way of evaluating the performance of the package for any problem. The difference between two profiles generated for the same problem using different data sets (for example, different cache sizes, different variable distributions, or different levels of caching) which we refer to as profile shifts can be used to determine which data set yields a better performance of the package for any problem. Another benefit of measuring the number of network transactions involved in a BDD computation is that it is an indicator of how much time the computation may take, since network transactions take some time to be completed.

An abstract representation of the profile obtainable when executing BDD operations on a NOW with for example 4 workstations is shown in Table 4.1. The profile shows the total number of messages exchanged between the workstations. The entries (i, j) denotes the number of request messages sent from workstation i to workstation j and the entry (j, i) denotes the number of response messages received by workstation i from workstation j where i goes through the rows of the profile and j goes through the columns with i and j between 1 and the number of

WS	1	2	3	4	Total sent	Total received	Total communication
1	$-$	a	b	c	$a+b+c$	0	$a+b+c$
2	a	$-$	d	e	$d+e$	a	$a+d+e$
3	b	d	$-$	f	f	$b+d$	$b+d+f$
4	c	e	f	$-$	0	$c+e+f$	$c+e+f$

Table 4.1: Profile generated from BDD manipulation

workers on the NOW used. For example, in Table 4.1, the number of request messages sent from workstation 1 to workstations $2, 3$, and 4 are a, b, and c, respectively. The number of requests sent from workstation 2 to workstations 3 and 4 are d and e, respectively while the number of requests sent from workstation 3 to workstation 4 is f. As mentioned earlier, due to the level-by-level distribution of consecutive variables to the workstations, requests from a workstation are always sent to workstations handling higher variable numbers compared to the ones assigned to the workstation. As expected, the number of responses received by workstation 1 from workstations $2, 3$, and 4 are a, b and c, respectively, which are the same as the numbers of requests that were sent to them from workstation 1. The same behaviour is also true for all other workstations. We note that a workstation does not require network transaction to process the nodes that belongs to it, thus no network accesses is involved with subrequests generated for the same workstation to handle because they are simply added to its queue. The last column in Table 4.1 shows the total number of network transactions that were completed on each workstation on the NOW.

Since BDD nodes can be distributed in various ways by the user as discussed earlier, each different distribution of variables across the workstations yields different profiles. The shifts (or differences) in the profiles generated, can be used to determined the best variable distribution for any particular problem. Obviously, a distribution that leads to more network transaction is not optimal because it will increase the computation time and may also cause some workstations to do more work than others. Examples of how profile shifting can be used to draw conclusions about the distribution that leads to the optimal performance of the distributed BDD package are discussed in Section 5.3.1 of the thesis.

Chapter 5

Experiments

In the previous two chapters, the different design decisions which were taken to implement and optimize the performance of the distributed BDD package implemented in this thesis were discussed. This chapter investigates the effect of the various design decisions and optimizations. In particular, it addresses the following questions:

- How efficient is the performance of the distributed BDD package in comparison with the sequential BDD package?
- How does the number of workstations in a NOW affect the performance of the distributed BDD package?
- What is the relationship between the time and memory consumption?
- How does the alternative distribution of variables by the user affect the performance of the package? And what is its effect on time and memory requirements?
- How does the use of caches affect the performance? What is the benefit of the two levels of caching implemented in the package?
- How do different cache sizes affect the performance? And what is the effect of different cache sizes on different network topologies?

A summary of the results is highlighted at the end of the chapter.

The experiments were conducted using the high performance computing cluster at the University of Stellenbosch. The cluster consists of 21 computing hosts each with 8 CPUs and 336GB of disk space and about 16GB of main memory. The computing hosts are composed of two Intel Quad-core processors each and are interconnected by an Ethernet connection. The message passing interface (MPI) library is used to provide communication between workstations on the cluster. Even though there are more than 160 cores available on the cluster, we were only able to use some of the cores due to the high grid workload. However, it was sufficient to evaluate the performance of the distributed BDD package. In the rest of this chapter, we refer to the cores as workstations. Time measurements for the experiments were made using the standard Unix `getrusage()` system call for evaluating resource usage on a machine during a program execution. The time reported includes both the user time and the system time used during execution. The measurements are given in seconds and are averaged over five runs.

A number of problems were chosen to evaluate the performance of the package since different problems lead to different BDD behaviours. This gives a fairly general view of the package. Some of the problems chosen for this purpose are shown in Table 5.1. Each problem takes one or two parameters thus giving us a wide range of different problems. The source code for the `DP` problem can be found in Appendix A. The general structure of the source code for other problems is similar to that of the `DP` problem. Although most of the results shown in this chapter reflect only the `DP` problem, all the experiments conducted were done using some or all of the other problems shown in Table 5.1. However, since most of the results look similar (that is, they follow the same trend), only the results from the `DP` problem are reported for all the experiments to allow comparison between the different experiments where necessary.

Problem	Description
`DP`n	The dining philosophers problem with n philosophers, $n \geq 5$
`CNTR`mn	A model of m counters of n bits each, $m = 4, n = 3$
`NET`n	A model of a network of n communicating processors and n network slots, $n = 3$
`TREE`n	A model of a tree arbiter with 2^n requester cells, $n = 1, 2$

Table 5.1: Problems selected for evaluating the performance of the distributed BDD package

5.1 BDD Node Generation

As explained in Section 3.4, BDD nodes are generated and stored on the workstations (if they are not already stored) as requests for BDD manipulation are processed. In this section, we observe the effect of the number of completed BDD operations and nodes generated on the time and memory requirements on the workstations when using the distributed BDD package. We compare the result with the sequential BDD implementation (Section 3.1) running on a single machine. Table 5.2 presents a summary of the amount of memory and time required for the dining philosopher problem as the number of philosophers increases. The experiment is completed using five workstations.

Problem	Implementation	Memory(bytes)	Time (sec.)
DP5	Distributed	3784360	0.27
	Sequential (with garbage collection)	1411156	0.03
	Sequential (no garbage collection)	1450568	0.03
DP6	Distributed	19357400	0.60
	Sequential (with garbage collection)	1468836	0.04
	Sequential (no garbage collection)	1526948	0.04
DP7	Distributed	21478792	2.23
	Sequential (with garbage collection)	1540332	0.06
	Sequential (no garbage collection)	1620744	0.06
DP8	Distributed	66891040	5.66
	Sequential (with garbage collection)	1625628	0.07
	Sequential (no garbage collection)	1731940	0.07
DP9	Distributed	162599896	44.66
	Sequential (with garbage collection)	1724724	0.09
	Sequential (no garbage collection)	1860536	0.08
DP10	Distributed	746185128	248.86
	Sequential (with garbage collection)	1837620	0.11
	Sequential (no garbage collection)	2006532	0.10

Table 5.2: Memory and time requirement for distributed and sequential BDD applications

We observe from Table 5.2 that the memory and time requirements of the distributed BDD package as well as that of the sequential BDD package increases as the problem grows larger. However, both the time and memory requirements of the distributed package is significantly more than that of the sequential package. This increase in the time and memory usage is due to

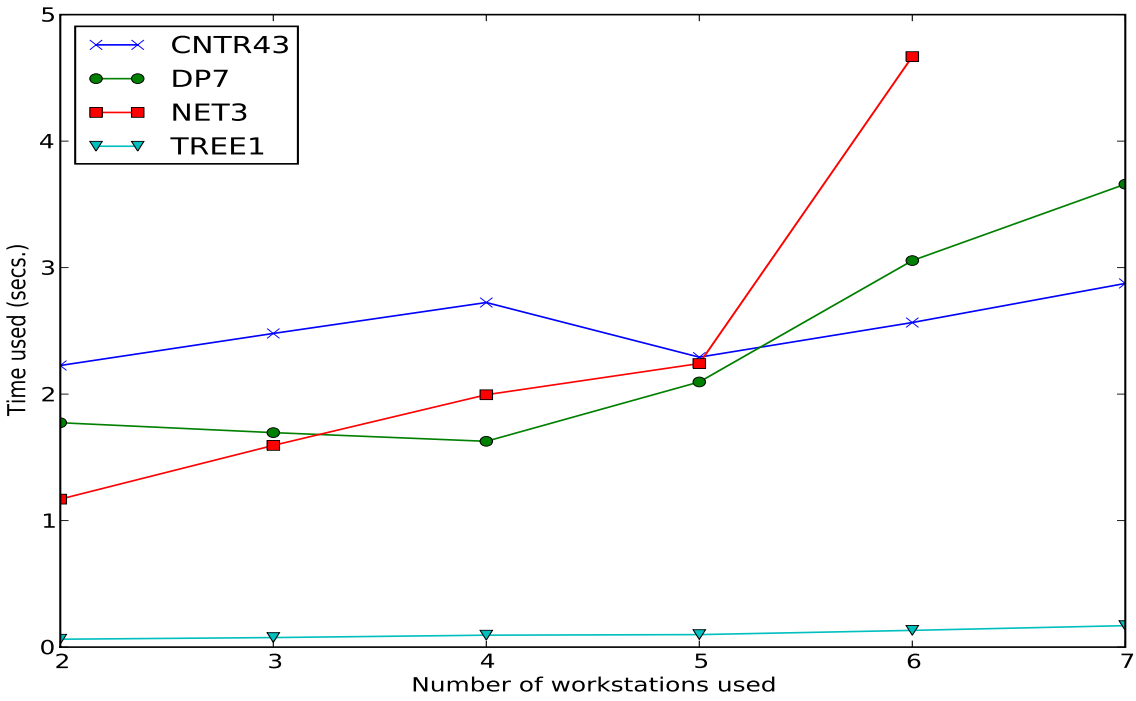
the overhead involved in performing network transactions and also the various data structures used in the distributed package. Another factor contributing to the increased memory requirement is that we did not implement garbage collection for the distributed package whereas it is implemented in the sequential package.

