BDDNOW: A Parallel BDD Package

Kim Milvang-Jensen
Department of Computer Science
University of Copenhagen (DIKU)*

August 17, 1998

Abstract

BDDs (binary decision diagrams) are ubiquitous in formal verification tools. The time and memory used by the BDD package are frequently the constraints that prevent application of formal verification. Parallelization is an obvious way to try to cope with this. In this paper, we present a parallel BDD package with several novel features. The parallelization scheme strives for a small communication overhead, and it is targeted at a network of workstations. We are able to demonstrate speed-up even running on networked commodity PC workstations. Average memory utilization per node is comparable to that of efficient sequential packages. In addition, the package supports dynamic variable reordering, and simultaneous computation of multiple BDD operations. Finally, the package is designed for portability – providing a subset of the CUDD API for the application programmer, and running on the widely available PVM package.

---

*This work was performed while visiting the University of British Columbia.

Contents

1 Preface

This document is a masters thesis for the Department of Computer Science at University of Copenhagen. The work was done in the period of November 97 to June 98, at the University of British Columbia. The work is based on a project done by Jan Bækgaard Petersen, Yvonne Coady, and myself for the course CPSC513 at the University of British Columbia (UBC), in fall 97. Some of this work will appear as a tool paper in FMCAD98 [6]. The tool paper was coauthored by Alan J. Hu, my supervisor at UBC.

2 Introduction

Propositional logic is a tool that can be used to solve various different problems. To make effective use of propositional logic, some kind of data structure is needed to represent and manipulate these formulae.

Binary decision diagrams (BDD) [1] have become the dominant data structure for representing propositional logic formulas in formal verification and CAD. In practice, the memory required by large BDDs is typically the limiting factor for formal verification tools. In some cases, especially with the advent of dynamic variable reordering [10], run time is also a limiting factor. Accordingly, considerable research has been invested in more efficient BDD implementations.

An obvious line of research is to use parallel processing to accelerate BDD operations, but BDDs are not easy to parallelize. Nevertheless, several groups have produced parallel BDD packages for various architectures. Most of the effort have been centered around shared memory multicomputers, and we are here only interested in distributed memory parallel computation. We refer the interested reader to [13] for references to other work.

2.1 Introduction to BDDs

Since this project is not only aimed at readers familiar with BDDs we give a short introduction to the field[1]. A more extensive introduction to BDDs can be found in the survey paper by Bryant [2].

A BDD is a data structure used to represent boolean functions of boolean variables. One way to represent a function of boolean variables is by using a decision tree. A decision tree is a rooted tree where each internal node has two children labeled THEN and ELSE respectively, and the node itself is labeled by one of the variables the function depends on. The leaves contain

[1]this introduction is mainly based on the introduction given in [4]

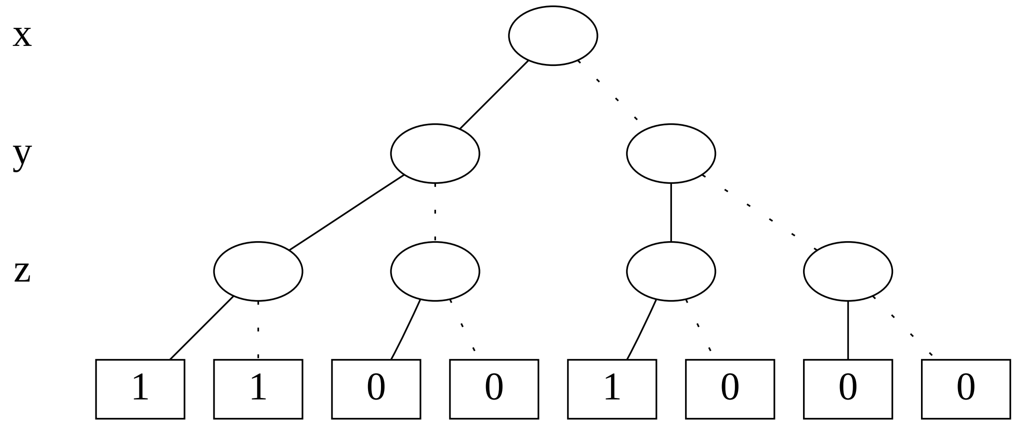


Figure 1: The decision tree for xy+yz. The solid edges represent the THEN branches where the variable is 1, and the dotted edges are the ELSE branches. The notes are not directly labeled with the variable, but all nodes on a given level are assumed to be labeled by the variable to the left of the diagram. If the individual nodes are labeled in a figure, it is to be able to refer to them easily.

the values of the function. To determine the value of the function, we start in the root, and then follow the edge to the THEN child if the variable in the node is 1 and the edge to the ELSE child if the variable is 0. The process is then repeated until a leaf is reached, and the value of the function is then the value in that leaf. Since, for a given assignment of values to the variables, the leaf is uniquely determined, a decision tree does represent a function. Showing that all functions of boolean variables can be represented by a decision tree, can be done by induction over the number of variables. Decision trees are not limited to representing functions with only boolean values, but we are only interested in functions with boolean values here.

BDDs can be constructed from the decision tree of a function in the following way: The decision tree may only have each variable appear once along any path from root to leaf, and the variables must always appear in the same order (figure 1). The tree is then reduced as much as possible by two rules: Any identical nodes (same label and same children) are merged (figure 2). If the two children of a node are identical, we delete the node, and change all in-going edges to go to the child instead (since both edges go to the same node, there is only one child) (figure 3). In practice BDDs are not created from the decision tree, but are built and manipulated in fully reduced form.

The resulting directed acyclic graph (DAG) is the BDD of the function.

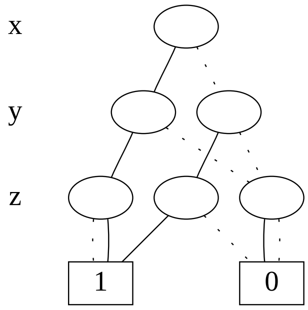


Figure 2: The function xy+yz, after merging identical nodes.

