UNIVERSITY OF CALIFORNIA
Santa Barbara

Implementation of an Efficient Parallel BDD Package

A Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical and Computer Engineering

by

Anthony L. Stornetta

Committee in charge:

Professor Forrest Brewer, Chairman

Professor Michael Melliar-Smith

Professor Klaus Schauser

December 1995

The dissertation of Anthony L. Stornetta is approved

Committee Chairperson

December 1995

December 16, 1995

ACKNOWLEDGEMENTS

I wish to express gratitude to Forrest for taking me into his research team and for his invaluable aid, without which I may never have been able to turn abstractions into tangible, working implementations! I must also thank Klaus and my lab mates for putting up with me while giving me much needed help and inspiration! Finally, I would like to thank my committee for taking the time to review my work.

ABSTRACT

Implementation of an Efficient Parallel BDD Package

by

Anthony L. Stornetta

Large BDD applications are generally limited by a processor's resources, especially physical memory. In large BDD applications, virtual memory is not an attractive solution for conventional depth-first BDD algorithms. Irregular memory access patterns in the algorithm cause thrashing, which severely impairs performance. One solution to overcoming this resource limitation is to utilize the memory of multiple networked workstations. A number of different forms of parallelism can be exploited in depth-first algorithms performed on a distributed BDD. The BDD structure is stored in a distributed hash table and can be accessed simultaneously by multiple processors. An unlimited number of multiple BDD operations may be concurrently issued. In essence, the package can simultaneously execute multiple threads of computation on a distributed BDD. This thesis presents an efficient parallel BDD package for a *distributed environment* such as a network of workstations (NOW) or a distributed memory parallel computer. The performance of this parallel BDD package is analyzed, and results for various circuit benchmarks are given.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Binary decision diagrams (BDD) are used in a wide variety of applications, such as circuit verification [14], combinatorial problems [13], FSM traversal [15], and other techniques where it is necessary to have an efficient means of representing and manipulating Boolean functions. Many of the current non-parallel BDD packages are based on the depth-first *if-then-else* (ite) algorithm [3].

Specialized programming techniques, such as attributed edges [5], dynamic variable re-ordering [6] and garbage collection [3], are often utilized in applications dealing with large BDD structures. However, these techniques are not always sufficient. Ultimately, a problem of arbitrary size is limited by the workstation's resources, primarily by the size of its physical memory. In conventional depth-first BDD algorithms, memory access patterns are highly irregular, such that when a BDD application begins to utilize a processor's swap space, performance drops considerably, as applications tend to thrash. An efficient swapping algorithm for BDD applications was proposed by [7], however, this method increased the size of a BDD by including redundant nodes. Furthermore, a problem that must rely on virtual memory is still much slower than one that can fit within the physical memory of one or more machines. As an alternative approach, parallel and distributed programming techniques may be used to combine the resources of several interconnected workstations, especially since network communication is generally faster than disk access. By combining resources, applications can take advantage of both more memory *and* more processing power. Such networks of workstations are common in the engineering workplace.

To date, parallel BDD implementations that have been developed include packages for shared memory multi-processor systems [2], for a distributed shared memory (DSM) platform [1], for SIMD architectures [10], and for vector processors [11]. In a shared memory multi-processor system, multiple processors share the same physical memory. Contention for shared memory is handled by semaphores and locks. While this method is fast, it is limited by the size of the shared memory machine. In a DSM platform, distributed memory is used to perform BDD algorithms. Memory can be increased by using DSM, however, as in the shared memory multiprocessor system, semaphores and locks must be used to prevent contention, and load balancing must be tuned for a particular application. SIMD architectures and vector processors suffer from a constrained instruction flow and are also limited to the amount of memory on the machine. This thesis describes a different type of parallel BDD library package developed for use in non-shared, distributed memory multi-processing environments such as a network of workstations (NOW) or a distributed memory parallel computer. This model is limited only by the number of workstations available to the computation and by network bandwidth. While it may need to be tuned for a particular network topology, the results obtained from our implementation of this model on the Meiko CS-2 are promising.

Often, parallelism may be extracted from a particular algorithm in several different ways. For instance, [10] explores parallelism in breadth-first BDD traversals [12]. In [2], parallelism in operation sequences is examined. This thesis describes techniques that allow several different forms of parallelism to be exploited in depth-first algorithms on a distributed BDD data structure. For instance, the BDD structure is stored in a distributed hash table and can be accessed simultaneously by multiple processors. Package library functions are also performed in parallel, utilizing multiple processors to perform operations on the distributed BDD. These library functions are similar to the non-parallel versions on which they are based. Finally, each processor executes its own thread, increasing the potential for parallelism at the application level. In essence, the package can simultaneously execute multiple threads of computation on a distributed BDD. These techniques apply to a wide range of different BDD applications.

Chapter 2

Binary Decision Diagrams

This description of binary decision diagrams is based upon the formal presentation of BDDs found in [4]. A binary decision diagram is a binary tree in the form of a directed acyclic graph (DAG) which represents one or more Boolean functions. Each level in this binary tree represents an input variable of the Boolean function. The only leaf nodes in this tree are the terminal nodes *1* and *0*, representing true and false, respectively. Each node points to child nodes along two edges: the *true* or *then* edge and the *false* or *else* edge. When the BDD is ordered such that input variables occur in ascending level order and is reduced so that no two nodes represent the same function, one obtains a *reduced ordered binary decision diagram* (ROBDD) [4]. This is the canonical internal representation of Boolean functions used in this parallel BDD package. This data structure is distributed across all processors involved in the application program's computation.

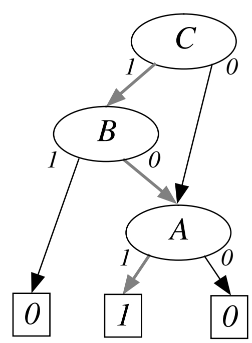


Figure 1: A BDD representing a Boolean function

The BDD in Figure 1 represents the Boolean function $A\bar{B}C + A\bar{C}$. Note that all variables at the first and lowest level in this tree must represent variable *A*, at the second level must represent variable *B*, and at the third and highest level shown must represent variable *C*. The *then* edge is denoted by a *1*, and the *else* edge, by a *0*. The path marked by the dashed arrows indicates that, if $A = 1$, $B = 0$, and $C = 1$, this function evaluates to true. This BDD has also been reduced so that the *don't care* node for variable B in $A\bar{C}$ does not appear.