Despite the increase in the resource requirements of the distributed package, one important advantage of the package which is the main purpose of distributing the BDD package is that it uses the collective memory available on the NOW. Thus, even though the sequential package requires less resources, as the problem gets larger, problems which may be lead to insufficient availability of memory when handled with a single machine can be handled using the distributed package. The threshold above which a single machine may be unable to handle a problem is specific to individual problems and it generally depends on the BDD operations carried out in such problem and the memory available on the machine used. Moreover, the memory requirement of the distributed package can also be further reduced by making use of a larger number of workstations to perform BDD manipulation. Further experiments showing the relationship between time and memory requirements and the number of workstations on the NOW when using the distributed BDD package are discussed in Section 5.2.

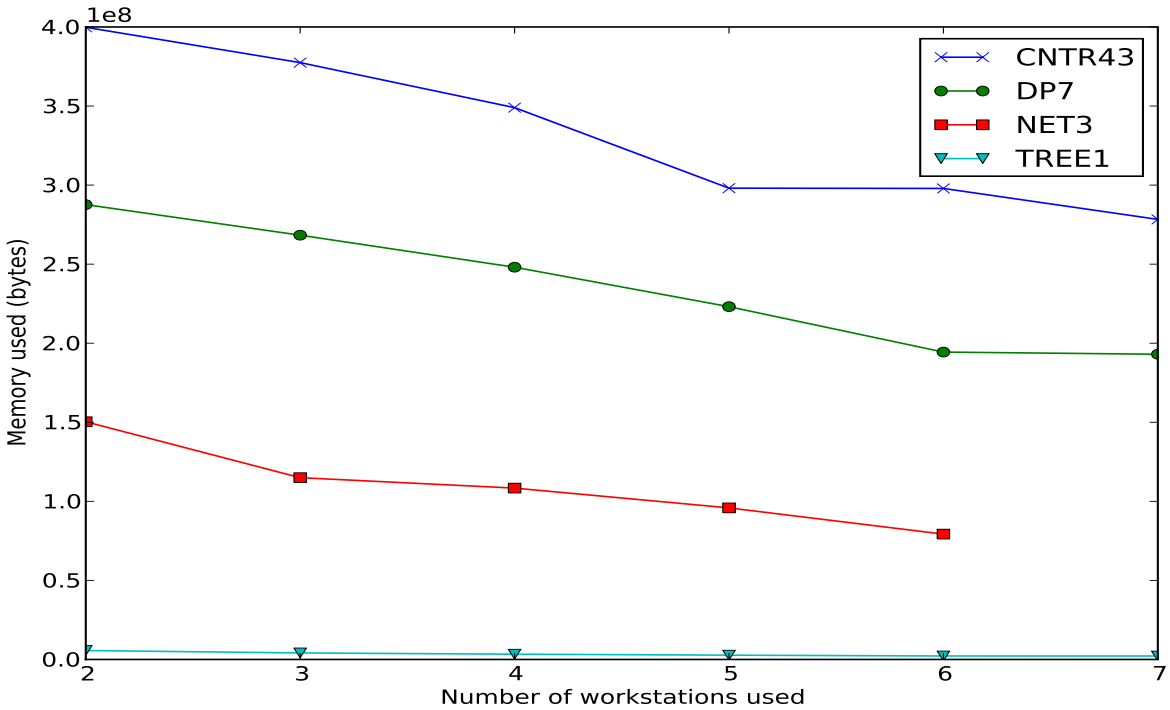
5.2 Time And Memory Requirements

In order to determine the relationship between time and memory requirements in the distributed package, a number of experiments were conducted using different numbers of workstations for the different problems. Figure 5.1(a) and Figure 5.1(b) show the plots of time and memory against the number of workstations used for BDD manipulation for various problems.

From Figure 5.1(a), we can conclude at least within the range of our experiments that the cost of communication outweighs the benefit of multiple processors. That is, the time required to complete BDD manipulation increases as the number of workstations increases. However, the converse is true for the memory requirements as shown in Figure 5.1(b). The relationship between the time and the memory requirement as seen in Figure 5.2 (where n ranges from 5 to 10 workers) is that there is a trade-off between time and memory requirements. That is, when a small number of workstations are used for BDD manipulation, the computation is faster but



(a) Time required for different number of workstations used



(b) Memory required for different number of workstations used

Figure 5.1: Time and memory requirements for different number of workstations

the memory required grows large. On the other hand, increasing the number of workstations used for BDD manipulation causes the computation to require more time and lesser amount of total memory to complete.

This relationship between memory and time can be attributed to the fact that using a small number of workstations leads to fewer communication transactions thus reducing the overhead due to network transactions. However, the purpose of distributing the BDD package will be defeated by using a small number of workstations since the available memory can easily be exhausted as the problem grows larger. On the other hand, using a large number of workstations requires more computation and thus more time. However, we utilize the collective memory available on the network of workstations and this results in a smaller memory requirement on individual workstations as the number of workstations increases.

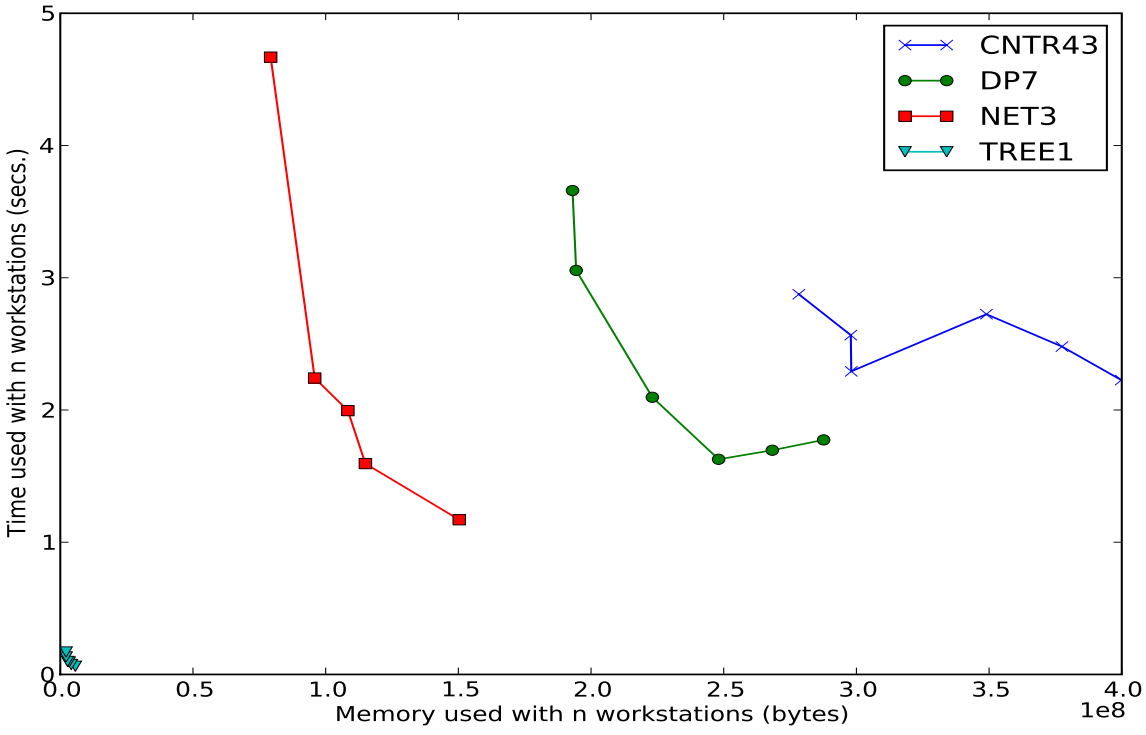


Figure 5.2: Relationship between time and memory requirement

The use of an excessively large number of workstation for a problem that can be handled adequately with fewer workstations will generally lead to very large communication overhead

thus affecting the time required for the computation to complete. For many of the experiments that follow, we shall only report the results for the DP7 problem; however, the results for other models are similar.