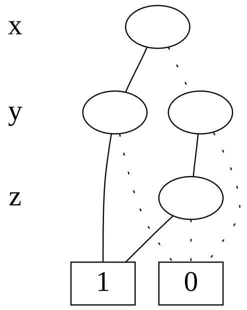


Figure 3: The BDD for xy+yz.

The correct term to use is actually *reduced ordered* BDD (ROBDD), but we will just use BDD. BDDs have several useful properties. One important property is that for a given order of the variables, a function has a unique representation. We will not prove this, but it can be done easily by induction over the number of variables. The number of boolean functions with n variables is $\Theta(2^{2^n})$, and it is only possible to represent $\Theta(2^{P(n)})$ functions with a BDD where the size is polynomial in n. This shows that most functions must have an exponential (or greater) size BDD. It turns out, however, that many interesting functions have fairly small BDD representations, so the real strength of BDDs is empirical rather than theoretical.

To save space and to make manipulation more efficient, the same BDD structure is normally used to represent several functions. This is done by having multiple roots in the BDD. Each root then represents a function (figure 4). Because of the canonical nature of the representation, this allows us to test if two functions are identical by testing if they are represented by the same node, rather than having to test for the BDDs being isomorphic.

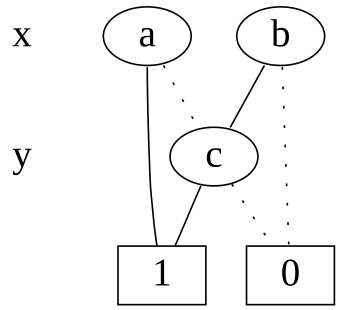


Figure 4: A multi-rooted BDD. The node a represents x+y and b represents xy.

2.1.1 BDD operations

Central to the use of BDDs is the ability to do operations on functions represented by BDDs. To do this we use the fact that variable substitution commutes with all operations.

$$(f \ <op> \ g)_{[y/x]} \ = \ f_{[y/x]} \ <op> \ g_{[y/x]}$$

We use this together with the shannon expansion.

$$f \ = \ if \ \ x \ \ then \ \ f_{x=1} \ \ else \ \ f_{x=0} \ \ = \ \ ite(x, f_{x=1}, f_{x=0})$$

Note that x=0 is used instead of [0/x]. From this we get:

$$(f \ <op> \ g) \ = \ ite(x, f_{x=1} < op > g_{x=1}, f_{x=0} < op > g_{x=0})$$

Now by choosing x to be the first variable in the order, of the variables in the root of f and g, finding $f_{x=1}$ and $f_{x=0}$ (called the cofactors of f with respect to x) is easy, as it is the children of f if it is labeled x, and f itself otherwise (likewise with g). In figure 4 the cofactors of a with respect to x is 1 and c, and the cofactors of c with respect to x are both c. The results of $f_{x=1} < op > g_{x=1}$ and $f_{x=0} < op > g_{x=0}$, can then be computed recursively, and the results of the operation is then the BDD with a node labeled x with the results as children, if the results are different, and just one of the results, if they are identical. The recursion ends when a base case for the operation ends, usually when one of the operands is a constant, or the operands are the same.

In order to make sure that the BDD does not contain two identical nodes, we need to determine if the node that is the result of an operation already exists in the BDD. To do this we keep a hash table of all existing nodes, and if the node already exists we use that as the representation of the function. The hash table containing all the nodes is called the unique table. When doing BDD operations, the same operation (with same arguments) may end up being done many times. To reduce the amount of work done, a result cache is kept. Note that the result cache is only used to improve efficiency, and results in it can be thrown away at any time without affecting the correctness of the computations.

In summary manipulating BDDs consists of applying the following step recursively.

1. Check for base case.
2. Do a result-cache lookup to see if this operation has been done already.
3. Get the top variable x of the operand BDDs f and g.
4. Find $f_{x=1}$,$f_{x=0}$,$g_{x=1}$ and $g_{x=0}$ (Cofactor by x).
5. Recurse on the left and right children.
6. If the result are the same return one of them.
7. Otherwise find/create the node *ite(x,left_result,right_result)*.
8. Update the result cache with the result.

The way functions are normally created is that one starts with BDD nodes representing just each variable. By performing operations on these, BDDs for new functions can be created, which in turn can be used to generate more complex functions, and so on.

2.1.2 Dynamic Variable Reordering

The order in which the variables appear have a large impact on the size of the BDD. Some functions have linear size BDDs for some orders, while other yield exponential size BDDs (figure 5). Hence, getting a good variable order is often essential. For some applications, good heuristics exist to choose such an order, but it is not always the case.

To cope with this problem, dynamic variable reordering has been suggested [10]. This process consists of changing the variable order of an existing BDD without having to rebuild it. When two neighboring variables in the order, changes places, the changes to the BDD are only local. A such exchange of two neighboring variables is called a swap. The order can then be changed by a series of swaps to get a better order. Dynamic variable reordering algorithms generally search through the orders by doing swaps in

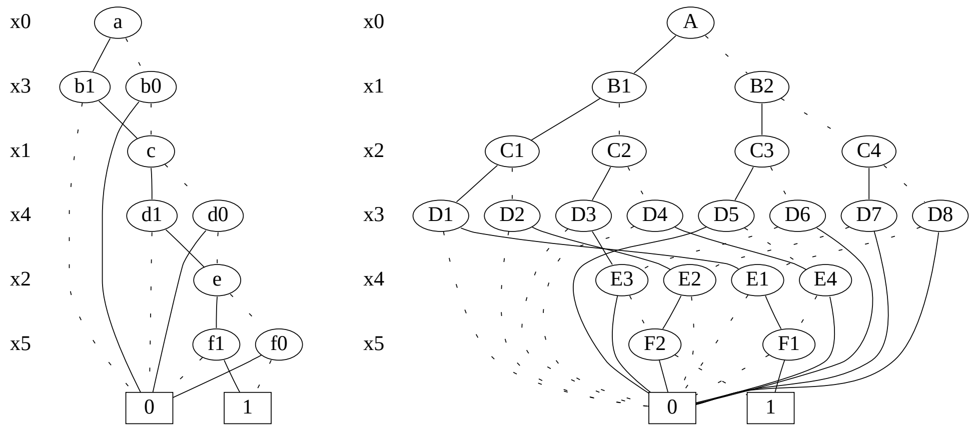


Figure 5: The BDD for the function that compares the first 3 bits with the last 3, using two different variable orders.

some order, keeping track of the resulting BDD size. After some time the search is stopped, and swaps are done to get to the best order seen along the way.