2.1 If-Then-Else Algorithm

Most manipulations of Boolean functions in a BDD are performed by the if-then-else operation [3]. Given the arguments *F*, *G*, and *H*, the if-then-else operation is defined as: *if F then G*, *else H*. The Boolean representation of this function is:

$$ite(F, G, H) = F \cdot G + \bar{F} \cdot H$$

For instance, the BDD in Figure 1 can be computed by the if-then-else operations that result from:

$$\begin{aligned} A\bar{B}C + A\bar{C} &= C(A\bar{B}) + \bar{C}A \\ &= C(B \cdot 0 + \bar{B}A) + \bar{C}A \\ &= ite(C, ite(B, 0, A), A) \end{aligned}$$

More generally, basic two-variable Boolean functions are defined in terms of if-then-else. These are used to create Boolean functions represented by a BDD.

$$\begin{aligned} FG &= ite(F, G, 0) \\ F + G &= ite(F, 1, G) \\ \bar{F} &= ite(F, 0, 1) \end{aligned}$$

Thus, in terms of the three equations above, a programmer would most likely implement the function $A\bar{B}C + A\bar{C}$ as the following set of if-then-else operations:

$$A\bar{B}C + A\bar{C} = ite(ite(A, \underbrace{ite(\overbrace{ite(B, 0, 1)}^{\bar{B}}, C, 0)}_{\bar{B}C}, 0), 1, \underbrace{ite(A, \overbrace{ite(C, 0, 1)}^{\bar{C}}, 0)}_{A\bar{C}})$$

In the depth-first algorithm, each call to if-then-else begins a recursive evaluation of the three functions supplied as the arguments in *ite* (F, G, H). The function with the highest level in the BDD determines the next two iterations of if-then-else calls. The results from these two iterations are used to generate a new node in the BDD. If function F in a particular iteration of if-then-else has the highest level, the next two iterations will be *ite* (F_1, G, H) and *ite* (F_0, G, H), where F_1 is the function to which F's *then* edge points and F_0 is the function to which F's *else* edge points. These functions, F_0 and F_1, are cofactors of function F.

The place in which BDD nodes are stored is referred to in the literature as the unique table [3]. To maintain a *strong canonical form*, the unique table is generally built as a hash table, where each node is identified by a *unique id*. In other words, the same Boolean function may be built in several different ways, but it will be represented in the unique table only once. The unique table stores nodes according to a fixed variable ordering, such that every level represents only one Boolean variable. Furthermore, BDD nodes in the unique table are reduced so there are never any *redundant* nodes whose *then* and *else* edges point to the same node. A redundant node represents a *don't care* variable. In these cases, nodes on level i may skip redundant nodes by pointing to nodes on levels lower than i - 1.

There are potentially many paths from the top of a function to the leaves in a BDD. To reduce the amount of computation necessary when calculating recursive functions such as if-then-else, the literature describes a *computed table* [3] that is used to keep track of previous computations. Thus, if *ite* (F, G, H) has already been computed and if the results can be found in the computed table, an immediate result may be returned rather than recalculating the operation. The computed table is typically implemented as a hash-based cache.

2.2 Definitions

The non-parallel BDD package described in [3] implements the unique table as a *hash table* and the computed table as a *hash-based cache*. Additional data structures and methods introduced for the parallel implementation include a two-level hash table, a work queue, a global stack frame, an uncomputed table, and threaded computations. These are used to support an efficient model for parallel processing of the depth-first BDD algorithms:

1. A *two-level hash table* is a data structure that consists of *blocks* of *values*. To obtain a value, one must use a *key* that is associated with that value. Applying an initial hash function to the key returns a block. A secondary hash function is then used on the key to return the value within the block. Section 3.1, describes the construction of the unique table as a two-level hash table. Section 3.2 details the importance of having a good hash function.
2. The *hash-based cache* in this implementation differs slightly from [3], in that a collision based chain is used. Cache replacements are made in a least recently used (LRU) manner. The node cache, which is covered in Section 3.3, and the computed table, covered in Section 3.4, both use hash-based caches.
3. A *work queue* is a prioritized queue data structure which consists of lists of various different types of tasks that need to be completed. The work queue is covered in Section 4.2.
4. A *global stack frame* is similar to the idea of a program's stack frame within its heap. In order to keep track of the state of a recursive BDD computation that spans multiple processors, space must be set aside, analogous

to a recursive procedure's stack frame. Section 4.3 contains more details on the implementation of the global stack frame.

5. An *uncomputed table* is based on the idea of a computed table, except that it stores intermediate results for portions of the computation that have not yet completed. The uncomputed table grows as the computation begins and then shrinks as parts of the computation finish. It is covered in Section 3.4.
6. A *thread of computation* is defined as a single user operation on the distributed BDD, such as if-then-else or counting nodes. In the current implementation, multiple threads of computation are supported, as detailed in Section 3.4 and Chapters 4 and 5.

Chapter 3

Implementation

This section introduces the data structures used to store the distributed BDD. Also presented are various data structures and mechanisms which aid BDD computations.

3.1 Partitioning the Unique Table

The *unique table*, described in [3], is the data structure which stores the unique nodes of the distributed BDD. A depth-first approach was chosen over breadth-first because of the way in which tree traversal of a distributed BDD can occur simultaneously across many processors. Specifically, anywhere one node points to another node on a remote processor, work on the BDD can be forked to other processors at those points. During the computation, processors work at any level in the tree, as opposed to performing phases on one or more consecutive levels at a time such as in [7]'s breadth-first algorithm. Work partitioning is strongly correlated to the unique table partitioning strategy and is discussed in detail in Chapter 4.

To achieve a high degree of parallel execution, nodes must be well distributed across all processors. Yet, it is also necessary to minimize communications when possible. To suit these purposes, a *two-level hash table* is used to store the unique table. The initial hash function for the unique table first chooses a block by applying a *pseudo-random* function that is dependent upon a node's level and the unique ids of its left and right children nodes. Once a block is chosen, a request to get or put a node is sent to the processor which owns that block. This processor then uses a secondary hash function to identify or place the node within the block. In cases where a block becomes full, it may become necessary to apply a rehash function in order to choose a different block.