5.3 The Effect of Alternative Distribution of Variables

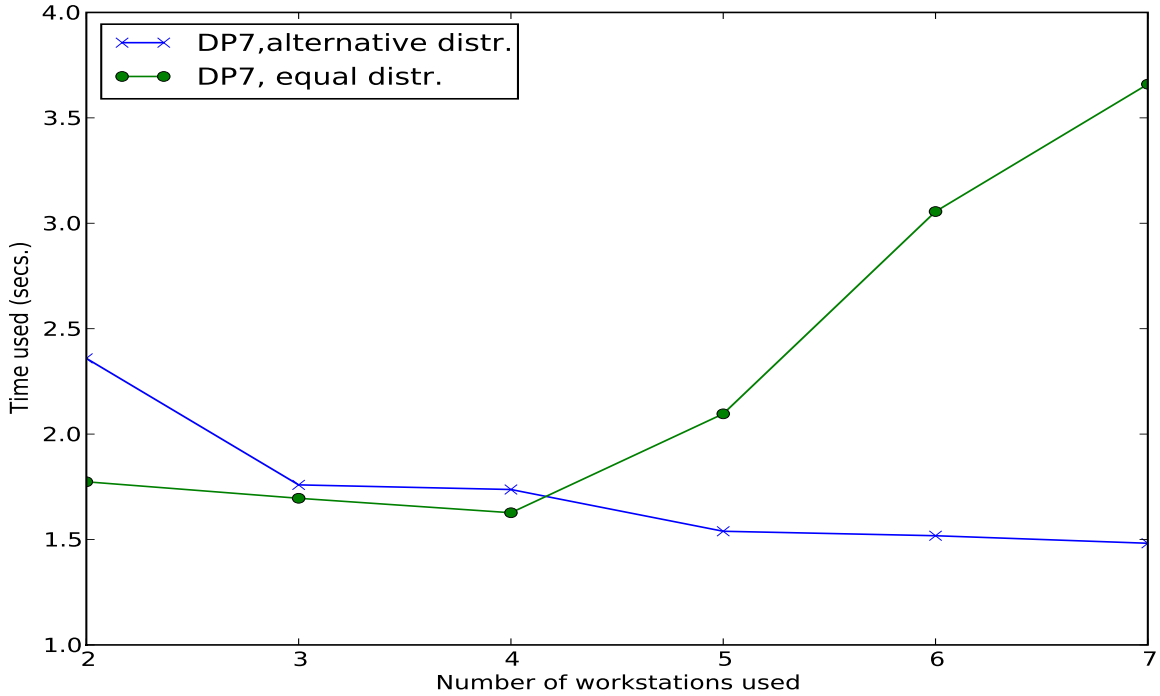
In Section 4.2, we described an alternative way of distributing variables over the workstations by implementing an optional method in which the user can specify the percentage of the total variables that will be assigned to each workstation on the NOW instead of the approximately equal distribution of variables originally implemented in the package. The variables are distributed to the workstations according to percentages read from a list.

Considering the `DP7` problem, we may choose to use any variable distribution as explained in Section 4.2. However, the best variable distribution for the problem involves assigning more BDD variables at the top of the BDD to some workstations and fewer variables at the lower part of the BDD to other workstations. The list of the percentage distributions used for the different number of workstations considered for handling the problem is shown in Table 5.3. The outcome of the experiments for both time and total memory usage (for the different number of workstations) are shown in Figures 5.3(a) and 5.3(b) respectively. The plots also show the time and memory requirements when an equal distribution of variables across the workstations is used.

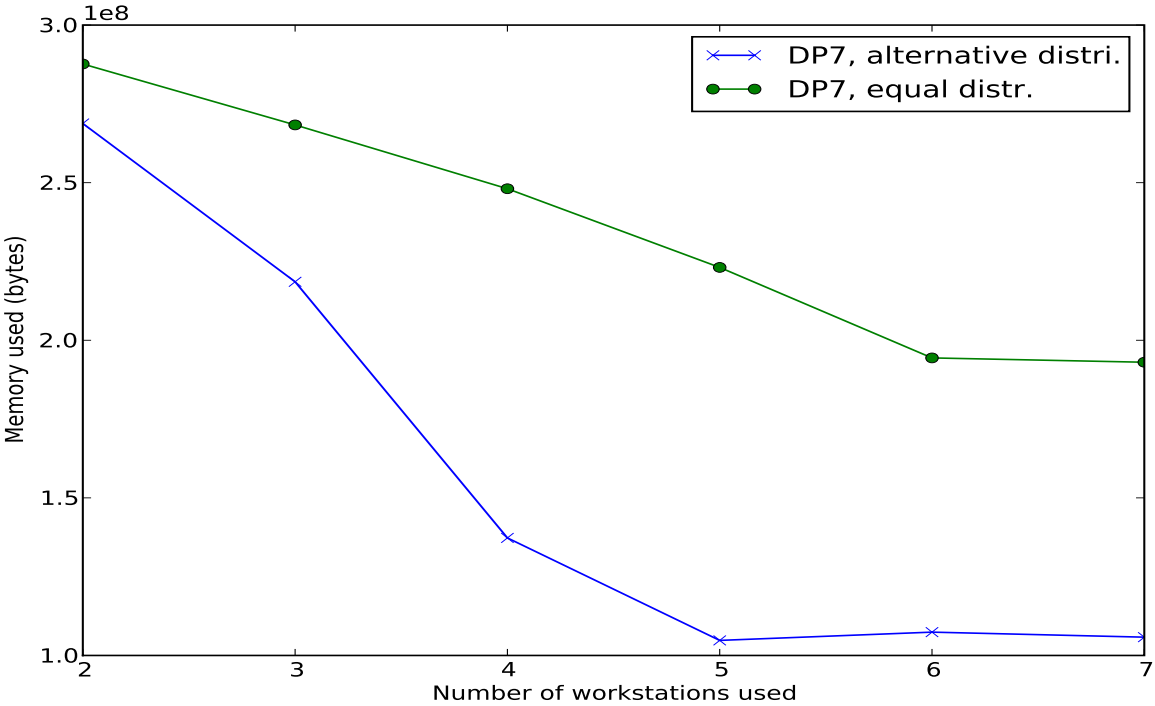
Workstations	Distribution of variables (%)
2	$(70, 30)$
3	$(60, 20, 20)$
4	$(50, 20, 20, 10)$
5	$(40, 30, 10, 10, 10)$
6	$(40, 30, 10, 10, 5, 5)$
7	$(40, 20, 10, 10, 10, 5, 5)$

Table 5.3: Alternative distribution of variables on workstations

As seen in Figure 5.3(a), the time required to complete BDD computation when using the alternative distribution of variables decreases as the number of workstations increases. This



(a) Time requirement for equal and alternative distribution of variables



(b) Memory requirement for equal and alternative distribution of variables

Figure 5.3: Time and memory requirements for equal and alternative distribution of variables

is contrary to what is obtained when variables are distributed approximately equally to the workstations in which case the time required to complete computation increases with the number of workstations used. The same trend of time requirement obtained for the alternative distribution of variables is also obtained for the memory requirements despite the usual relationship between time and memory consumption. Although the memory required also reduces as the number workstations increases for an equal distribution of variables to workstations, we observe that the memory reduces much faster when the alternative distribution of variables is used. Up to a 50% reduction in memory requirement is obtainable on some number of the workstations when an appropriate distribution is used.

