Implementation of an Efficient Parallel BDD Package

Tony Stornetta Forrest Brewer

Department of Electrical and Computer Engineering
University of California, Santa Barbara, U.S.A.

Abstract

Large BDD applications push computing resources to their limits. One solution to overcoming resource limitations is to distribute the BDD data structure across multiple networked workstations. This paper presents an efficient parallel BDD package for a distributed environment such as a network of workstations (NOW) or a distributed memory parallel computer. The implementation exploits a number of different forms of parallelism that can be found in depth-first algorithms. Significant effort is made to limit the communication overhead, including a two-level distributed hash table and an uncomputed cache. The package simultaneously executes multiple threads of computation on a distributed BDD.

1. Introduction

Binary decision diagrams (BDD) are used in a wide variety of applications where it is necessary to have an efficient means of representing and manipulating Boolean functions, such as in circuit verification [13] and combinatorial problems [12]. A problem of arbitrary size is generally limited by a workstation's resources, primarily the size of its physical and virtual memory. For instance, when a BDD application begins to utilize its swap space, it tends to rely on it heavily. An efficient swapping algorithm was proposed in [7]. However, an alternative is to combine the resources of several workstations. This is advantageous because, by combining resources, there is both more memory *and* more processing power.

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 [9], and for vector processors [10]. This paper 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).

Often, parallelism may be extracted from a problem in several different ways. For instance, [9] explores parallelism in breadth-first BDD traversals [11]. In [2], parallelism in operation sequences is examined. This paper describes a technique that allows several different forms of parallelism to be exploited in depth-first algorithms on a distributed BDD data structure.

A formal presentation of definitions and terminology can be found in [4]. Our parallel implementation does not currently support compression techniques such as attributed edges [5] or dynamic variable re-ordering [6]. However, because the distributed algorithms are strongly based on an efficient non-parallel BDD package [3], these techniques will be supported in future work.

2. Implementation

This section introduces the data structures used to store the distributed BDD. As in [3], our parallel BDD package uses *hash tables*, *hash-based caches*, a *strong canonical form*, a *computed table* and a *unique table*. However, the hash-based cache in this implementation differs slightly, in that a collision based chain is used and cache replacements are made in a least recently used (LRU) manner. The node cache, covered in Section 2.3., and the computed table, in Section 2.4., use hash-based caches.

A *two-level hash table* is a data structure that consists of *blocks* of *values*. To obtain a value associated with a *key*, one must first use an initial hash function on the key, which returns a block. A secondary hash function is then used on the key to return the value within the block. Section 2.1., describes the construction of the unique table as a two-level hash table.

2.1. Partitioning the Unique Table

The *unique table*, described in [3], is distributed across *N* processors. A depth-first *ite* was chosen over breadth-first to allow parallel traversal of sub-trees on a BDD distributed across many processors. Specifically, anywhere one node points to a node on a remote processor, traversal of the BDD can be forked to the other processor. Thus processors work at any level in the tree, as opposed to performing level-constrained phases as in [7]'s breadth-first algorithm. This strategy also simplifies redistribution of the BDD as work progresses. Task partitioning strongly relates to the unique table partitioning strategy discussed in Section 3.

To achieve a high degree of parallel execution, nodes must be well distributed across all processors. Yet, it is also necessary to minimize communications overhead. 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 dependent upon a node's level and the unique ids of its left and right children nodes. Given a block choice, 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 place the node within the block. In cases where a block fills up, it may become necessary to apply a rehash function in order to choose a different block.

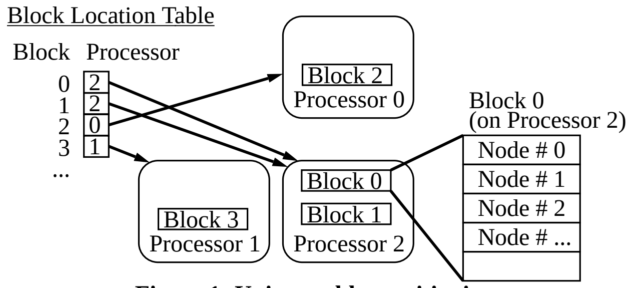

Figure 1: Unique table partitioning

Each processor maintains a *block location table* (see Figure 1) to indicate dynamically which processor owns which block. Each block is associated with a range of unique node id's that correspond to its position in the block location table. This allows fast location and placement of distributed nodes while also enabling physical redistribution of blocks between segments of large calculations. This feature is advantageous since the distribution of nodes can change a great deal during garbage collection.