2.1.3 Some Properties of BDDs

We will now give a quick overview of the main properties of BDDs.

- The BDD representation of a function is canonical. This makes it easy to test if two functions are equivalent. Satisfiability (tautology) can be tested by comparing the constant function 0 (1).
- Many interesting function have compact BDD representations. Unfortunately other common functions like multiplication and division do not.
- Evaluating the function takes linear time in the number of variables.
- Given the BDD of a function, finding an assignment to the variables that evaluates to 1 can be done in time linear in the number of variables. This is because the canonical nature of BDDs ensures that all other functions than the constant one, has a path to 0. Using this it is possible to walk down through the BDD and get assignments to the variables, until 0 is reached (variables not encounter on the path are irrelevant).
- The procedure involved in doing operations on BDDs is simple. The number of computation steps needed is normally not much greater than

the size of the resulting BDD, and it is limited by the product of the sizes of the operand BDDs (assuming that the same computation is never done more than once).

2.2 Aim

The goal of this project is to design and implement a parallel BDD package. The package should be easily accessible, both in terms of system requirements and in terms of the knowledge of parallel applications required by the users.

Networks of workstations (NOWs) are currently the most widely available parallel computing resource, so we have designed our package for this type of platform. For this reason the package have been named BDDNOW.

The main reason for running an application in parallel is to make the application run faster. There are several different reasons why an application may run faster in parallel. The most common one is the availability of more processing power, which allows the computation to be done faster. Another possible gain is related to the available physical memory. The gain is only possible if the data does not fit into the memory of one machine, and if the amount of available memory increases with the number of machines used. The gain is achieved, because the parallel version does not have to spend (as much) time swapping pages in and out of the physical memory to fit all the data. A third possible gain has to do with the cache hierarchy. When the data is distributed across several processors, each processor will be working with a smaller sample of the data, and this can possibly increase the hit rate in the cache.

The primary aim of this work is enabling the use the pooled physical RAM of the networked workstations to work on BDDs too large to fit into the memory of any one workstation. We are also interested in gaining speedup over optimized sequential code, but this is secondary to being able to handle large BDDs.

2.3 Challenges

There are three main challenges in parallelizing BDDs.

- The basic operation in BDD manipulation is one step of the recursion, and the amount of computation involved in each step is small. This complicates effective load balancing, as the cost of moving work around is high compared to the computation involved in the work itself.

- BDD operations have poor locality of reference on large shared data structures.

- The memory requirements of BDDs are very high, so replication must be limited to get high utilization of the total available memory.

The memory requirement is what makes BDDs a challenging application to parallelize, from a theoretical perspective. Not much work have been done in parallelization, where both execution times and memory utilization are important factors.

2.4 Previous work

2.4.1 Sequential BDD packages

There have been quite a few different BDD packages produced since Bryant published the first in 1986 [1]. We will only give some details on one of these, namely the CUDD package, but we want to present some of the techniques used when implementing BDDs. The version of CUDD used for this project was version 2.1.2.

Keeping the size of each BDD node small is important, both because once the processor runs out of physical memory, it will rely heavily on swapping, and because the bandwidth to the main memory is a major bottleneck. For this reason the BDD is implemented with only one way pointers.

Another important issue is to make effective use of the cache hierarchy. The hash function used in the unique table is not perfect, so the unique table must contain collision chains. When traversing the collision chains in the unique table, the pointers to the children of a node are needed for comparison. By storing the pointer, used to link the collision chains, with the node, the link to the next node will often be fetched into the cache together with the pointers to the children. In CUDD, effort have been taken to make the size of a BDD node exactly 16 bytes, because that is commonly the size of a cache line.

One area where differences in approaches exist, is in the way the BDD manipulation is done. The recursive algorithm does not require that one result be computed before a second computation can be started. This can be used to do breadth first traversal of the computation, rather than the obvious depth first traversal achieved with standard recursion. The reason for doing breadth first traversal is that it increases the large scale locality of reference, which is important if the processor runs out of physical memory [7].

To reduce the number of nodes in the BDDs, the same node is used to represent both a function, and the inverse of that function. This is done by marking the in-going pointer with a flag that indicates if the function the

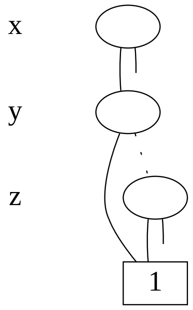


Figure 6: The function (x XOR (y+z)) represented by a BDD with negated edges. Dashed edges are negated ELSE edges. NB: the dashed edges did not turn out too well on this figure.

node represents, or the negated function, should be used. This introduces one ambiguity since

$$ite(x, f, g) = !ite(x, !f, !g)$$

This ambiguity can be solved by forcing one of the children of a node never to be marked as negated. In CUDD the THEN child is chosen as the one that never is negated. Marking a pointer requires only one bit, and to preserve space a trick is used. On modern computers all pointers have an even value, so the last bit of the pointer to the children is then used to indicate whether the function is negated. Figure 6 shows a BDD using negated edges.

2.4.2 Parallelizing BDDs

One prior attempt to parallelize BDD for distributed memory architectures is the work of Stornetta and Brewer [13]. Stornetta and Brewer mention networks of workstations as a promising platform for their package. However, we believe their package is more suitable for a parallel distributed memory supercomputer, as their scheme of randomly distributing nodes to processors (which gives excellent load balancing) results in very high communication overhead. Indeed, their reported results using a 32-node Meiko CS-2 showed speed-up over the sequential code only when the sequential code had exhausted its physical memory and was swapping to disk. They did test the package on the ISCAS85 benchmarks, but did not give sequential execution time, or execution times using only one processor, so we are unable to compare speedups.