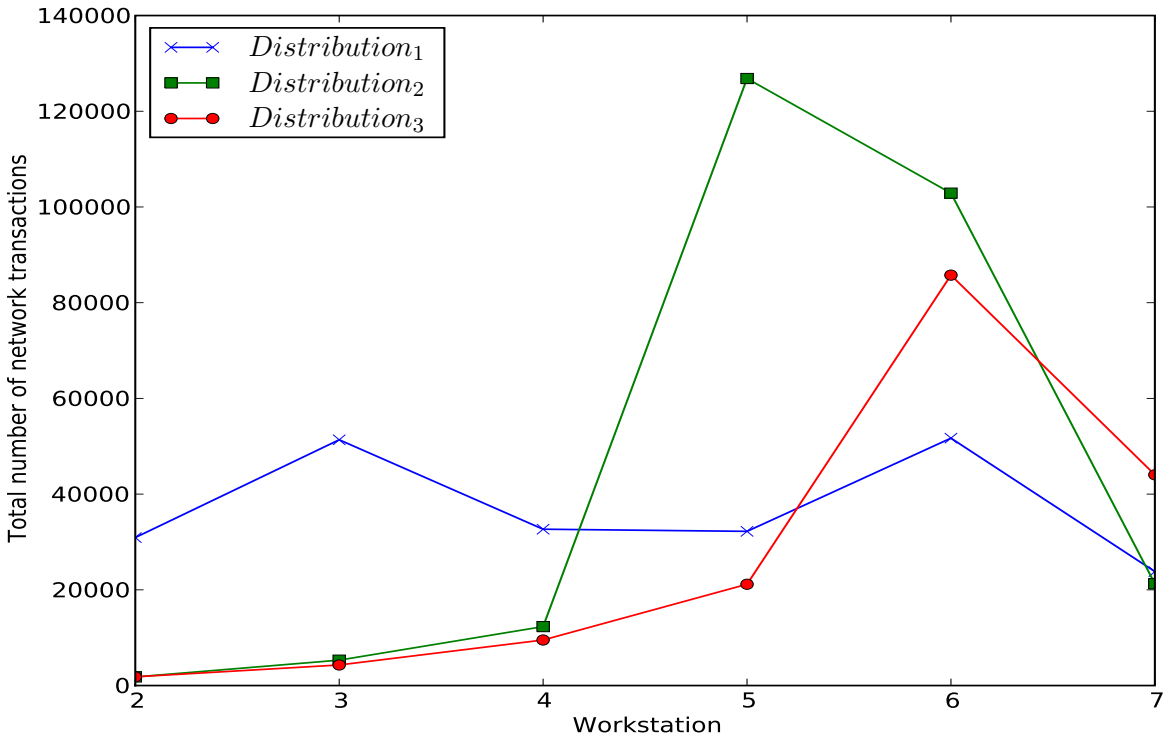


Figure 5.4: Relationship between time and memory requirement

An important conclusion that can be drawn from both Figures 5.3(a) and 5.3(b) is that both time and memory requirements can be reduced to a large extent by finding an alternative way of distributing variables other than the approximately equal distribution of variables across the workstations. It is also important to note that the aim of the alternative distribution of variables is not to make the package similar to the sequential package by shifting many

variables to a single workstation. Even though we assign more variables to a workstation, if the distribution is poor, it affects the performance of the package negatively. For example, if the same distributions of variables to 7 workstations shown in Table 5.3 is used in different orders, say, $distribution_2 = (5, 10, 10, 40, 20, 10, 5)$ and $distribution_3 = (5, 5, 10, 10, 10, 20, 40)$, we observe different performances of the package. Figure 5.4 shows the total number of network transactions performed by each of the workstations with the different distributions. As can be seen from the graph, the alternative distribution $distribution_1 = (40, 20, 10, 10, 10, 5, 5)$ makes the load on each of the workstations to be within a close range, while the distributions $distribution_2$ and $distribution_3$ cause some workstations to have more load than the others. Thus, the alternative distribution of variables must be done in an appropriate order to achieve the best results.

5.3.1 Interpretation of the Profile Shifts

In Section 4.3, we discussed the idea of profile shifting and how it supplies information about all completed executions for which the distributed BDD package is used. A profile showing the total number of messages transfered between each of the workstations on the NOW is associated with every execution.

		Requests messages							Total requests	Total network
	WS	1	2	3	4	5	6	7	sent	trans.
Response messages	1	0	2198	342	342	342	342	338	3904	3904
	2	2198	0	5279	1102	1150	1150	1102	9783	11981
	3	342	5279	0	11717	3288	3308	3312	21625	27246
	4	342	1102	11717	0	27545	8963	9342	45850	59011
	5	342	1150	3288	27545	0	73718	33036	106754	139079
	6	342	1150	3308	8963	73718	0	28672	28672	116153
	7	338	1102	3312	9342	33036	28672	0	0	75802
Total Responses received		3904	9783	21625	45850	106754	28672	0	216588	433176

Table 5.4: Profile generated for DP7 using equal distribution of variables

		Requests messages							Total requests	Total network
	WS	1	2	3	4	5	6	7	sent	trans.
Response messages	1	0	18159	4800	2904	4994	66	0	30923	30923
	2	18159	0	21177	4863	10023	130	0	36193	54352
	3	4800	21177	0	7410	3248	32	0	10690	36667
	4	2904	4863	7410	0	16943	100	0	17043	32220
	5	4994	10023	3248	16943	0	16483	0	16483	51691
	6	66	130	32	100	16483	0	0	0	16811
	7	0	0	0	0	0	0	0	0	0
Total Responses received		30923	36193	10690	17043	16483	0	0	111332	222664

Table 5.5: Profile generated for DP7 using the alternative distribution of variables

As an example of the use of profiles, the profiles generated for the `DP7` problem for both the equal and alternative distribution of variables discussed in Section 5.3 are shown in Tables 5.4 and 5.5, respectively.

From the two profiles, we note that the number of request messages sent from a workstation ws_i to another workstation ws_j is always equal to the number of response messages sent from ws_j to ws_i. That is, each workstation always receives a response for every request it sends to another workstation. In addition, the number of network transactions shown in the profiles explains why the computation of the `DP7` problem with the variable distribution in Table 5.3 completes faster than when the computation is done with an approximately equal number of variables assigned to each workstation. Up to 50% of the network transactions performed when variables were distributed equally is avoided when using the alternative distribution. The overhead involved in performing network transactions is a major drawback to the performance of the distributed BDD package. Although profile shifts can be used to determine which alternative distribution of the variables performs better, we believe that the performance will be greatly increased if network transactions become faster than what is currently available or if the number of network transactions required to complete a computation can be further reduced.

5.4 The Effect of Local and Global Caching

Apart from the distribution of variables to workstations on a NOW, another factor which can affect the number of network transactions and thus the time and memory requirements, is the use of the cache. In Section 4.1 we discussed the two levels of caching (local caching and global caching) implemented in our distributed BDD package. For all the experiments in previous sections (such as that reported in Table 5.4), local caching was used to reduce the run-time. In this section, we evaluate the performance of the cache and how the different levels of caching affect the time and memory requirements in the distributed BDD package.

		Requests messages							Total requests	Total network
	WS	1	2	3	4	5	6	7	sent	trans.
Response messages	1	0	2264	344	344	344	344	341	3981	3981
	2	2264	0	7191	1192	1192	1192	1144	11911	14175
	3	344	7191	0	22215	4132	4132	3940	34419	41954
	4	344	1192	22215	0	65768	14020	13252	93040	116791
	5	344	1192	4132	65768	0	185873	43012	228885	300321
	6	344	1192	4132	14020	185873	0	458073	485073	663634
	7	341	1144	3940	13252	43012	485073	0	0	546762
Total Responses received		3981	11911	34419	93040	228885	485073	0	857309	1687618

Table 5.6: Profile generated for DP7 when no cache is used

In order to evaluate the cache, we use the `DP7` model and consider the number of network transactions, memory requirement and time required when the problem is completed with no caching at all, with the use of local caching only and also with the use of both local and global caching.

The profiles generated for each of the experiments mentioned above are shown in Tables 5.6, 5.7 and 5.8. From the difference in the number of transactions required to complete the experiments, we can already draw conclusions about the performance of the cache. When a BDD computation is performed without any cache at all, the number of network transactions is very large. The same is also true of the time and memory requirements in this situation.

		Requests messages							Total requests sent	Total network trans.
	WS	1	2	3	4	5	6	7		
Response messages	1	0	2237	342	342	342	342	338	3943	3943
	2	2237	0	6494	1150	1150	1142	1102	11038	13275
	3	342	6494	0	14678	2674	2899	2599	22850	29686
	4	342	1150	14678	0	39495	9384	8801	57680	73850
	5	342	1150	2674	39495	0	63153	9977	73130	116791
	6	342	1142	2899	9384	63153	0	110768	110768	187688
	7	338	1102	2599	8801	9977	110768	0	0	133585
Total Responses received		3943	11038	22850	57680	73130	110768	0	279409	558818

Table 5.7: Profile generated for DP7 when local caching is used

		Requests messages							Total requests sent	Total network trans.
	WS	1	2	3	4	5	6	7		
Response messages	1	0	2198	342	342	342	342	338	3904	3904
	2	2198	0	5279	1130	1134	1150	1102	9795	11993
	3	342	5279	0	10817	2908	2950	2919	19594	25215
	4	342	1130	10817	0	26734	10183	8918	45835	58124
	5	342	1134	2908	26734	0	8450	6337	14787	45905
	6	342	1150	2950	10183	8450	0	3651	3651	26726
	7	338	1102	2919	8919	6337	3651	0	0	23266
Total Responses received		3904	9795	19594	45836	14787	3651	0	97567	195133

Table 5.8: Profile generated for DP7 when both local and global caching is used

Cache used	Total network transactions	Memory (bytes)	Percentage memory requirement (%)	Time (sec.)
No cache	1687618	413372416	100	7.39
Local caching only	558818	189370160	46	2.87
Global caching only	311488	18139472	4.5	1.37
Global and local cache	195133	13785752	3.5	0.86

Table 5.9: Summary of computation details

When the computation is done using only local caching, we observe that the number of network transactions required to complete the computation is reduced compared to when no cache is used. Similarly, we observe that the entire computation becomes almost three times faster in this case. A summary of the total number of network transactions, memory requirements and the time taken to complete the `DP7` problem in the different situations are shown in Table 5.9.