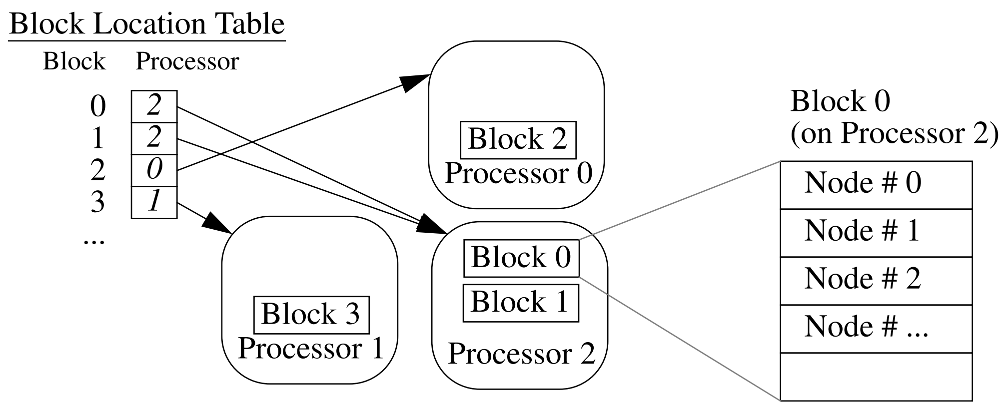


Figure 2: Unique table partitioning

Each processor maintains a *block location table*, as in Figure 2, to keep track of the processor location of blocks as they are created and moved. Each block is associated with a range of unique ids that corresponds to its position in the block location table. While this ties in with the unique table's strategy for fast location and placement of distributed nodes, it also allows the programmer to physically redistribute blocks. This feature is advantageous when one processor may have more memory than another and can store more blocks or when dynamic load balancing is required.

Unique table consistency is critical, especially in a distributed system. For example, two instances of the same node must not be created within the table. Likewise, the system cannot create two instances of the same block when two processors need to put nodes into the same block at the same time. To solve the first problem, unique table consistency is maintained by giving each processor responsibility for the nodes that it owns. Each processor updates its portion of the unique table uninterrupted, in an atomic region of code. To avoid block creation contention, as in the second problem, each processor is given an initial range of block numbers for which it is responsible. Once a block is created, the processor that created it sends an asynchronous message to the other processors which instructs them to add the new block to their block location tables.

3.2 The Hash Functions

In this model, there is a fixed maximum number of nodes that one can have in the unique table. Depending upon the size of the application and the resources available, the application may choose an appropriate size at run-time and garbage collect when necessary. In any case, the efficiency of this parallel BDD package is very sensitive to changes in the unique table's hash functions and can depend upon the maximum size of the unique table. Thus, one must be especially careful in choosing a hash function.

There are two levels of hashing when putting and getting nodes from the unique table. The first hash function, at the block level, chooses the initial block. The second hash function, at the node level, chooses a node within that block. A miss requires that a rehash function be used to choose a different node within the same block. If a block is too full, a rehash to another block is necessary. When putting a node into the unique table, a sequence of rehashes occurs until an empty spot in the unique table is found. To retrieve the node, the same hash sequence must be followed until the node is found. Nodes can be placed and retrieved more quickly when the average length of hash sequences is short. Thus, it is important that the block level hash function be well distributed. A good distribution at the block level ties in with the division of labor strategy, as discussed in Chapter 4.

For example, during development phases of the parallel BDD package, a hash function that encouraged a convoy effect was initially chosen for the block level. It was observed that, due to the poor distribution of the hash function, very few blocks out of the maximum number of blocks were actually being used, and the ones that were used filled up quickly. In general, the parallel BDD package did not require much memory. Small problems ran quickly, however, larger problems exhibited severe performance penalties due do the hash function. When the block level hash function was modified so that nodes were well distributed, all blocks were used. Much more memory was required to allocate every block, however, the performance of large problems improved a great deal.

To improve hash performance in the unique table, a *node rehash threshold* and a *block rehash threshold* were added. At the node level, the threshold determines the number of times a rehash can occur within one block before rehashing to another block. At the block level, if the number of block rehashes exceeds its threshold, the node is inserted into the next available block, ignoring the node rehash threshold. Every time a rehash at the block level is required, the chances are pretty good that it involves remote communication, especially if the unique table is widely distributed. The block rehash threshold helps keep interprocessor communication down. Nonetheless, if the block rehash cycles through every block in the unique table, then the hash table is completely full. A good hash function at the node level is helpful in maintaining an even distribution, but it does not have as much effect as the block hash function. Section 6.4 investigates the effects on performance when these thresholds are varied.

3.3 Node Cache

The node cache is unique to the parallel BDD package. Accessing nodes stored on other processors is a time consuming event that should be done only when necessary. Therefore, each processor has a read-only hash-based cache of nodes, used to store frequently used nodes that are located on other processors. Since they are so frequently accessed, the *0* and *1* terminal nodes are permanently cached. All other nodes are cached in an LRU fashion.

3.4 Computed and Uncomputed Tables

The *computed table* is a distributed hash-based cache that is used to store intermediate computational results [3]. The *uncomputed table* is a distributed hash table used to keep track of ongoing computations. The uncomputed table is necessary because results in an ongoing recursive BDD computation are most likely not immediately available. For instance, the current thread of computation may be temporarily stopped because it requires a result from another processor. Thus, it is necessary to keep a record of having started a computation to prevent it from being started multiple times. These tables reduce the execution time in problems that have exponential complexity [4]. They are also tied to the tree traversal strategy and are always locally accessible, as will be discussed in Section 4.1.

The uncomputed table provides a *forward* mechanism that allows stopped threads of computation to eventually receive the results of a step in a computation that has already been started but not yet completed. A list of forwards point to the threads of computation that are expecting the uncomputed result. An uncomputed result may have multiple forwards by the time it is finally computed. Thus, when it is obtained, the result is propagated via the forwarding mechanism so that any threads of computations that were waiting for the result can be restarted. The forwarding mechanism is similar in concept to *futures* described in [17]. When results are put into the computed table, they are removed from the uncomputed table. One advantage to the forward mechanism is that it coordinates the processing of multiple simultaneous threads of computation.

Chapter 4

Division of Labor

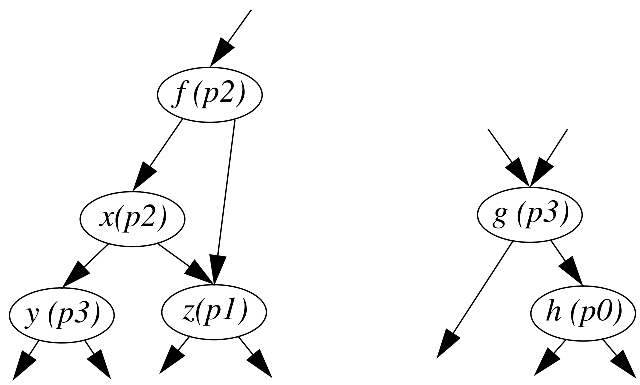
Chapter 3 described how the unique table was partitioned among processors. This section discusses how work is partitioned among processors. Work is defined as a small chunk of a calculation where tree traversal is necessary, such as performing one step in an if-then-else computation or in counting nodes.