Using distributed shared memory (DSM) is another way to implement BDDs on distributed memory architectures. This has been done by Parasuram et al [8]. The results obtained running on a CM-5 look good, but it is not clear how good the results actually are, as the paper compare wall clock time of non-optimized sequential code to “busy time” of the parallel version. The DSM system used is one written specifically for the CM-5, and since the CM-5 is a high performance multicomputer with much support for data sharing, their results may not be immediately generalized to distributed memory computers running DSM in general.

3 Design

This section describes the choices made in designing the package. Because the main focus of the the project is implementation, we will cover interesting implementation related details here too.

3.1 Priorities

Our Parallel BDD package has been designed according to 3 main criteria.

- Optimize for large BDDs, not small ones.
- Target platform is a NOW.
- Portability.

3.2 Portability

The package is developed to run on Parallel Virtual Machine (PVM). PVM has been chosen because it is widely available, and because we had some experience with using it.

To make the package portable on the programming level (as opposed to the architecture level) we have decided to conform to the API of an already existing BDD package. For this purpose CUDD has been chosen, because it is widely regarded as a very efficient BDD package. We have designed the package so the parallelism is transparent to the user, and the package is used exactly as CUDD would be used. This implicitly assumes that programs using BDDs spend most of the time doing BDD operations.

3.3 BDD Operations

The computation involved in doing a BDD operation can be represented as a DAG (figure 7), where the nodes represent one step in the algorithm described in section 2.1.1, and the edges represent recursive calls and cache hits. All base cases are sinks.

This DAG can be traversed in any order, which allow us do many of the operations in parallel. Though the operations are to some degree independent, they all operate on the large shared data structures, and this is what complicates the process.

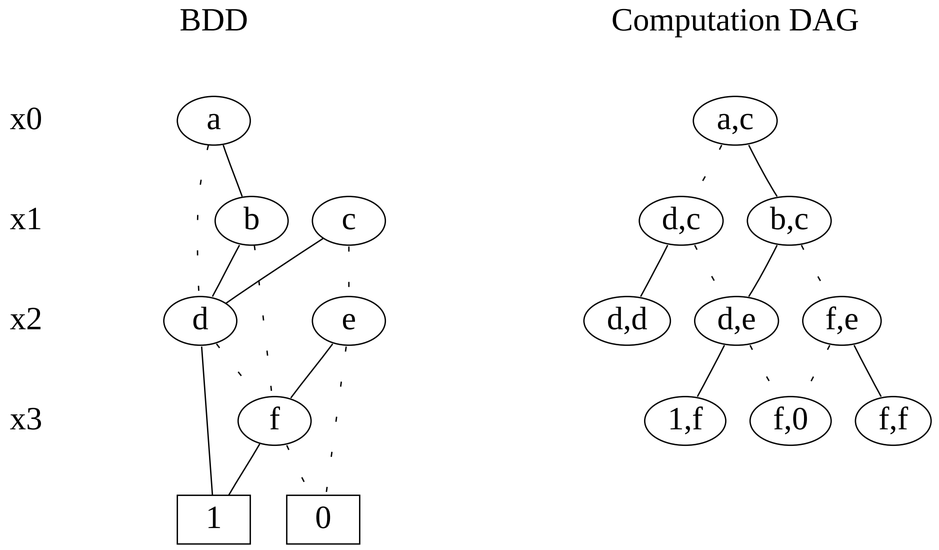


Figure 7: Example of BDD and the resulting computation DAG when doing a $<$ *op* $>$ c. Note that each node in the computation DAG is labeled with two nodes from the BDD, being the operands for that recursive call. For the computation DAG the variable levels to the left is the topmost variables involved in the call.

3.4 Data Sharing

To understand why BDD operations are fundamentally hard to parallelize, at least if little or no replication is used, we have to look at both the computation DAG and the connection with the BDD itself.

The computation can be divided into two parts, the downward part, where splits and recursive calls are made, and the upward part, where results are returned, and node lookups and creations are done.

In the downward part, the step that complicates parallelization is the splitting of the nodes with respect to the highest variable, because this requires reading the contents of the node. This means that there are two or three locations involved in the computation, namely the place the recursive call was made from, and the location(s) where the argument(s) depending on the highest index, is stored. It is worth noting that the two argument to a recursive call can only be related due to application specific conditions, since they are internal in each of the two original BDDs that the user program is doing an operation on, and thus any relation (other than their index being greater than the last index split on), must come from the user program.

The complications in the upward computation is in the location and creation of BDD nodes. Determining whether a node has already been created is only hard, because the limited replication means we can only store the hash table that contains the information, in one place. Node creation is hard even if we allow replication. Either we must ensure that the same node cannot be created by more than one process, (which would result in the node not being unique), or we must use some kind of optimistic algorithm, and deal with conflicting creation of nodes, when they are detected. Though having different processors create the same node would probably be rare enough to make an optimistic algorithm feasible, the fact that no information of the predecessors of a node is stored makes it hard to track down the locations of the corrupted data.

3.5 Node Distribution

To cope with the data sharing we use the same locality of reference that is used when doing breadth first BDD manipulation [11]. The observation is that for any of the recursive calls all access to the shared data structures is limited by the highest index of the arguments in the following way:

- Only entries from the computed cache with the topmost index equal to the topmost index in the call, can result in a hit.
- Only nodes with index equal to the topmost index are split.
- Any node lookups/creations will have index equal to the topmost index.

Using this observation it is obvious that limiting the number of processors working at a given level, will cut down the data sharing. We have decided to use this locality fully, and only have one processor work at each index (level). The advantages of this are that it is simple, and it has a fairly low communication requirement. The disadvantage of the scheme is the lack of flexibility in load balancing. This also limits scalability, since the level containing the most nodes must fit on one machine, and the level with most work must also be handled by only one machine. The choice is mainly governed by our target platform, since a NOW normally has a high communication overhead and the number of machines available is normally also limited, so the lack of scalability in the high end is not a primary concern.

3.6 Memory Usage

The CUDD package has fairly low memory usage, and we have largely maintained this in our package.