The outcome of the last experiment in which case both the local and global caching of operations are used shows a significant increase in the speed at which the computation is completed compared to the other caching options. The computation is more than 8.5 times faster than when no cache is used and up to three times faster than when local caching alone is used. Also, both the number of network transactions and the memory required to complete the computation were significantly reduced, as only 3.5% of the memory required when no cache is used is now required when both caching options are used. As mentioned earlier, Table 5.9 also shows that the time and memory requirements are proportional to the number of network transactions required to complete a computation and this can be associated with the overhead involved in network transactions.

For any problem (as the experiment was carried out for other problems too), the use of caching will definitely allow much larger problems to be handled on a smaller number of workstations since the memory required is reduced. It will increase the speed at which the BDD computations are completed while keeping the number of network transactions small. It is also important to note that when global caching alone is used, the performance of the package is better than when making use of local caching alone. However, the combination of both global and local caching gives the best results. All these benefits achieved from the use of global caching leads to an increase in the overall performance of our distributed BDD package. Although global caching was implemented as an optional technique, we remark that it should always be used when making use of the distributed BDD package.

5.5 The Interaction of Cache Size and Network Topology

Apart from the distribution of BDD nodes and the use of global and local caching, another important factor that affects the performance of our distributed BDD package is the size of the

cache. As mentioned in Section 3.4.1, the size of the cache which is the total cache entries that can be stored in all the caches set up for the different operations is specified by the user before BDD manipulation. In this section, we evaluate the performance of the package in relation to cache size and also discuss other issues that may affect the performance of the cache.

The evaluation of the cache sizes is done using the `DP7` problem and the variables are equally distributed on all the workstations. Also, due to the advantages of using both local and global caching earlier identified, we use both levels of caching in the experiments. The number of requests sent to other workstations is used to evaluate the cache performance since it is an indicator of how much time the computation will take to complete.

One other important factor which we take into account that affects the performance of the cache is the interconnection between the workstations on which the BDD manipulations are done. This is because the speed at which messages are sent between different pairs of workstations depends on the connection between each pair. For example, we expect communication between two workstations on the same computing host to be faster than workstations on two different computing hosts. The difference in the rate at which messages are transfered between different pairs of workstations affects the order in which both request and response messages are transfered on the NOW and also the performance of the cache. We consider three different cases that can be observed on the computing grid and discuss each of them further in the next sections:

- Case 1: Workstations used for BDD manipulation reside on the same computing host. That is, the computing host has more than one processor and each processor is assigned as a workstation on the host. This is similar to using a shared memory multiprocessor for BDD manipulations.

- Case 2: All the workstations used for BDD manipulation are on different computing hosts. This is similar to a situation in which computers with a single processor are interconnected.

- Case 3: Some workstations reside on the same host while others reside alone on their host computer. This situation is similar to having a mixture of single and multiprocessor computers on a network where each processor is used a workstation or a situation in which

the grid engine is allowed to distribute workstations required for processing randomly on the available computing hosts.

5.5.1 Interaction of Cache Size and Network Topology (Case 1)

The DP7 model was analyzed using different cache sizes starting from 10,000 cache entries to 500,000 cache entries. The total number of requests sent between the workstations for the different cache size is shown in Figure 5.5.

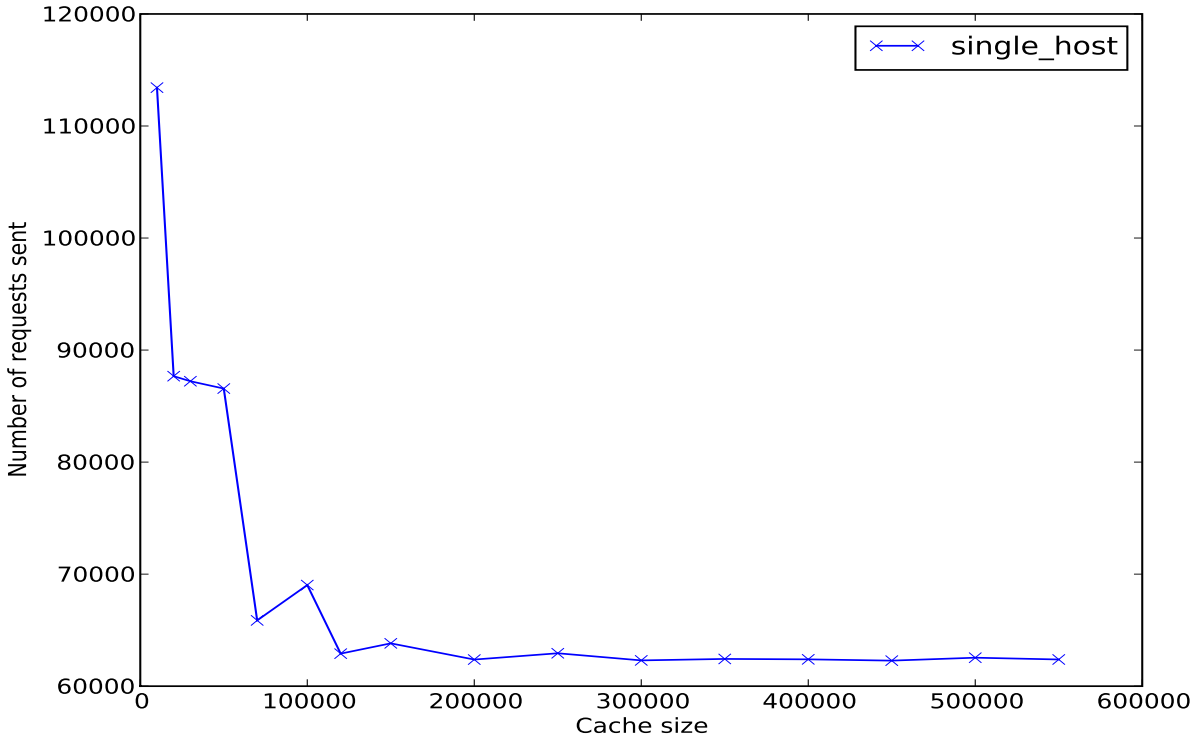


Figure 5.5: Number of requests sent with different cache sizes in Case 1

The results show that when the cache size is smaller than a certain number (which is different for specific problems), more requests are sent between the workstations. This can be attributed to the fact that when the cache size is small, most of the cache entries which might be later used are replaced with other entries before they can be used and thus request for such operations are sent again for it to be re-performed. The large number of network transactions also results in

an increase in the time required to complete BDD manipulation since some requests may have to be performed several times. However, as the cache size increases, the number of requests sent between workstations also reduces because operations which have been previously performed are more likely to be found on the cache when such operations are to be performed again and the results are simply returned instead of sending requests for such operations again. The number of requests sent reduces as the cache size is increased until a certain cache size after which the number of requests remains approximately the same even if the cache size is further increased. This cache size can be seen as the *optimal cache size* for the specific problem. A possible explanation for the constant number of requests sent after the optimal cache size is reached is that most of the operations that are usually done several times are all already cached and are always retrieved from the cache when needed. Another possible explanation may be that all the operations performed during BDD manipulation are cached without ever replacing the cache entries. But this is not the case, as the experiments show that there are cache replacements after the optimal cache size is reached. The cache replacements are also responsible for the little differences in the number of requests sent after the optimal cache size is reached. Thus, the only viable explanation is that most of the operations usually re-performed are all already stored when using the optimal cache size. For example, for the DP7 problem shown in Figure 5.5, the optimal cache size is around 200000. The optimal cache size is however different for specific problems depending on which operations are performed and how often the operations have to be re-performed.

We also discovered that even if the number of workstations on which the problem is handled is slightly increased (e.g., up to 8 workstations instead of the five workstations used for the `DP7` problem), the optimal cache size remains the same. Moreover, the time taken to complete BDD manipulation for the different cache sizes follows the same trend as the number of requests sent across the workstations. It is important that the cache size specified by the user is not unnecessarily larger than the optimal cache size because even though the number of network transaction and the time taken remains the same, the memory usage on the workstations are increased since more memory will be required to store the additional operations that will be cached.

5.5.2 Interaction of Cache Size and Network Topology (Case 2)

The DP7 problem was also analyzed using workstations which are all residing on different computing hosts. In this situation, we expect communication between the workstations to take longer than the previously discussed case in which workstations are on the same computing host. In addition, other factors that may affect the performance of the cache in this case include the order in which request and response messages are received on the different workstations and the speed at which communication between any two workstations is completed. These factors, which can not usually be controlled affect the number of requests sent and received between the workstations. The result obtained from executing the DP7 model on workstations residing on different computing hosts is shown in Figure 5.6.