Unique table consistency is critical, especially in a distributed system. We must insure that multiple instances of the same node cannot be created. 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. Similarly, the system must not create two instances of the same block when two processors create nodes at the same time. To avoid this, 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.

2.2. The Hash Functions

The execution time of this parallel BDD package is very sensitive to changes in the hashing functions for the unique table. Since the unique table is a two level hash, one must be especially careful in choosing the hashing functions. If the block-level hashing function is well distributed, nearly all blocks in the system are allocated, helping to distribute the nodes evenly. At the node-level, a *rehash threshold* is used to control the number of times the package performs a rehash for a node in one block. Our experiments indicate that the block-level hashing function has much more of an effect on performance than the rehash threshold.

In the current implementation, each block contains 1,024 nodes. The package makes allowances for up to 2K blocks in the unique table, for a total of up to 2M nodes overall. As long as both the number of nodes in a block and the total number of blocks equal a power of 2, these specifications can be easily changed. In future work, these values will be varied in order to find the optimal block size and unique table maximum since both parameters affect the performance of this hash-based implementation.

2.3. Node Cache

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.

2.4. 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 waiting because it requires a result from another processor. It is necessary to keep a record of having started a computation to prevent it from being started multiple times. Both tables are tied to the tree traversal strategy and are always locally accessible, as will be described in Section 3.1.

The uncomputed table provides a *forward* mechanism that allows stopped threads of computation to receive the results of a step in a computation that has already been started but not yet completed. This forward points to the thread of computation that is 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 are restarted. When results are put into the computed table, they are removed from the uncomputed hash table. An advantage of the forward mechanism is that it coordinates the processing of multiple simultaneous threads of computation.

3. Division of Labor

Section 2 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 tree traversal algorithm, such as an *ite* (if-then-else) or node counting computation. Several new data structures are required to support parallel computation.

A *work queue* is a data structure which consists of a list of various different types of tasks that need to be completed. The work queue is discussed in Section 3.2.

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 5.3 contains more details on the implementation of the global stack frame.

A *thread of computation* performs individual work queue items on a single processor. The next sections will describe how multiple threads of computation are supported.

3.1. Tree Traversal

The tree traversal strategy is related to the node partitioning. As in depth-first algorithms, a recursive operation similar to *ite* is applied to the BDD. In our strategy, the *node* with the highest level in a given operation's arguments determines which processor performs the actual work. Thus, when the children of this node point to remote nodes, the effort forks new threads (work items) on the other processors. Given a distribution of nodes across processors, children will frequently point to nodes on other processors, and so all processors will be busy working on some portion of the calculation.

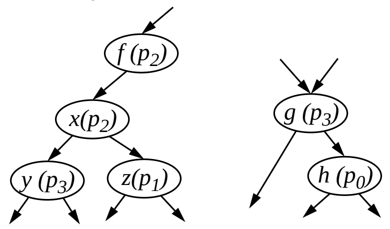
Arbitrary nodes in a distributed BDD:
$f (p_2)$
$x(p_2)$
$g (p_3)$
$y (p_3)$
$z(p_1)$
$h (p_0)$
In *ite (f, g, h)*, processor 2 performs the work

Figure 2: Work partitioning

For instance, in the case of `ite(f, g, h)`, if node *f* has the highest level among *f*, *g*, or *h*, the processor on which the node *f* resides is the processor that performs this stage of the computation. Figure 2 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. This function typically generates two more recursive if-then-else operations, each of which might also execute on remote processors.

At any given step of a thread of computation, there are three cases to consider: (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 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 thread of computation is still active on this processor as well. In the last case, the local processor performs the next step of the computation.

As part of the strategy of assigning work to the processor that owns the node with the highest level, the portions of the computed and uncomputed tables relating to that node also reside on the same processor. Space in each block is reserved for the portions of these tables that correspond to the nodes within that block. Thus, the computed and uncomputed tables are also tied to the tree traversal strategy so that they can always be accessed locally. For example, at the beginning of a step in an *ite* call, the computed and uncomputed tables will always reside on the same processor where this step is being computed. This allows for efficient speedup of the traversal. An added benefit of this strategy is that when blocks move, the computed and uncomputed tables move with them.