To be able to determine the topmost variable in a call, we use a representation where all pointers are annotated by the index of the node pointed to, just as it is normally done in breadth-first BDD operations. This increases the size of each node slightly. Offsetting this increase, we eliminate the reference count. Keeping track of reference counts in a distributed graph is costly, and garbage collection should be done by using mark and sweep instead.

Normally a BDD is just represented by a pointer to a BDD node. However, in a parallel implementation, having a pointer to memory on a different machine is not good design and may cause problems running on heterogeneous machines. It also makes moving indices from one machine to another costly, as it is not possible to guarantee that a node will get the same address on the new machine as it had on the old. Alternatively, one could modify the virtual memory system to create a linear address space over all the machines, but this too would be difficult on heterogeneous setups. To avoid these problems, we introduce a different addressing scheme. Within each index we create an array of arrays of nodes. We refer to the individual arrays as *pages* and to the array of page addresses as the *page-table* (figure 8). The double indirection structure gives finer granularity over memory allocation. Each node is uniquely identified by the triple (index,page,number within page), and this gives us a machine independent addressing. The pair (page,number within page) will be referred to as the address of a node. There is some waste of space, because node space is allocated per index, as each index will have a full page allocated, where only some of the space may be in use. There is also an extra memory reference required for each node access, but this reference will be to the page-table, and since this is fairly small and often used, it should have good locality, and since accessing nodes in general has poor locality, the time it takes to access a node, may not be significantly increased. The nice features of this scheme are that it is clean, moving indices is easy, and nodes require exactly 8 shorts to represent. The 8 shorts is good, because assuming that the the length of a short is a power of two, the size of a node will be a power of 2 as well, and we can then expect nodes to align nicely with cache lines. When talking about actual node sizes we will assume that shorts are 2 bytes, and thus a node is 16 bytes. For a 32 bit machine where pointers are 4 bytes, this is the same as the node size in CUDD.

The use of the hash tables is linked closely with the memory use. In CUDD hash tables are arrays of pointers to nodes, and each node then has a pointer to the next node in the collision chain. Since the collision chains are only local, it would not break with the principle of not storing pointers to other machines, to use the same representation in BDDNOW. The problem with using pointers is that looking up in the table would give the pointer to the node in question, and converting the pointer to the address is not that simple. It may be that it can be done easily with the right guarantees from

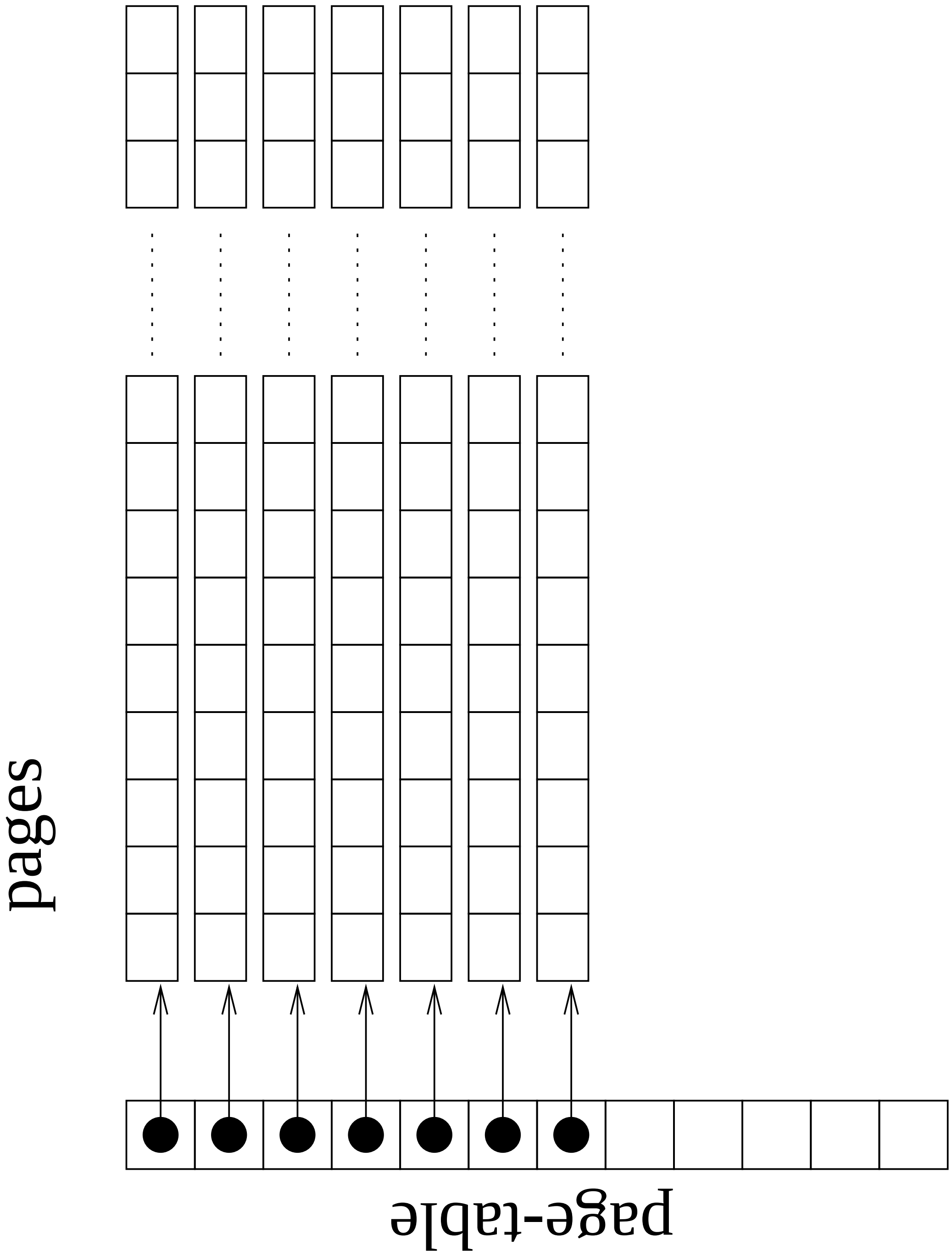


Figure 8: Illustration of the memory allocation sceme.

the virtual memory system, but we want to avoid that, as it leads to too much platform dependence. Instead we link the collision chains using the addresses, which means that we need to do the address to pointer conversion for each link. Without guarantees from the memory system, it is still possible to convert pointers to addresses, but we have to find the page by looking at what interval the pointer belongs to and then find the number within the page from the pointer and the start of the array. We believe it is cheaper to do an address to pointer conversion at each link, than to determine what interval the pointer belongs to, but this will depend on the average number of links followed when doing node lookups.