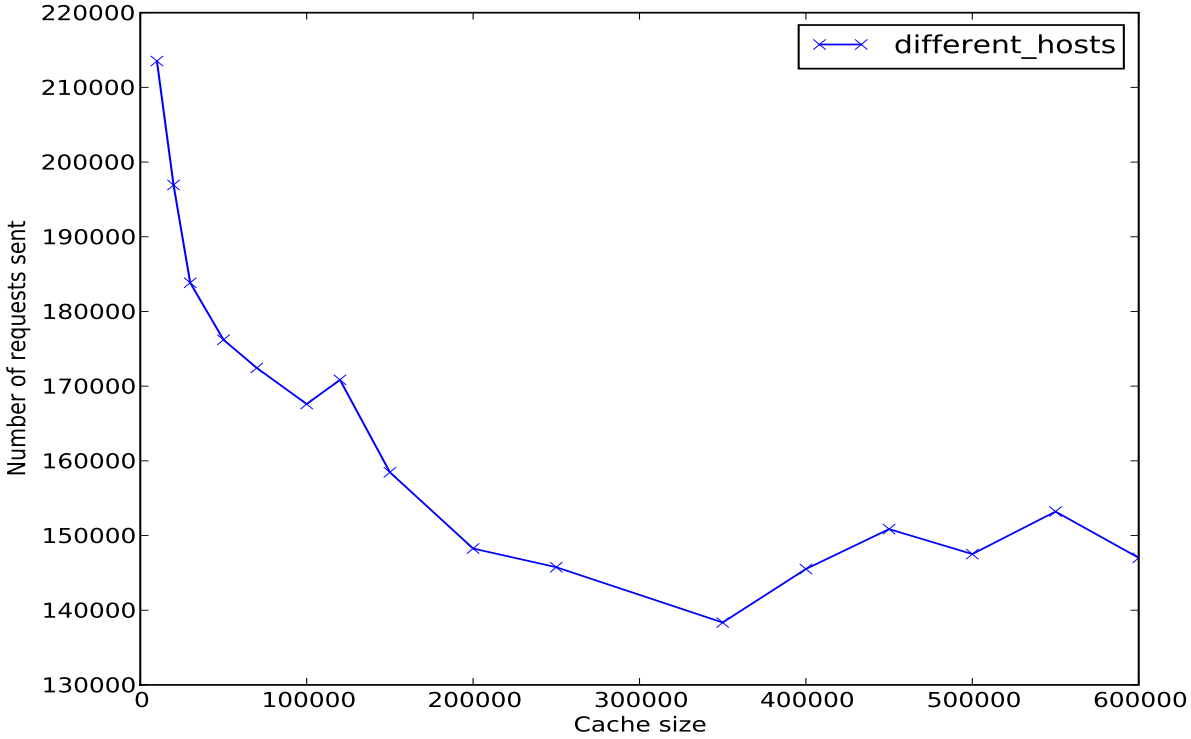


Figure 5.6: Number of requests sent with different cache sizes in Case 2

Similarly to case 1 discussed earlier, the result shows that the number of network transaction reduces as the cache size is increased until the optimal cache size is reached. However, increasing the cache size above the optimal size causes the number of requests to vary within a close range

unlike in case 1 where the number of requests remains almost the same. This variation can be attributed to the order in which messages are received on the different workstations. That is, in case 1, it is most likely that messages are received at their destinations according to the order in which they were sent from different workstations so that once the optimal cache size is reached, the number of requests remain almost the same. On other hand, in case 2 the order may differ depending on the interconnection between any two communicating workstations and this may in turn affect the number of requests sent even for different instances of the same problem. Since the experiment for each cache size examined is conducted individually, the number of requests in each situation varies within a close range depending on how messages were sent on the network in that particular situation. We note that the optimal cache size in this case is also almost the same as in case 1 which is around 200000. The time taken to complete BDD manipulation also follows the same trend as the number of requests sent when the cache size is changed.

However, one major difference between the case in which workstations are on the same host and when they are on different hosts is that in the first case, the total number of requests sent is lesser than those sent in the second case even if the same cache sizes are used. For example, in case 1 discussed above, the number of requests sent when using caches sizes between 10000 and 500000 ranges from 115000 to 60000 requests while in case 2, it ranges between 220000 and 130000. This difference can be attributed to the order in which requests and response messages are received on the workstations which is usually different with every execution of a problem.

5.5.3 Interaction of Cache Size and Network Topology (Case 2)

Another possible interconnection of workstations used for processing is to have a mixture of multiprocessor and single processor machines on the network. The experiment was conducted by allowing the grid engine to assign workstations to available computing hosts randomly. Thus, there are computing hosts on which only one workstation resides while other computing hosts may have two or more workstations running on them. Using the same number of workstations used in the earlier cases for the `DP7` problem, the result obtained in this situation shows that for any particular cache size, the number of requests sent among the workstations can vary between some maximum and minimum values. An average of the possible maximum and

minimum values found for different cache sizes are shown in Figure 5.7.

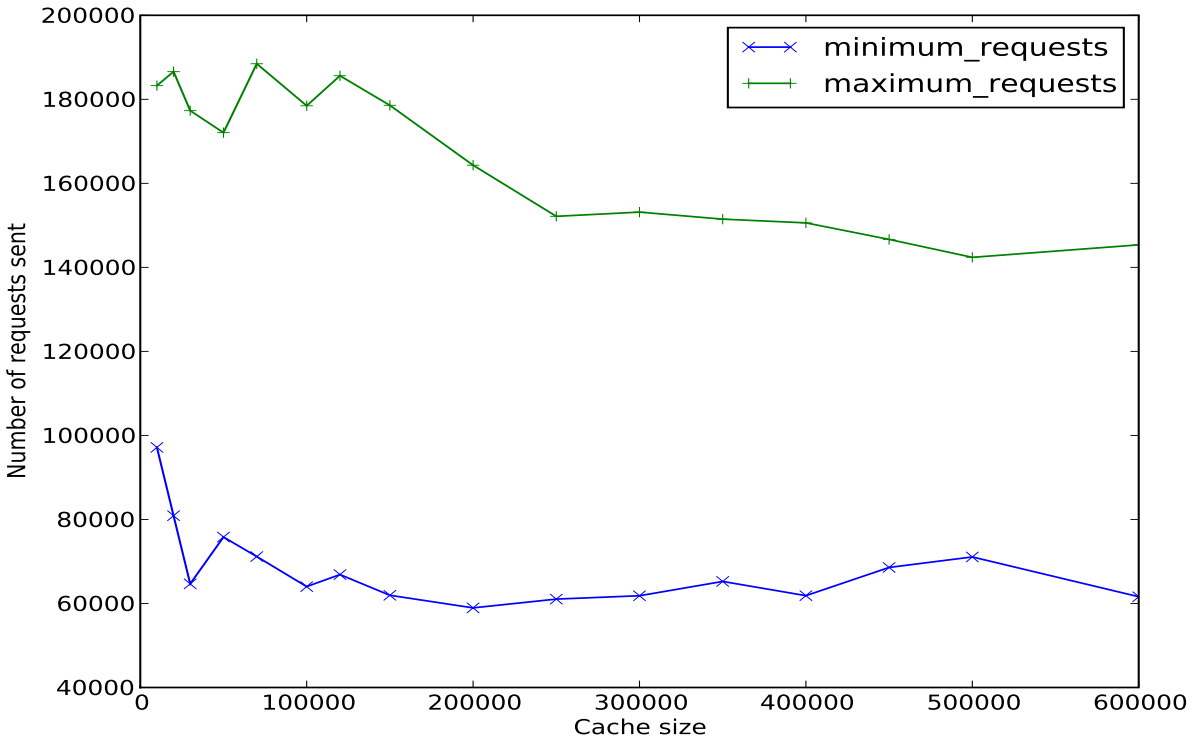


Figure 5.7: Number of requests sent with different cache sizes in Case 3

Some of the reasons responsible for the variations in the number of requests sent during different executions using the same cache size include:

1. The assignment of workstations on available computing hosts, which in turn affects the rate at which communication between two workstations can be completed.

2. The order in which messages meant for the same workstation are received.

For example, we observed that for any particular cache size, a lower number of requests are sent when more workstations are on the same computing hosts while a higher number of requests are sent otherwise. The number of requests sent also increases as the workstations are assigned to more different computing hosts. This result can be seen as a combination of case 1 and case 2 discussed earlier. Thus, the number of requests sent can be anything between the number of requests in case 1 and those in case 2.

In addition, another information obtainable from the result is that even though the number of requests sent can be anything within an interval, as the cache size increases the number of requests sent tends to certain maximum or minimum values which becomes almost stable after the optimal cache size is reached. We note that the time taken to complete BDD manipulation in this case follows the same trend as the number of requests sent. Thus, the time taken also vary between certain maximum and minimum values which depends on the number of requests that are sent.

In summary, considering the three different cases of interconnection of workstations discussed, case 1 results in a better performance of the cache than case 2 and case 3 since it is not affected by issues pertaining to the network. However, the difference between case 1 and case 2 is not so much as the results obtained in both cases show a major similarity (that is, number of requests sent reduces as cache size increases until the optimal cache size is reached). Thus, the number of requests that will be sent when the cache size is increased and the time it will take to complete BDD manipulation can be predicted in both cases. Case 3 shows that the number of requests sent when a particular cache size is used can vary between two different values depending on the connection between the workstations used for manipulation and the order in which messages are received. We observe that the minimum and maximum number of requests that can be sent in case 3 are almost the same as the number of requests sent in case 1 and case 2, respectively. Thus pointing to the fact that case 3 is a combination of both case 1 and case 2. Moreover, we can say that the use of a single computing host for BDD manipulation gives the lower limit of the number of requests that can be sent during BDD manipulation while the upper limit is obtained when each of the workstations are on different computing hosts. Although these two values are relatively close for the `DP7` problem, the difference may likely increase for larger problems. However, for predictability, it is better to use either multiprocessor systems or an interconnection of single processor machines when making use of the distributed BDD package. Lastly, in all the three cases discussed, we note again that the optimal cache size remains approximately the same (200000 for the `DP7` problem) and the number of requests sent remains within a very close range or approximately the same when the cache size is increased beyond the optimal cache size.