3.2. 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 nondeterministic. 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 nondeterministic. This randomness is reflected in the total number of if-then-else calls, in the number of garbage nodes generated, and in the execution times for a given application. Nonetheless, the output of the computation and the structure of the unique table for a computed function are always the same after every execution.

A benefit of the work queue design is that multiple types of parallel operations can be performed simultaneously. For instance, while one processor helps calculate the number of nodes in a function, several other processors work on multiple *ite* operations at the same time. The package can thus support parallel execution of multiple user functions. Ideally, all processors will be kept busy generating results for one or more operations.

3.3. Global Stack Frames

A 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 is also necessary to orchestrate the transfer of information between processors for each step of the calculation which cross processor boundaries. To meet these needs, a persistent data structure, the global stack frame, was constructed so that processors can 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 also points to the parent stack frame to which it can send its result. In addition, global stack frames contain forwards to any other stack frames that are waiting for the same result. As in a recursive procedure, objects in this global stack are created during recursive call and are deleted on return.

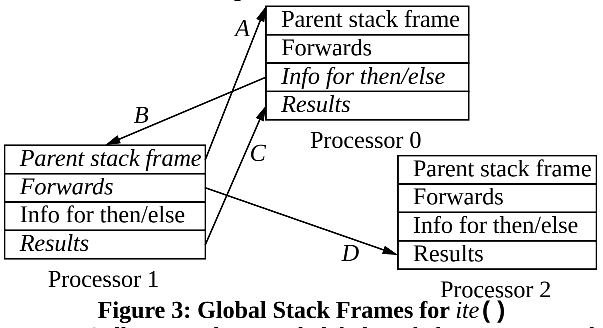

Figure 3: Global Stack Frames for *ite*( )

Figure 3 illustrates the use of global stack frames in part of an if-then-else computation. In this example, processor 0 generated a stack frame as a result of an if-then-else computation. To finish, it requires the result of the next step from processor 1, so it sends the work queue on processor 1 a work item. Meanwhile, processor 0 addresses other items in its work queue. Processor 1 creates a stack frame which contains a pointer to the parent stack frame (***A***). Next, processor 1 obtains the then/else information from processor 0 necessary to complete that step of the computation (***B***). Once processor 1 has a result, it is sent back to processor 0 as a work item (***C***). At some point during this computation, processor 2 discovered, via the uncomputed table, that it also needed the same result, so it left a forward on processor 1. Before the stack frame on processor 1 is deleted, the result is also forwarded to processor 2 (***D***).

4. Experimental Results

The Parallel BDD package is implemented in C++ with a Split-C [8] communications interface. In Split-C, each processor runs a *single* thread of execution. This thread may either be a local thread of execution or one or more remote requests, however only one of these is serviced by the processor at a time. The threads do not execute in lock-step, however, as primitives are provided to the programmer for blocking and barriers. While the Split-C environment is single threaded, the work queue and global stack frames allow us to simulate multiple simultaneous threads of computation.

The machine used in this implementation was a Meiko CS-2, configured with 64 scalar processor nodes. Each processor featured a SPARCstation 10 with 32MB of memory, 25 MB of swap space, and a 1MB external cache. Furthermore, processors were networked together by an Elan fat-tree interconnection. Each processor featured a communications coprocessor that performed internode communication via DMA transfers. In the following experiments, a 32 processor partition was used to generate results for the parallel version of our BDD library. For comparison purposes, some of the experiments were also executed using a conventional, optimized BDD library that runs on a single workstation. We tested this package on a puzzle application and on various ISCAS85 benchmarks.

4.1. Puzzle Application

The first application utilized the BDD package to compute the number of solutions on an arrangement puzzle. The puzzle consisted of a square grid, *n* x *n*, cut along grid lines into *p* pieces of arbitrary shapes. Given the pieces and the size of the grid, the application computed the number of placements by which all the pieces could be arranged onto the square grid. This application was chosen because the bulk of the computation, several large Boolean *and* operations, do not parallelize cleanly. Unlike BDD construction benchmarks, the puzzle application preforms complex operations on the BDD to generate a useful result. Thus, it was a good test of the parallel package's usefulness in non-specific problems of large sizes.

Table 1: 7x7 Puzzle, 9 pieces

Processors	Time (sec)	Total Nodes	Nodes/ Processor	if-then-else Calls
non-parallel	3357	358,261	358,261	1,285,853
1	------	>109,154	>109,154	>162,846
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

In the first experiment, we generated a 7x7 puzzle problem with 9 pieces to test the performance of the parallel library. Table 1 illustrates the results of a number of configurations. The parallel version did not complete on one processor, having run out of virtual memory. The non-parallel version, running on a single processor, took nearly an hour to complete and spent a fair amount of time thrashing. By combining resources, our parallel version was able to beat this time by more than a factor of 10 with just 8 processors.