4.1 Tree Traversal

The tree traversal strategy is related to the way in which the unique table is partitioned. As in depth-first algorithms, a recursive computation such as if-then-else is applied to the BDD. In our strategy, a top-down method, the *node with the highest level* in a given computation's arguments determines which processor performs the actual work. Thus, when the children of this node point to nodes on remote processors, work is forked to create new tasks on those processors. Given a wide distribution of nodes across processors, children will frequently point to nodes on other processors so that all processors will be busy working on some portion of the calculation.

For instance, in the case of *ite* (*f*, *g*, *h*), a recursive if-then-else computation on nodes *f*, *g* and *h*, if node *f* has the highest level among these nodes, the processor on which the node *f* resides is the processor that completes this stage of the computation. Figure 3 illustrates a typical scenario. If nodes *g* and *h* are not locally accessible and do not exist in the node cache, they must first be obtained from their respective remote processors. Then, two more recursive if-then-else computations are generated, each of which might also execute on remote processors.

Arbitrary nodes in a distributed BDD:



In ite (f, g, h) processor 2 performs the work

Figure 3: Work partitioning

At any given step of a thread of computation, there are three cases to consider, based on the processor location of children nodes: (1) work may be continued on two other processors, (2) work may be continued both the current processor and a remote processor, or (3) the current processor continues all the work itself. In the first case, the processor must, for the time being, stop working on the current thread of computation. It checks the work queue and begins on a different thread of computation. In the second case, part of the computation is dependent upon another processor's result. However, the same thread of computation is still active on this processor. In the last case, no other processor aids in computing the next step of the computation.

The tree traversal strategy is particularly sensitive to variation. For instance, in initial development, tree traversal order was determined by the first node argument in a particular function call, rather than by the node argument with the highest level. In the case of if-then-else, the processor that owned node *f* always performed the work. At first, this appeared to be a reasonable way to handle work assignment in the tree traversal. Upon closer inspection, it was discovered that this strategy exhibited extremely poor behavior in cases where node *f* was at the bottom of the BDD. When this happened, it would tend to stall the computation in such a way that the other processors frequently ran out of work in their queues, while the processor that owned node *f* was busy fetching nodes from other processors and computing its portion of the if-then-else call. By changing the strategy to the top-down method discussed above, the distribution of work was improved such that fewer processors remained idle.

Related to the tree traversal strategy, the portions of the computed and uncomputed tables pertaining to the highest level node also reside on the same processor on which the work is performed. Since the node argument with the highest level determines which processor does the work, it also determines on which processor, and specifically, in which block, the computed and uncomputed results for that function call are stored. Space in each block is reserved for the portions of these tables corresponding to the nodes in that block. Thus, rather than storing the computed and uncomputed tables in separate distributed data structures, they are tied to the tree traversal strategy so that they can always be accessed locally. For example at the beginning of a step in the if-then-else call, the local computed and uncomputed tables, are checked for already computed results or computations in progress. One benefit to this strategy is that when blocks move, the computed and uncomputed tables move with them.

4.2 Work Queue

The *work queue* facilitates parallel tree traversals and is the mechanism by which simultaneous threads of computation may be executed. It is an unordered, prioritized queue to which any processor may add tasks. Each processor examines its queue, one item at a time, and, as work is completed, asynchronous messages are sent to remote processors to add tasks to their appropriate work queues. Every time work is assigned to a remote processor, the computation bifurcates, allowing parallel execution of both sub-trees. It should be noted that the order in which this computation is completed is non-deterministic. There is no predefined ordering of interprocessor messages nor of tasks in the work queue. Consequently, the order in which cache replacements occur will also be non-deterministic. This randomness is reflected in the number of if-then-else calls and the execution times for a given application. Regardless, the output of the computation and the structure of the unique table (i.e. the number of nodes in a function) are always the same after every execution.

The two general types of tasks that can be added to a processor's work queue are *WorkDown* and *WorkUp* requests. WorkDown requests are generated when a computation is traversing one or more functions in the BDD. WorkUp requests are generated as portions of the computation are completed. In the current implementation, WorkUp requests have a higher priority than WorkDown requests because a WorkUp request frees the memory required by global stack frames, as discussed in Section 4.3.

A benefit of the work queue design is that multiple types of parallel computations can be performed simultaneously. For instance, in the user application, one processor can start any number of if-then-else computations. In effect, all processors begin simultaneous processing of all if-then-else computations. Different types of parallel computations may also be simultaneously executed. As an example, one processor may be working on the result of an if-then-else call while another processor is busy calculating the number of nodes in a function. Ideally, all processors will be kept busy generating results for one or more computations.

4.3 Global Stack Frames

Some mechanism is required to temporarily store a portion of the recursive computation until it can be continued at a later time. In other words, processors continually process their work queue and do not wait for results from other processors. It was also necessary to orchestrate the transfer of information between processors for each step of the calculation which crossed processor boundaries. To meet these needs, a persistent data structure, the global stack frame, was constructed so that processors could obtain the required information and perform steps of the calculation at their own convenience. The work queue and global stack frames cooperate to simulate multiple threads of computation.

Each stack frame keeps state information necessary to complete that step of the computation. It points to the parent stack frame to which it can send its result. Similar to futures [17], each frame reserves space to hold incoming results from recursive cofactor calls that traverse the BDD. In addition, global stack frames contain a list of zero or more forwards to any other stack frames that are waiting for the same result. As in a recursive procedure, frames in this global stack are created during recursive descents and then deleted on return.