5.6 Summary

To summarize the experiments discussed in this chapter, we found the following results:

- The distributed BDD package requires more resources (both time and memory) than the sequential package. However, since it uses the collective memory on a NOW, it has the potential to solve larger problems which can lead to insufficient availability of memory on a single machine.

- When using an equal distribution of BDD variables among the workstations, the time taken to complete BDD manipulations increases as the number of workstations on the NOW increases. This is due to the number of network transactions involved. On the other hand, memory requirement reduces as the number of workstations is increased. Thus, large problems with high memory requirement can be handled by increasing the number of workstations used to handle the problem.

- Also, when BDD variables are equally distributed on the workstations, there is a trade-off between time and memory requirements. That is, the time required for a computation can be reduced by increasing the memory requirement (by reducing the number of workstations) and memory requirements can be reduced by making the computation to take a longer time (by using more workstations).

- An alternative distribution of the BDD variables (selected by the user depending on specific problems) can be used to obtain a better performance of the distributed BDD package. Both time and memory requirements for a problem can be significantly reduced by using a carefully selected distribution of the BDD variables. Moreover, time and memory requirements are proportional in this case.

- The use of the cache improves the performance of the distributed BDD package. When local caching alone is used, time and memory requirements are reduced to 39% and 46% of what is required without caching, respectively. When using the global caching alone, the requirements are reduced to up to 19% and 4.5%, and when both global and local caching are used, the requirements are reduced to 12% and 3.5% of what is required without caching. These figures are the average results from five runs. They may therefore

not be perfectly accurate but they give a good indication of the true behaviour of the system.

- The effect of the cache size on the performance of the package depends on the network topology of the NOW. When workstations are on the same computing host (similar to a distributed shared memory multiprocessor), the number of network transactions reduces as the cache size increases until the optimal cache size is reached after which the number of network transactions remain approximately equal. When the workstations are on different computing hosts (similar to an interconnection of single processor machines), the number of network transactions also reduces until the optimal cache size is reached. However, the number remains within a close range if the cache size is further increased. In a network with mixtures of single processor and multiprocessor machines, the number of network transactions varies between a minimum and a maximum value which are approximately the number of network transactions carried out in the first and second situations mentioned earlier, respectively.

Chapter 6

Conclusion

The purpose of this thesis was to develop a distributed BDD package that uses the collective resources available on a network of workstations in order to avoid the time and memory insufficiency problem usually encountered in the manipulation of BDDs on single machines.

A distributed package for manipulating binary decision diagrams on a network of workstations was developed. The algorithm uses the collective memory available on a network of workstations (NOW) and exploits the breadth-first search technique for parallel computation of BDDs. The message passing interface (MPI) was used to handle the communication between workstations on the NOW. Workstations were utilized both for storing of BDDs and also for parallel BDD computations. Each of the workstations handle its own variables, unique table and caches. Techniques to improve the performance of the distributed BDD package, including global and local caching and the alternative distribution of variables to workstations on a NOW were implemented and evaluated in detail.

The results obtained from the evaluation of the distributed BDD package show that even though the total memory used on all the workstations on a NOW may be more than what is used when a single machine can be used to solve the problem, the package has the potential to handle larger problems. Increasing the number of workstations used to solve a problem also causes a reduction in the memory requirement of each workstation on the NOW and increases the overall performance of the package. The time taken to complete BDD manipulation in the distributed package is generally more than that of the sequential package, primarily because of

network overhead and other data structures used for parallel processing.

We discovered that an alternative distribution of variables to workstations can yield a better performance of the package than an equal distribution of variables. In addition, the use of the two different levels of caching introduced in the thesis improved the performance of the distributed BDD package by reducing the number of network transactions necessary to complete BDD manipulation and thus reducing the communication overhead and the time required.

We were able to test how the package behaves in various network interconnections by making use of special facilities provided on the computing grid used for evaluating the package. The use of a shared memory multiprocessor may yield a better performance of the distributed package than using an interconnection of single processor computers. However, one major advantage of using the NOW is that shared memory multiprocessors are specialized pieces of hardware, whereas NOWs are common and easy to construct. The package can however be easily adapted to work in both situations so that for very large problems where memory available on a shared memory multiprocessor may be insufficient for BDD manipulation, a NOW can always be used to handle BDD manipulations.

Even though the package could not be evaluated against other similar work [37] due to the unavailability of the original source code, we believe that the distributed BDD package discussed in the thesis produced promising results.

The package can be improved in many ways, but the following are likely to produce the greatest benefits:

- Implementation of garbage collection: Garbage collection is currently not implemented in the package and this is part of the reasons why the distributed BDD package currently uses more resources than the sequential version. The implementation of garbage collection in the distributed BDD package is likely to improve the performance of the package. For example, when the specified cache size is reached, garbage collection can be used to free memory that are used by BDD nodes that are no longer needed and this causes such nodes to be deleted from the cache also, thus making it possible to store more nodes without performing a replacement. In addition, accessing nodes in the hash table becomes faster if unneeded nodes can be deleted.

- Dynamic variable re-ordering: The experiments conducted show that the performance of the package can be improved by distributed BDD variables in better ways. Although we have considered different distributions of variables to workstations, the variables are still statically distributed. Various studies on load balancing [18, 21, 45] in a distributed system have shown that statical distribution of variables can lead to an uneven distribution of load in situations where a task can generate other tasks which is true of a BDD operation. Dynamic distribution of load has been suggested as a way of combating this problem. The use of dynamic variable re-ordering which is not currently implemented in the package has also been suggested as a way of getting a good variable order which may improve BDD manipulation process [38]. Implementing the technique will be an interesting improvement of the package.

- Improving the alternative distribution of variables technique: Since the alternative distribution of variables lead to an improved performance of the distributed BDD package, it will be interesting to implement more efficient ways of finding alternative distributions of variables.

- Addition of other features: The package can also be extended by implementing additional Boolean functions in it.

Appendix A

The Distributed BDD package

A.1 Using the BDD distributed package

The user makes use of the distributed BDD package through the command line with at least three command line arguments which include, the name of the user file containing calls to the routines the user would like to perform, the number of workstations the user would like to use and the cache size to be used. The user can also specify the option to distribute BDD variables by themselves in which case a file containing the list of percentages to be used will also be specified. The user file should include the header file `bdd2.h`. The package includes a make file that compiles the user file. The `mpirun` command with all the necessary parameters specified is used to execute the compiled source code.

Note that the first routine that must be called in the user file is the `bdd_init` function which initializes all necessary variables and also the message passing interface (MPI) on all the workstations. The main routines for BDD operations required by the user can then be called. The `bdd_shutdown` routine is called after all the BDD operations required by the user have been called. The routine causes all the workstations to terminate by sending each of them a `BDD_QUIT` message and exiting the package.

A.2 Source code for solving the Dining philosopher problem

The source code for the dining philosopher problem is listed below. Although, other problems are solved using the different BDD operations applicable to them, source codes for the other problems mentioned in the thesis follow the same format.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "bdd2.h"

/*Macros for variable encoding, more readable names for L', L, R', R*/
#define LEFT_P(x) (((x) *4 + 0))
#define LEFT(x) (((x) * 4 + 1) )
#define RIGHT_P(x) (((x) * 4 + 2))
#define RIGHT(x) (((x) * 4 + 3))
#define TOUCH(v,b) { \
    int w; \
    for (w = 0; w < b; w++) touched[v + w * 2] = 1; \
}
#define CLEAR { \
    int v; \
    for (v = 0; v < F; v++) touched[v * 2] = 0; \
    aaa = bdd_true ; \
    SIZE(aaa, F); \
}
#define IS_F(v) {\
    bbb = bdd_f(v);\
    aaa = bdd_and(aaa, bbb); TOUCH(v, 1) ;\
    SIZE(aaa, F); \
}
#define IS_T(v) { \
    bbb = bdd_t(v); aaa=bdd_and(aaa, bbb);  TOUCH(v, 1) ; \
    SIZE(aaa, F); \
}

#define ADD_INIT { \
    I = bdd_or(I, aaa) ; \
    SIZE(I, F); \
}
#define ADD_TRANS { \
    int v; \
    for (v = 0; v < F; v++)  \
        if (touched[v * 2] == 0) { \
            bbb = bdd_equal(v * 2, v * 2 + 1); \
            aaa= bdd_and(aaa, bbb);\
        } \
    R = bdd_or(R, aaa); \
}
#define PEAK(n) { \
```