As described in section 3, the number of if-then-else calls is nondeterministic. However, in Table 1, the number of if-then-else calls decreased for increasing numbers of processors. The reason for this is that as more processors cooperated in the computation, the cumulative size of the cache increased. Thus, the cache size for 32 processors was much larger than the cache on one processor and consequently had more hits. A much larger version of the problem was run on a version of the package with 8M available nodes. Although at this size, each of the processors in the distributed version actually swapped, there was still significant speedup as shown in Table 2. In this example, the non-parallel version is also a Sparc-

Table 2: 8 x 8 Puzzle, 12 Pieces

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

10 but one with 128MB of RAM and 200MB of swap space. Even though both implementations required swapping, the parallel version was more than 8 times as fast, even given the communication overhead.

Table 3 shows some statistics generated for a smaller puzzle problem. Part of the execution overhead time is the time required to load the program executable to all processors involved in the computation (a few seconds). There is clearly significant overhead in processing the distributed BDD structure. In fact, the lack of any communication overhead allowed one processor to finish the parallel application more quickly than two processors. The non-parallel BDD package, which has no communications overhead and is optimized for a single processor, finished this small problem in about 4 seconds. However, the distributed version must contend with communication delays between the components performing the calculation. It is clear that much of this communication is performed in parallel in the large partitions to obtain the observed performance. Small problems on the distributed BDD package may take several times as long as a non-parallel implementation. However, this performance penalty is tolerable compared to that incurred when local workstation resources are exhausted.

Table 3: 6 x 6 Puzzle, 6 pieces

Processors	Time (seconds)	Total Nodes	Nodes/ Processor	if-then-else Calls
non-parallel	4.25	40,316	40,316	162,024
1	23.30	40,316	40,316	87,696
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

4.2. Hashing Function Results

Several portions of the implementation required tuning to obtain best performance, of these, the hashing related tuning was most critical. The block rehash threshold provided significant improvement up to about 3 (rehashes in block), after which the performance slowly degraded. Node caching provided an additional factor of about 4 in performance on the Meiko. The surprising result was that block distribution apparently worked best when there was very little locality in the blocks with relation to the tree traversal. This is shown in Table 4 where the run-times for variations in hash locality are detailed. In this case, a pseudo-random function forced parent nodes into the same block as their children with a given probability. The increase in messages stem from this requiring updates to the block tables. However, the large increase in run time is due to several processors going idle due to the poor distribution of nodes.

Table 4: Node Locality 6 x 6 Puzzle

Hash	Total ite	Time (sec)	Messages	Local nodes
Random	112,764	13.1	43,561	0.4%
1/16	121,482	12.8	44,686	2.4%
1/8	121,497	12.7	44,293	4.2%
1/4	104,718	183.4	89,200	62.2%
1/2	104,824	190.6	91,963	70.3%

4.3. ISCAS Benchmarks

The ISCAS85 benchmarks that could fit within a 2M node-space on 16 processors without garbage collecting and other compression techniques generated results as shown in Table 5. These statistics were generated by choosing a predefined ordering of input variables

Table 5: ISCAS 85 Benchmarks

Benchmark	Number of Nodes	If-then-else calls (32P)	16 Proc. Time (sec.)	32 Proc. Time (sec.)
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

5. Conclusions

Given the results of our experiments, it would appear that this package will scale to a large number of processors. In running BDD applications of larger sizes, 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.

This paper has presented a means of developing an efficient parallel BDD package on a distributed memory multi-processor system. Several techniques have been introduced which allow parallelization of depth-first search algorithms on a BDD. A two-level hash provides a means by which to store a distributed BDD's unique table. Together, the computed table and the uncomputed table coordinate parallel threads of computation. The forwarding scheme complements the uncomputed table by keeping track of these threads. A node cache helps reduce communication overhead. Finally, 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.

The experimental data in the puzzle application is not a standard circuit benchmarks, but it does demonstrate the effective performance of a parallel BDD package with a distributed unique table. Our experiments clearly show the advantages in using multiple distributed processors to solve larger problems, where single machine resources become exhausted.

6. 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] 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.

[10] 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.

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

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

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