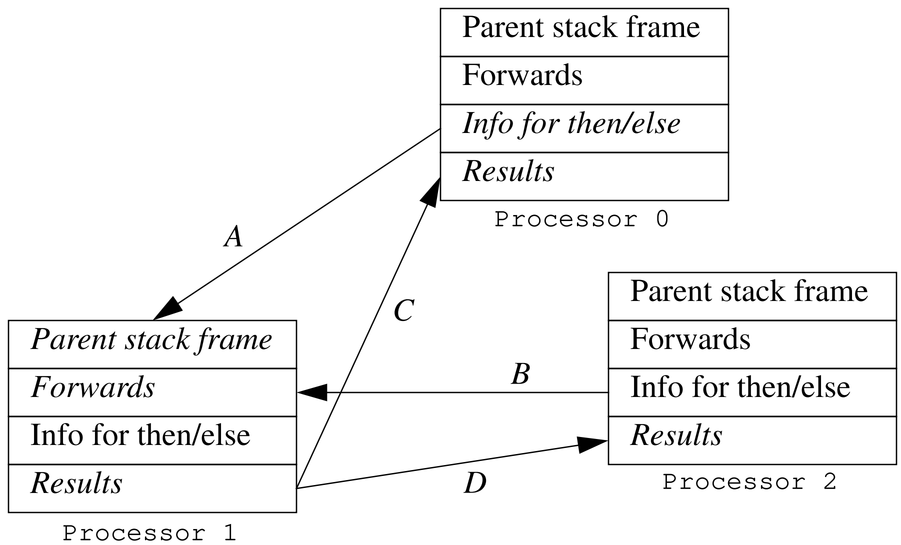


Figure 4: Global stack frames pertaining to if-then-else

Figure 4 illustrates the use of global stack frames in half of an if-then-else computation. In this example, processor 0 generates a stack frame as a result of an if-then-else computation. To complete, it requires the result of the next step from processor 1. Processor 0 sends processor 1 a WorkDown message and then addresses other items in its work queue. Upon receiving the WorkDown message, processor 1 creates a stack frame which contains a pointer to the parent stack frame on processor 0. Next, processor 1 obtains the then/else information from processor 0 necessary to complete that step of the computation (*A*). At some point during this computation, processor 2 discovers, via the uncomputed table, that it also needs the result of the computation in progress. Rather than recomputing this step of the computation, it leaves a forward on processor 1 (*B*). Once processor 1 has finished, it sends the result to processor 0 in a WorkUp message (*C*). Before the stack frame on processor 1 is deleted, it checks the list of forwards and propagates the result to processor 2 (*D*).

4.4 Node Locality

We define *node locality* as the frequency with which a node's children appear in the same block, and consequently the same processor, as the parent. A high degree of node locality means that a processor must go to a remote processor with less frequency when dereferencing children nodes in a traversal of the BDD. By modifying the hash function, an initial block may be chosen which directly increases the degree of node locality.

Initially, we thought that a high degree of node locality would serve to decrease execution time by increasing the amount of work done by a given processor without having to send work to a remote processor. Our experiments determined that a high degree of locality actually increased execution times a great deal. Our statistics revealed that the number of idle cycles per processor had increased dramatically. We determined that the reason for this is that node locality affects the performance of the tree traversal strategy. Since the computation bifurcates based on the location of child nodes, a low degree of node locality ensures that more processors are busy at the same time. Thus, a uniform, random distribution of nodes works very well in terms of keeping all processors busy. In a random distribution, approximately 6% of the nodes had one child node which was located on the same processor, while less than 1% of the nodes had one child node in the same block.

Based on modifying the hash function, our experiments seem to indicate that a small degree of node locality may work well with a uniform, random distribution of nodes. Currently, a node in the BDD is represented by one data structure. A better method for achieving a fine grained control over the degree of node locality may be to represent more than one nodes in a single data structure. In future work, we will further investigate the effects of different degrees of node locality.

Chapter 5

Communications Subsystem

The Parallel BDD package is implemented in C++ with a Split-C [8] communications interface. Split-C was used because it provided communication and synchronization primitives that were free of deadlock. Note that while these primitives do not deadlock, a poorly written application might still deadlock. Presented here are the methods Split-C provided and their restrictions.

Each processor runs a *single* thread of execution. In addition to its local thread of execution, multiple remote processor requests may be pending service on a processor, however, only one of these is serviced by the processor at a time. These single threads do not execute in lock-step, however, primitives are provided to the programmer for blocking and barriers. While the Split-C environment is single threaded on each processor, the work queue and global stack frames allows simulation of multiple simultaneous threads of computation.

Split-C provides both blocking and non-blocking function calls for exchanging data between processors and for performing remote procedure calls. Remote procedure calls are implemented as short pieces of code that are executed atomically on the remote processor. These may also either be blocking or non-blocking, depending upon the need. For instance, adding to a remote processor's work queue is a non-blocking operation that does not require acknowledgment. Putting a new node into the unique table is a blocking operation, returning the unique id as a result.

Due to its reliance on Active Messages [9], remote procedure calls in Split-C limit the amount of information passed with each remote procedure call. This posed a number of implementation difficulties for the parallel BDD package and is one of the reasons that global stack frames were developed. Furthermore, the specifications of

Split-C do not permit atomic procedures to initiate remote communications. In cases where these features would have been useful, a set of blocking communication calls accomplish the same thing.

Chapter 6

Experimental Results

The machine used in this implementation was a Meiko CS-2, configured with 64 scalar nodes. Each node features a SPARCstation 10 with 32 Mb of memory, 25 Mb of swap space, and a 1 Mb external cache. Furthermore, nodes are networked together by an Elan fat-tree interconnection. Each node has a communications coprocessor that performs internode communication via DMA transfers. Results are generated using partitions of sizes ranging from 2 to 32 processors. We used a puzzle application and various ISCAS85 benchmarks to analyze the performance of the parallel BDD package. The same programs were also compiled with an optimized, non-parallel BDD package for purposes of comparison.

6.1 Puzzle Application

The puzzle application computes the number of solutions in an arrangement puzzle. The puzzle is an n x n square grid cut up into p pieces, each of arbitrary shape. Given the pieces and the size of the grid, the application computes the number of placements by which the pieces could be arranged into the square grid such that no pieces overlap and all fit within the boundary. This application was chosen because the bulk of the computation, several large Boolean *and* operations, does not parallelize cleanly. Unlike the BDD construction benchmarks, the puzzle application must construct a BDD and perform complex operations on the BDD to generate a result. This application is a good test of the parallel package's usefulness in non-specific problems of large sizes. This application relies on the parallel version of the if-then-else operation to perform Boolean operations such as *and*, *or* and *not*. It also uses parallel versions of variable shift and node counting. The application has several procedures which are executed in parallel, some of which also utilize the parallel functions just mentioned.

6.2 Speedup in Execution Times

The puzzle application was used to determine the scalability of the parallel BDD package with different numbers of processors. Based on the results, it appears that, on the Meiko's fat-tree interconnect, the package scales quite well. For one 7x7 puzzle with 9 pieces, the following statistics were obtained:

Table 1: 7x7 Puzzle, 9 Pieces

Processors	Real Time (seconds)	Total Nodes	Avg Nodes/ Processor	Total if-then-else Calls
2	1,340.0	361,501	180,750	1,204,262
4	343.0	363,440	90,860	995,981
8	138.8	361,229	45,153	951,464
16	82.8	361,370	22,585	937,945
32	58.0	362,124	11,316	860,963
non-parallel version	34.9	358,261	358,261	1,285,797