```
    unsigned long int m; m = bdd_get_nodes(); if (m > peak[n]) peak[n]= m; }
#define LOGTWO (0.693147181)
#define LOG2(x) (log(x) /LOGTWO)
#define POW2(x) (exp(x*LOGTWO) )
#define SIZE(b, n) { \
    double k, f; \
    k = bdd_fraction(b), f = LOG2(k) + (n); \
    if (f < 31) printf("%ld", (long int) (0.5 + POW2(f) ) ); \
    else printf("%g of 2^%ld", k, (long int) (n)); \
}
int N;                /*number of philosophers*/
int F;                /*number of forks*/
int V;                /*number of BDD variables*/

BDD I;                /*the set of initial states*/
BDD R;                /*the transition relation*/
BDD S;                /*the set of reachable states*/
int iter;                /*number of iterations before fix point is reached*/
unsigned long int peak[4];        /*different maxima computed during expansion*/

int *touched;            /*dynamic array: which variables have been altered? */
int main(int argc, char *argv[])
{
    if ((argv[1][0]<48)||(argv[1][0]>57)) {
        /*Read the command line parameter*/
        printf("Usage: %s <nphil>\n", argv[0]);
        exit(-1);
    }
    N = atoi(argv[1]);
    F = 2 * N;
    V = 2 * F;
    touched = (int *) malloc(sizeof(int) * V);
    if (touched == NULL) {
        printf("Couldn't allocate the \"touched\" array!\n");
        exit(-1);
    }
    printf("Nr of philosophers: %d\n", N);
    bdd_init(&argc, &argv, V);

    {    /* Build the initial state */
        BDD aaa, bbb; int i;
        I = bdd_false;
        CLEAR;
        for (i = 0; i < N; i++) {
            IS_F(LEFT_P(i));
            IS_F(RIGHT_P(i));
        }
        ADD_INIT;
    }
    {    /* Build the transition relation */
        BDD aaa, bbb; int i;
        R = bdd_false;
```

```
    printf("transitions:"); fflush(stdout);
    for (i = 0; i < N; i++) {
        printf("%d", i); fflush(stdout);
        /* Build the first transition for philosopher i */
        CLEAR;
        IS_F(LEFT(i));
        IS_F(RIGHT(i));
        IS_F(RIGHT((i + N - 1) % N));
        IS_T(LEFT_P(i));
        ADD_TRANS;
        printf("."); fflush(stdout);
        /* Build the second transition for philosopher i */
        CLEAR;
        IS_T(LEFT(i));
        IS_F(RIGHT(i));
        IS_F(LEFT((i + 1) % N));
        IS_T(RIGHT_P(i));
        ADD_TRANS;
        printf(".");fflush(stdout);
        /* Build the third transition for philosopher i */
        CLEAR;
        IS_T(LEFT(i));
        IS_T(RIGHT(i));
        IS_F(LEFT_P(i));
        ADD_TRANS;
        printf(".");fflush(stdout);
        /* Build the fourth transition for philosopher i */
        CLEAR;
        IS_F(LEFT(i));
        IS_T(RIGHT(i));
        IS_F(RIGHT_P(i));
        ADD_TRANS;
        printf(".");fflush(stdout);
        aaa = bdd_false; bbb = bdd_false;
    } printf("\n");
}
/*Pre-report the result */
printf("|I| == "); SIZE(I, F); printf("\n");
printf("|R| == "); SIZE(R, V); printf("\n");

/* Compute the full state space */
{    BDD uus;
    iter = 0;
    uus = I;
    printf("expansions:"); fflush(stdout);
    do {
        S  = uus;
        uus = bdd_shift(S, 1);
        uus = bdd_and(R, uus);
        uus = bdd_exists(uus, 1, F, 2);
        uus  = bdd_or(uus, S);
        iter++;
    } while(BDD_NE(uus, S));
```

```
        printf("\n");
        printf("Iter: %d\n", iter);

    }
    /* Report the results */
    printf("Non- Restricted number of iteration \n");
    printf("|S| == "); SIZE(S, F); printf("\n");

    bdd_shutdown();
    exit(EXIT_FAILURE);
}
```

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Transaction on Computers*, 27(6):509–516, 1978.

[2] Pranav Ashar and Sharad Malik. Fast functional simulation using branching programs. In *ICCAD*, pages 408–412, 1995.

[3] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 188–191. IEEE Computer Society Press, 1993.

[4] A. Biere. ABCD: an experimental BDD library, 1998. Manual.

[5] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996.

[6] K. S. Brace, R. E. Bryant, and R. L. Rudell. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[7] Daniel Brand. Verification of large synthesized designs. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 534–537. IEEE Computer Society Press, 1993.

[8] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[9] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Survey*, 24(3):293–318, 1992.

[10] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *DAC '91: Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 403–407. ACM, 1991.

[11] 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 Computer-Aided Design of Integrated Circuits and Systems*, 13:401–424, 1993.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51. ACM, 1990.

[14] Hyunwoo Cho, Gary D. Hachtel, and Fabio Somenzi. Redundancy identification/removal and test generation for sequential circuits using implicit state enumeration. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 12(7):935–945, 1993.

[15] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In *IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design, IEEE Computer Society Press*, pages 111–128, November 1989.

[16] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 365–373. Springer-Verlag New York, Inc., 1990.

[17] O. Coudert, J. C. Madre, and H. Touati. TiGeR version 1.0 user guide., December 1993. Manual.

[18] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel Distributed Computing*, 7(2):279–301, 1989.

[19] Rolf Drechsler, Nicole Drechsler, and Wolfgang Günther. Fast exact minimization of BDDs. In *DAC '98: Proceedings of the 35th annual Design Automation Conference*, pages 200–205. ACM, 1998.

[20] A. Geist, A. Beguelin, Jack Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994.

[21] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, 1997.

[22] Andersen R. Henrik. An introduction to binary decision diagrams, April 1998. Lecture notes.

[23] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *DAC '93: Proceedings of the 30th international Design Automation Conference*, pages 266–271. ACM, 1993.

[24] Richard E. Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.

[25] S. Kimura, T. Igaki, and H. Haneda. Parallel binary decision diagram manipulation. In *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, volume E75-A, pages 1255–1262, October 1992.

[26] Y. T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *DAC '92: Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 608–613. IEEE Computer Society Press, 1992.

[27] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[28] Patrick C. McGeer, Kenneth L. McMillan, Alexander Saldanha, Alberto L. Sangiovanni-Vincentelli, and Patrick Scaglia. Fast discrete function evaluation using decision diagrams. In *ICCAD*, pages 402–407, 1995.

[29] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[30] Kim Milvang-Jensen and Alan J. Hu. BDDNOW: A parallel BDD package. In *FMCAD '98: Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, pages 501–507. Springer-Verlag, 1998.

[31] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC '93: Proceedings of the 30th international Design Automation Conference*, pages 272–277. ACM, 1993.

[32] H. Ochi, S. Yajima, and N. Ishiura. A vector algorithm for manipulating Boolean functions based on shared binary decision diagrams. *Supercomputer*, 8(6):101–118, November 1991.

[33] Hiroyuki Ochi, Koichi Yasuoka, and Shuzo Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 48–55. IEEE Computer Society Press, 1993.

[34] Peter S. Pacheco. A user's guide to MPI, 1995. Manual.

[35] Y. Parasuram, E. Stabler, and Chin Shiu-Kai. 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*, pages 16–25, January 1994.

[36] R. K. Ranjan and J. Sanghavi. CAL-2.0: Breadth-first manipulation based BDD library, June 1997. Manual.

[37] Rajeev K. Ranjan, Jagesh V. Sanghavi, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstation. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 358–364. IEEE Computer Society, 1996.

[38] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47. IEEE Computer Society Press, 1993.

[39] Hyongkyoon Shin. *Verification of combinational circuits using conjunctively decomposed implications*. PhD thesis, Boulder, CO, USA, 1997.

[40] F. Somenzi. CUDD: CU decision diagram package, 1998. Manual.

[41] F. Somenzi. Binary decision diagrams. In *Calculational System Design,volume 173 of NATO Science series F: Computer and Systems Sciences*, pages 303–366. IOS press, 1999.

[42] Tony Stornetta and Forrest Brewer. Implementation of an efficient parallel BDD package. In *DAC '96: Proceedings of the 33rd annual Design Automation Conference*, pages 641–644. ACM, 1996.

[43] Hervé J. Touati, Robert K. Brayton, and Robert P. Kurshan. Testing language containment for omega-automata using BDD's. *Inf. Comput.*, 118(1):101–109, 1995.

[44] Congguang Yang, Maciej Ciesielski, and Vigyan Singhal. BDD decomposition for efficient logic synthesis. *International Conference on Computer Design*, 0:626, 1999.

[45] Songnian Zhou. A trace-driven simulation study of dynamic load balancing. Technical report, Berkeley, CA, USA, 1986.