3.7 Load Balancing

Having decided to keep each level at one processor, the only possible load balancing is in the assignment of levels to processors.

There are opposing arguments for how this distribution should be done. When the computation DAG is traversed, the main workload will be near the top in the beginning, and then move down, and in the end the workload will move back up again. This suggests that to distribute the work, neighboring levels should be on different processors. On the other hand by keeping neighboring levels together, less communication is needed, since most recursive calls have a topmost variable close to the one last split on. We find keeping communication down is more important, so we have chosen the latter solution. We have not closely studied prospects of interleaving levels, but ran a few tests with it, and did not get good results.

To maximize memory utilization, we have decided to distribute levels so that each processor gets an approximately equal number of BDD nodes. For BDDs that use up most of the available physical memory, this is by far the best choice, as the poor locality means that once a processor runs out of physical memory, it will rely a lot on swapping.

In order to get a better understanding of the load distribution, we record how much work is done at each level (i.e. where the top variable is on that level). Some of these plots can be found in appendix A. The main observation is that the main workload lies a little later in the order than the peak in nodes. This suggests that distributing levels only based on nodes is not the optimal solution, when there is some memory to spare. As we are mainly focusing on large BDDs, and due to time constraints, we have not given priority to implementing a load balancing scheme that takes both the amount of available memory and the work distribution into account. We have used the observation in the way that when rounding, levels are given to processor first in the order, since it will normally have the smaller workload.

Another interesting area that we have not at all looked at, is how to make effective use of heterogeneous setups, where processors may have different speeds and physical memory available.

Load balancing has to be done dynamically, as it is not possible to know the node distribution beforehand. As long as the BDD is small, there is no benefit in distribution, so up to a certain size, we keep all nodes on one processor. After that we distribute as explained above periodically. At the moment first load balancing is done at 100000 nodes, and then every time the BDD doubles in size. These parameters may need to be tuned.

3.8 New Data Structures

To handle the parallel processing, we have added two new data structures.

3.8.1 Work Queue

In a depth-first traversal, keeping track of available work is done implicitly by the call stack. To some degree this would also be possible in the parallel implementation by recursing if the call is local. It can unfortunatly not just be done the same way it is done in depth-first, where the recursive call actually return the results. This is because the call may very well not be able to complete locally, and the work related to the second of the two children cannot be started before the result of the first one is available. The result of this is a low level of parallelism. The easiest way to handle the problem, is to use a work queue. With a work queue we can also do explicit scheduling. At first we just implemented the work queue as a stack, because this ensures that all entries in the work queues are direct children of some ancestor of the node being processed so the work queue never grows bigger than the number of variables managed by the processor. As we will describe below we later altered the stack implemention slightly.

3.8.2 Computation DAG

When a processor does a recursive call, it has to decide whether the work should be done locally, or passed off to another processor. If it does pass off the work, it can go on doing other useful work, but some information about the state of the computation must be stored, so the step can be completed when the results are both available. This requires representing the computation DAG in some way.

The solution we have implemented does this by using a tree, and by overloading the functionality of the cache. We introduce what we call the

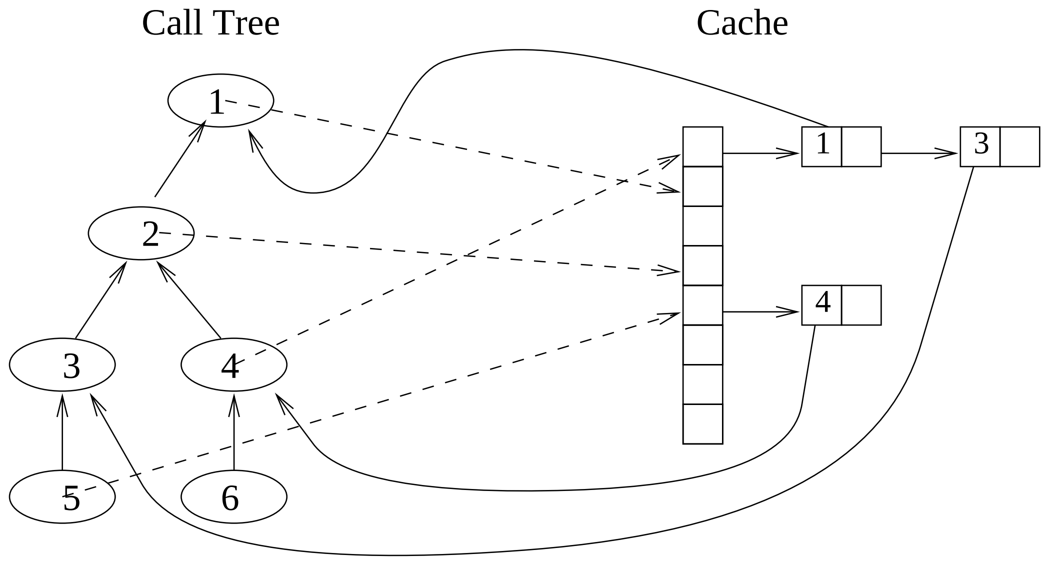


Figure 9: The tree and the computed cache together form a DAG, since multiple parents of a node can be reached through the list of waiting processes in the cache.

call tree. Its nodes correspond directly to the nodes in the computation DAG. A node can be in different states, depending on what results of the recursions are available. When both results have arrived, the node lookup/creation can be done, and the result be passed on.