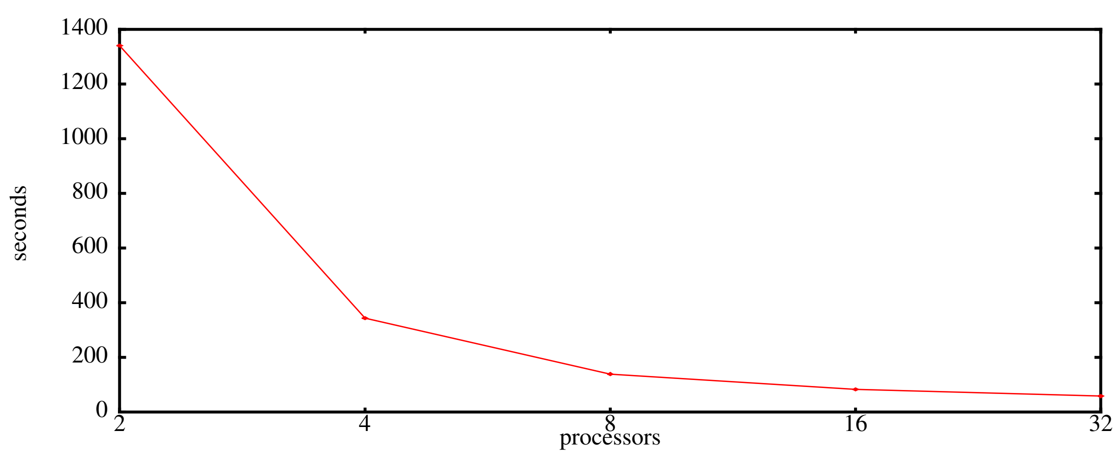


Figure 5: Execution times for 7x7 puzzle, 9 pieces

This 7x7 puzzle problem consisted of about 360K nodes and at least 850K if-then-else calls. The number of nodes and the number of if-then-else calls is nondeterministic in the parallel version. Depending upon the order of execution, fewer intermediate nodes may be created during the calculation. Furthermore, the success of the computed and uncomputed tables affects the total number of if-then-else calls. In the example above, garbage collection was not used. For purposes of comparison, the same example was compiled for a non-parallel, optimized BDD package running on a SPARCstation 10 with 128Mb of physical memory. It ran faster than the parallel version with 32 processors, as the overhead in the parallel version is still large. The parallel version did not finish on one processor due to memory constraints.

For a 6x6 puzzle with 6 pieces, the following statistics were generated:

Table 2: 6x6 Puzzle, 6 Pieces

Processors	Real Time (seconds)	Total Nodes	Avg Nodes/ Processor	Total if-then-else Calls
2	37.22	41,354	20,677	130,582
4	22.75	41,862	10,465	110,516
8	12.79	41,077	5,134	101,994
16	10.21	41,078	2,567	96,389
32	9.99	41,817	1,306	94,925
non-parallel version	4.25	40,316	40,316	162,022

There is significant overhead involved in processing the distributed BDD structure. In the table above, comparing the non-parallel version with the times for multiple processors reveals a substantial amount of overhead. With a distributed BDD package, the computation may take more than twice as long as an optimized non-

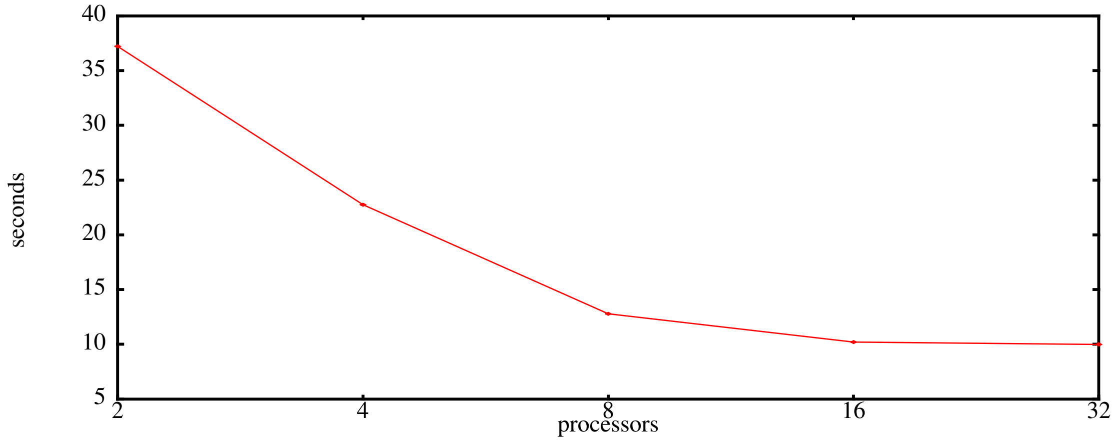


Figure 6: Execution times for 6x6 puzzle, 6 pieces

parallel implementation, especially in problems of smaller size. However, when computing larger problems, this performance penalty is tolerable when compared to the time it would take a machine with insufficient resources. For instance, the number of average nodes per processor, and consequently the memory required to store them, is substantially less as more processors join the computation.

To illustrate the benefit of a parallel BDD package over a non-parallel BDD package, the puzzle application was run on an 8x8 example with 12 pieces without garbage collection. The parallel version consisted of 32 processors on the Meiko. The non-parallel version ran on a SPARCstation 10 with 128Mb of physical memory. Table 3 shows that this application was able to run on the 32 processors and fit in memory without thrashing. The non-parallel version did not have enough physical memory, and thrashing dramatically reduced its performance.

Table 3: 8x8 Puzzle, 12 Pieces

Processors	Real Time (seconds)	Total Nodes	Avg Nodes/ Processor	Total if-then-else Calls
32 processors	3,015	6,971,273	217,852	15,358,229
non-parallel	23,980	6,950,150	6,950,150	25,450,950

6.3 Node Cache Results

The node cache speeds up execution time by locally caching nodes that are located on remote processors. We ran the c1355 benchmark on 32 processors using the original variable ordering to determine the effect of such a cache.

Table 4: Node Caching Effects on c1355 Benchmark

Cache status	If-then-else calls	Execution time (seconds)
Without node caching	1,408,282	140.1
With node caching	1,406,604	43.1

As Table 4 reports, when node caching was turned off, execution time took more than 3 times as long. In this particular example, we measured an average hit rate of 0.67 and an average miss rate of 0.33 for the node cache on all processors. In the puzzle application, much better average hit rates, between 0.75 and 0.85, have been observed. Node caching is likely to have a greater effect on performance in network topologies where communication latency and overhead is greater.

6.4 Hash Function Results

In the current implementation, each block may contain up to 1,024 nodes. Currently, each node requires 16 bytes of memory. The total size of the unique table is set at the beginning of each execution and can range from 128K nodes to 8M nodes, incremented by powers of 2. On the Meiko, an application can run comfortably with 128K nodes per processor. For example, given 128K nodes on 32 processors, applications can utilize a total of 4M nodes. Applications have been tested with 256K and 512K nodes per processor, however performance decreases considerably as swapping begins to occur.

The block rehash threshold determines how quickly a node can be placed or located in the unique table. Generally, if the block rehash threshold is set high, nodes can have longer rehash sequences, especially across processor boundaries. A long hash sequence usually overlaps well with other concurrent computation, however the following table illustrates that it does affect performance. To generate these statistics, a

Table 5: Block Rehash Threshold Variance

Rehash Threshold	Time (seconds)	If-then-else calls	Total number of communications
0	128.1	938,198	335,667
1	74.4	975,915	380,937
2	76.9	913,950	405,838
3	76.4	912,544	441,661
4	83.3	923,211	478,071
5	82.1	955,587	517,378
6	85.1	926,565	539,461
7	86.6	934,874	566,570
8	88.6	906,045	590,331
9	92.6	912,203	613,277
10	90.7	875,461	625,648
11	103.8	988,261	666,156

7x7 puzzle application with 9 pieces was executed on 16 processors. Execution times contain an error factor that ranges approximately ± 4 seconds, depending upon the order of execution and the hits on the computed and uncomputed tables. The total number of communications includes all blocking and non-blocking communications. It is interesting to note that this problem took longer with a rehash threshold of 0, despite the lower total number of communications. Our statistics revealed that

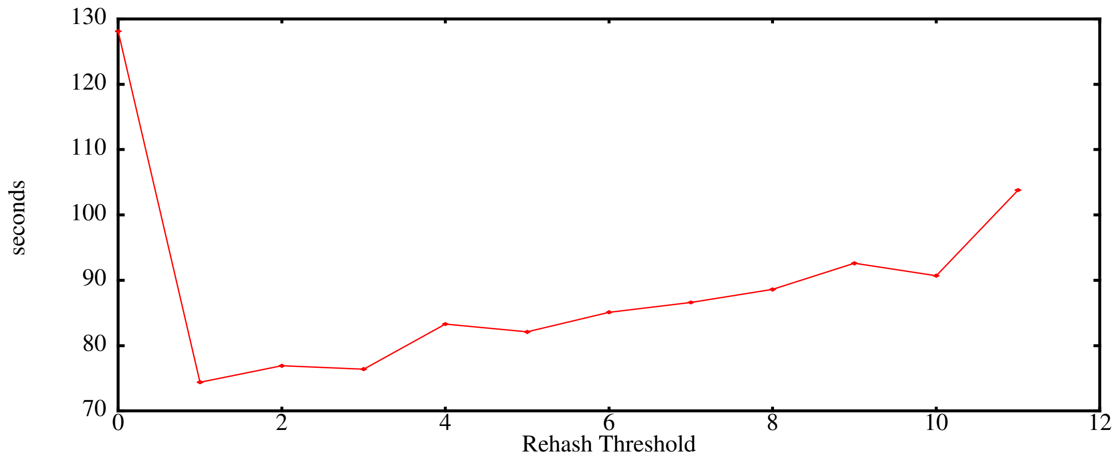


Figure 7: Execution times for different block rehash thresholds

processors were idle about twice as long the same problem with a rehash threshold of 1. In this case, fewer work requests were being generated on remote processors.

The node rehash threshold does not seem to play as much of a role in affecting performance. While it does tend to keep things well distributed, block rehashes incur a much larger penalty and therefore affect performance to a higher degree. The following statistics were gathered for the 7x7 puzzle application with 9 pieces. The block rehash threshold was set to 3, and it was also run on 16 processors.

Table 6: Node Rehash Threshold Variance

Rehash Threshold	Time (seconds)	If-then-else calls	Total number of communications
0	78.0	950,902	452,918
1	74.8	922,411	447,508
2	76.2	924,983	446,715
3	73.1	868,170	433,731
4	72.4	861,167	431,676
5	73.1	869,858	433,572
6	74.6	866,653	433,003
7	73.4	896,546	437,067

Table 6: Node Rehash Threshold Variance

Rehash Threshold	Time (seconds)	If-then-else calls	Total number of communications
8	76.2	960,171	448,742
9	74.7	915,220	440,686
10	73.4	866,524	433,877

6.5 LogP Analysis

Using LogP analysis [16], the execution time for a sample application can be predicted for a NOW environment with a different network topology than the Meiko. The ISCAS 85 benchmark c1355 was run on the Meiko with the original variable ordering, generating the following statistics:

Table 7: ISCAS 85 Benchmark c1355

Processors	Execution Time in seconds	If-then-else calls/processor	Number of nodes/ processor	Communications/ processor
32	44.1	43,727	21,286	176,481

In the table above, the communications/processor field refers to the average number of blocking communications per processor such as a remote read or a blocking atomic function call. If we consider that the Meiko's elan fat-tree network has a latency, *L*, of 10 μs and an overhead, *o*, of 2 μs, the time for one blocking communication can be expressed as at least $4o + 2L$, or approximately 28 μs. Given 176,481 blocking communications per processor, we can estimate that each processor spent at least 4.9 seconds or 11.2% of its time waiting on the network and at most 39.2 seconds computing.

In a network of workstations, connected via ethernet, a much higher latency of about 500 μs must be assumed. If the time for one blocking communication can be expressed as 1 ms, then the time spent waiting on the network jumps to at least 176 seconds. If we ignore contention, the LogP model predicts that the same application would execute on a network of workstations in at least 216 seconds. Depending on the network load, contention may increase the predicted execution time.

6.6 ISCAS85 Benchmarks

The ISCAS85 benchmarks that could fit within a 2M node-space on 32 processors, without garbage collecting and other compression techniques, generated results as shown in Table 8. These statistics were generated by choosing a predefined ordering of input variables.

Table 8: ISCAS 85 Benchmarks

Benchmark	Number of Nodes (32 Procs)	If-then-else Calls (32 Procs)	Execution Time in seconds (16 Procs)	Execution Time in seconds (32 Procs)
c432	30,008	129,532	6.6	4.4
c499	52,608	164,439	7.6	5.0
c880	106,474	303,214	14.7	10.4
c1355	228,230	536,419	21.9	17.0
c1908	92,343	232,259	9.0	6.1
c5315	22,955	53,997	3.5	2.9

6.7 Node Locality

As discussed in Section 4.4, the hash function was modified so that we could investigate varying degrees of node locality on a block level. Our current strategy uses a uniformly random placement of nodes into the unique table. By changing the hash function, we could control the probability with which a parent node would point to its children. In Table 9, the results of modifications of the hash function which affect node locality are presented.