When doing anything but a depth first traversal of the DAG, we have to keep track of operations that have been started, but not yet completed, to avoid starting the same operation more than once at the same time. Our idea was that we can use the cache for storing both available results and to indicate that an operation is already being done. Instead of a cache lookup resulting in either a hit or a miss, it can then also result in a waiting hit. We then add a list of nodes on the call tree, waiting for that particular result, and when the cache entry is updated, notification is send to all the call tree nodes, waiting for that result. Registering what work is in progress is easy. All operation starts by doing a cache lookup, and if it misses, we simply note in that entry that the operation is in progress.

The net result of the tree and overloading the cache is the computation DAG. Figure 9 depicts how this look.

The scheme works well as long as there is a lot of cache. However when the cache is small, there are effects that may make this scheme a bad choice. Since the cache entry is reserved when a computation starts, a result of a computation that is needed later in the computation may be discarded. A worse effect happens when an operation tries to register in a cache entry that is already used by an operation in progress. Since the entry can only hold only one operation, we can not keep record of both operations being started. If the same result is needed again later, the operation can be started multible times, which especially is a problem if it causes the same operation to be sent to another processor multiple times. When we made the design, we where expecting the cache to be large, however, this may very well not be the case in general. When looking at the manipulation algorithm (section 2.1.1), we observe that each recursive call has associated with it one cache lookup/update. There are more recursive calls than BDD nodes since each call generate a most one node. By this analysis if the cache is to be considered big, it must be a significant size compared to the BDD, so for large BDDs, having what can be considered a large cache may take up too much space. CUDD suggest a limit of 100000 cache entries, which we have adopted. For some of the test cases there are more than 10 million calls, so that assumption that the cache is large, does not hold.

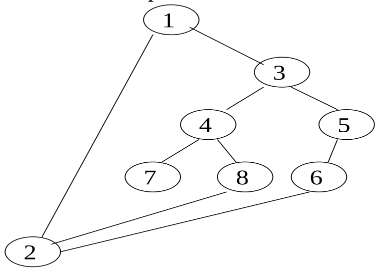
We discovered this effect late in the project, so we have not implemented anything to deal with the problem. Possible solutions would include having a separate hash table for operations in progress, adding collision chains to the cache, so several ongoing operations could be stored in the same entry, or having a secondary cache for conflicting ongoing operations.

To implement the quantification operations, some changes are needed to the upward computation phase. This is because quantification does AND or OR the two results, rather than just a node lookup. The changes required are not significant. We just add possible states to the nodes in the call tree.

The computation DAG can grow quite big, and since memory is a limited resource, limiting the size is important. As soon as the computation in a node is done, the node can be freed, so it is only the part of the computation that is in progress that contributes to the size. At the cost of some performance, it is possible to limit the size of the DAG. This can be done by just stopping the downward computation when the DAG reaches a certain size, and wait for external results to arrive, freeing up nodes in the DAG. It is obvious that some minimum size of the DAG is required to guarantee that there are nodes in the DAG only waiting for external results, so a deadlock situation can be prevented. How big this limit must be will depend on the number of variables handled on the processor (n). At first we thought the use of a stack to store work should give a depth-first traversal, so the number of nodes should only be n. Unfortunately this is not the case in DAGs, which is illustrated by an example in figure 10.

The problem is that operations can be waiting for a result of an operation waiting in the work queue, and a limit on the number of DAG nodes that

Computation DAG



Execution History

Node processed	Children Generated	Work Queue
1	2,3	2,3
3	4,5	2,4,5
5	6,base case	2,4,6
6	2,base case	2,4
4	7,8	2,7,8
8	2,base case	2,7
7	out of space	2

Figure 10: Example of how the computation DAG can blow up without waiting for external results. Node 6 and 8 generate waiting hits on node 2, and cannot complete before 2 has completed. If the maximum size was eight nodes node 7 would be unable to generate children. The base cases could just as well be cache hits or computaionse that could complete.

can be waiting in this way, is in the order of 2^n. To deal with this problem is simple. We just modify the stack used to manage the work queue. When work is pushed, it is placed either on top or as the 2nd element, in a way so that the top element has the higher topmost index of the operands involved. This solves the problem because the topmost index involved is always increasing when we move down in the DAG, so sorting the two topmost elements at push, is really only sorting the two calls generated at one node, and thus one child will be completely processed, up to the calls that can not complete, because they are waiting for other operations, before the other child gets processed. This way all entries in the work queue will still be direct children of some ancestor of the node currently being processed. Always processing the child with the higher index, ensures that the work in the work queue has a topmost index lower or equal to the one being processed. Since the operations generated at the node being processed have topmost index higher than the one being processes, it cannot already be in the work queue.

We discovered the potential deadlock problem late in the process, and sorting the top stack elements on push was the easiest solution we could think of to deal with it. The problem is itself mainly of theoretical nature, as it requires some nodes in the DAG to have a very high number of parents. A better solution would be to keep track of the number of jobs send to other processors, and if the DAG grows too big and there is no jobs outstanding, then find some way to resolve the deadlock. This is better because the computational cost is paid when the deadlock occurs, instead of a cost per operation.

3.9 Communication

As with any parallel application, we need to consider some communication issues.

When a processor determines that the topmost index of an operation is managed by another processor, the operation (job) must be sent. To send a message for each job would generate far too many messages, so we need somehow to batch the jobs.

Our communication strategy is based on two assumptions: Most of the jobs will be sent to the processor managing the levels just below, and keeping the number of messages down is more important than keeping the total number of bytes sent down. Now on this basis we have chosen always just to send jobs to the processor below, rather than direct to the processor that will handle the job in the end. The reasoning is that since a processor is sending a lot of jobs to the one below, the jobs going further down can be batched with the ones that is going down one step anyway. The design also gives a good abstraction, as each processor just views the next processor as a machine doing operations on the remaining variables, and returning the results.

There are several different approaches to how jobs can be batched. We have chosen a simple strategy, where the sender sends the jobs once it has collected a certain number. After trying different sizes, we ended up setting the limit at 1000 jobs. We also tried to let the limit change during each operation, so early we had a small limit to get some work down to the lower processors fast, but this did not seem any faster than just batching the same number all the time. What size works well may be very network dependent. Since the total number of jobs need not be a multiple of the number of jobs batched, we implement a flush opeation to force the jobs at the end of an operation, sent by the user program, to be sent down, and later to force the results to be sent back up.