Table 9: Effects of Node Locality

Hash strategy	Total if-then-else calls	Execution Time (seconds)	Average communications /processor	Average node locality
Random	112,764	13.1	43,561	0.4%
1/16 probability	121,482	12.8	44,686	2.4%
1/8 probability	121,497	12.7	44,293	4.2%
1/4 probability	104,718	183.4	89,200	62.2%
1/2 probability	104,824	190.6	91,963	70.3%

The statistics in the table above were generated by running the puzzle application on a 6x6 puzzle with 6 pieces. The average node locality is a measurement of the percentage of nodes in the unique table which point to child nodes that are in the same block. The hash strategy employs a pseudo-random function to force parent nodes to exist in the same block as children nodes with the given probability. As average node locality increases slowly, execution time begins to drop, however, when a 1/4 probability hash strategy is used, locality as well as execution time jumps dramatically. The average communications per processor increases by about twice as much because processors must grab nodes on remote processors more frequently. However, the primary reason that execution time exploded is that with fewer processors being busy at the same time, due to an increased node locality, the partitioning of work failed such that more processors were idle more often than in the cases where there was less node locality.

Chapter 7

Conclusions

This thesis has described an efficient parallel BDD package for a distributed memory multi-processor system. Not only have the resources of multiple processors been utilized by storing a distributed BDD, but they can also simultaneously execute multiple threads of computation on a distributed BDD. Several techniques have been introduced which allow depth-first algorithms on a BDD to be performed in parallel. A two-level hash was introduced as means by which to store a distributed BDD's unique table. The uncomputed table coordinates multiple threads of computation so that the same computation is not continually restarted. The forwarding scheme complements the uncomputed table by helping to keep track of these threads. Finally, the node cache reduces communication overhead, which would be particularly important in NOW environments.

Some compression techniques, such as attributed edges [5] and dynamic variable re-ordering [6], are currently not supported. However, because the parallel algorithms are based on efficient non-parallel versions of a BDD package [3], these techniques can be easily added to the parallel BDD package in future work. For example, a parallel garbage collection function has been added that is based on a depth-first garbage collection algorithm typically used in non-parallel BDD packages.

A dynamic memory allocation scheme for the unique table may result in better performance than the fixed table size scheme implemented in the current parallel BDD package. This may especially be the case on a network of workstations, where the communication overhead on an arbitrary network topology is large. There would be no fixed limit on the size of the unique table, and block rehashes could be eliminated. Uneven distributions of nodes among processors with varying sizes of physical memory could be obtained by partitioning the uniform distribution of the hash function. This may result in a parallel BDD package with increased flexibility. One drawback to a dynamic memory allocation scheme is that it would be more difficult to move a set of nodes from one processor to another. Furthermore, the block rehash threshold seems to solve the problems of excessive communications in the current fixed size unique table scheme. Dynamic memory allocation in the parallel BDD package will be investigated in future work.

The experimental data in the puzzle example is not one of the standard circuit benchmarks, but it is representative of the effective performance of a parallel BDD package with a distributed unique table. While the parallel package was slower than an optimized conventional non-parallel BDD package for smaller examples, the results clearly show the advantage of utilizing a pool of processors for solving larger problems where a single machine's resources are insufficient.

This research is applicable to any system comprised of workstations that are networked together. In future work, we would like to test this parallel BDD package on different network interconnects. In particular, we would like to test it on a conventional NOW. Given the fat-tree network connecting the processors on the Meiko and the statistics gathered from the results of the puzzle program, it would appear that this package is scalable to a large number of processors on such a network. Execution times always dropped with the addition of more processors.

In future work, we will further analyze the behavior of this model, especially on different network interconnects. Code profiling will allow us to make further optimizations and adjustments to improve performance. For instance, putting nodes into the unique table could be changed into a split-phase operation as opposed to the current blocking implementation. It is also necessary to look for various ways to reduce communications between processors. We discovered that too much node locality breaks the distribution of work strategy, however a slight increase in node locality may be beneficial. One way to do this may be to store nodes in small clusters. Finally, in the distributed recursive operations, there is currently a sizeable amount of overhead. Reducing this overhead will allow us to increase the size of our unique table and to potentially gain improvements in execution time.

References

[1] Y. Parasuram, E. Stabler and Shiu-Kai Chin. "Parallel Implementation of BDD Algorithms Using a Distributed Shared Memory." In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences Vol I: Architecture*, pp 16-25, January 1994.

[2] S. Kimura, T. Igaki, H. Haneda. "Parallel Binary Decision Diagram Manipulation." In *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E75-A, No. 10, pp 1255-62, October 1992.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. "Efficient Implementation of a BDD Package." In *Proceedings of 27th ACM/IEEE Design Automation Conference*, pp. 40-45, June 1990.

[4] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation." *IEEE Transactions on Computers*, Vol. C-35, No. 8, pp. 677-691, August 1986.

[5] S. Minato, N. Ishiura and S. Yajima. "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation." In *Proceedings of 27th ACM/IEEE Design Automation Conference*, pp. 52-57, June 1990.

[6] R. Rudell. "Dynamic Variable Ordering for Ordered Binary Decision Diagrams." In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 42-47, Santa Clara, CA, November 1993.

[7] H. Ochi, K. Yasuoka, S. Yajima. "Breadth-First Manipulation of Very Large Binary-Decision Diagrams." In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 48-55, 1993.

[8] D. Culler et al. "Parallel Programming in Split-C." In *Proceedings SUPERCOMPUTING '93,* pp 262-73, November 1993.

[9] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser. "Active Messages: a Mechanism for Integrated Communication and Computation." In *Proceedings of the 19th International Symposium on Computer Architecture*, May, 1992.

[10] S. Gai, M. Rebaudengo, M. S. Reorda. "A Data Parallel Algorithm for Boolean Function Manipulation." In *Proceedings. Frontiers '95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pp 28-34, February 1995.

[11] H Ochi, S. Yajima, N. Ishiura. "A Vector Algorithm for Manipulating Boolean Functions Based on Shared Binary Decision Diagrams." *Supercomputer*, Vol. 8, No. 6, November 1991.

[12] P. Ashar and M. Cheong. "Efficient Breadth-First manipulation of Binary Decision Diagrams." *Proceedings IEEE International Conference Computer-Aided Design*, pp. 622-627, 1994.

[13] S.-I. Minato. "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems." Proc. 30th *ACM/IEEE Design Automation Conference*, pp. 272-277, 1993.

[14] S. Kimura. "Residue BDD and Its Application to the Verification of Arithmetic Circuits." Proc. 32th ACM/IEEE Design Automation Conference, pp. 542-545, 1995.

[15] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. "Algorithms for Approximate FSM Traversal." Proc. 30st *ACM/IEEE Design Automation Conference*, pp. 25-30, 1993.

[16] D. Culler et al. "LogP: Towards a Realistic Model of Parallel Computation." *SIGPLAN Notices*, Vol. 28, No. 7, pp. 1-12, July 1993.

[17] A. Agarwal et al. "APRIL: A Processor Architecture for Multiprocessing." *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 104-14, 1990.