Other approaches to batching include trying to maintain some idea of how much work the lower processor(s) has, to send more when calculations show more work is needed below, and having a demand driven batching, where work is send when the processor(s) below request it.

There are a lot of jobs flowing between machines, but the number of bytes send per external operation including reply is low (around 30 bytes), so the effective bandwidth required should be reasonable. In our experiments (see section 5 for specifications), the maximum effective communication rate between any two processors was around 10 Mbps.

To cut down on the number of jobs sent, we observe that testing for base case in an operation does normally not require reading the contents of the nodes involved. Instead of testing base cases as first step in the recursion, we then test it before the recursion is done. Similarly we move the cache lookup up before the recursion, and keep results for level i at level i-1, because all calls must come from there or above.

When work arrives at a processor we read all of it and store it in the work queue. This has two advantages, namely that work intended for processors lower in the order, can be forwarded faster. furthermore it allows us to do more work management, if we find that ordering the work by some criteria gives better performance. The cost is of course that it takes time to move it from the communication buffer to the work queue. Moving it from the buffer also makes it easier to receive replies, as we can reuse the receive buffer.

3.10 Superscalar Execution

The way the user program sends tasks to the processes running the BDD package is essentially no different than the way the processes send jobs to each other. Since internally many jobs are done at the same time, it would be fully possible for the user process to start new jobs before the old ones have completed.

This should allow more work to be available to the processors and should result in less idling. Ideally the package would have the calls to operations terminate with just a hook to the result. The package could then determine which call did not depend on the results of other calls, and use this to have several operations executing at the same time. The calls need to terminate to allow the sequential user program to issue more operations to the backage. This would require comparison of BDDs to be done via a function call, as comparison cannot be done before the results of the operands are available, so the current simple pointer comparison no longer makes sense. The process of sceduling the BDD operations would in many ways be similar to instruction sceduling in superscalar microprocessors.

Doing such superscalar execution is beyond the scope of our project, but integrating it with the package should be simple.

3.11 Interface to CUDD

For the main part, making BDDNOW conform to the CUDD API is simple; however, there are a few problems with the way the user program handles BDDs.

In CUDD, a BDD is defined as a pointer to a BDD node. The problem with this is that since it is defined as a pointer to something it is impossible to change it to the representation we use internally without changing the

API. Even if we could change this, BDD comparison is done using a simple '==' which requires the type to be simple, and our triple representation is not a simply type. Finally keeping track of what nodes are currently in use at the user level, is done by explicitly referencing and dereferencing the nodes, and we must use these calls to keep track of live and dead notes so garbage collection can be done.

To handle the problems we have chosen to put a shell around BDDNOW, so that seen from the user, a BDD node is still a pointer. This is done by allocating what we call *user nodes* in the memory of user program, and in these user nodes, we store the triple identifying the BDD in the internal representation. Note that only BDDs that have been returned to the user will have a corresponding user node, so the memory use for this should be manageable.

To allow BDD comparison to be done as a simple pointer comparison, we maintain a hash table of the user nodes, making sure user nodes are unique, and, as is custom, we use the last bit in the pointer to indicate negated BDDs.

To keep track of live nodes, we simply store a reference count together with the triple. As opposed to keeping reference counts in general, this only requires local updates, since user nodes are stored in the user program memory, and they are only affected by explicit reference call from the user program.

One last problem with the interface is that CUDD gives explicit access to the structure of a BDD node, and it is theoretically legal to read and write these fields. We will give only very limited support for this, that is, it legal to read the reference count and the index, and changing the reference count will probably produce the same effect as in CUDD, but the children are not accessible. There are macros to access the children in CUDD, and we expect that in future versions of BDDNOW, these will be implemented as functions. The reason why we have not done this yet, is that it involves quite a bit more work the the sequential version, as we need to contact the processor that stores the node to get the child, and then find/allocate a user node for it. The only reason to allow reading index and reference count is to allow more CUDD programs to run without modification.

4 Dynamic Variable Reordering

Dynamic variable reordering [10] is becoming increasingly more important, and a good BDD package must include this. The basic operation in reordering is the swapping of two adjacent variables. Though doing each swap parallelized would be interesting, we have concentrated our efforts in doing the individual swaps locally, but making it possible to do several independent swaps at the same time.

4.1 Reference Counts

One of the harder parts in a distributed reordering is keeping track of live nodes. One solution would be just to ignore the problem, and do a garbage collection at the end of the reordering, however this is not good enough for dynamic reordering, as we need a close count on the resulting size of the BDD to determine when a good order is found.

If we instead had an algorithm, that given a BDD, found a good order without doing swaps, we could just do the swaps required, and then garbage collect after that. Unfotunatly we are not aware of any good algorithms that can do this.

Though the effects of a swap are only local, and only nodes on the lower level involved in the swap can die, determining what nodes die as result of the swap cannot be determined locally without storing extra information with the nodes. The only actual information required is whether a node is referenced only from the level directly above. This information can fairly easily be precomputed for the BDD, but maintaining it after each swap cannot be done based only on local information.

The use of reference counts is the normal way BDD implementations keep track of which nodes die during reordering. Though this is fairly easy solution, it has the problem that the effects of a swap is no longer only local, as the reference count of nodes lower in the BDD may change.

The solution we have come up with to cope with these problems, is to use a modified version of reference counts. The observation is that if the BDD is divided at a given level, the nodes referenced across this division, will remain referenced, as long as no swaps are done across the division. By putting a number of divisions in the BDD, a number of blocks are formed. Nodes referenced only internally in a block then have reference counts, while nodes referenced from outside are just marked as such. The advantage of this scheme is that we get a number of independent blocks, and no information need to flow between them as long as swaps are done internally in the blocks. The disadvantage is that swaps across divisions become costly. Though it would be possible to have a block span over several processors, we have limited our work to cases where the divisions correspond to the machine boundaries.

Adding the reference count increases the size of the nodes. This is in general bad, and it is especially a problem as the nodes are designed to be 8 shorts to make them align with cache lines. In this case our choice of node addressing give a possible solution to this. We can just allocate another array of arrays of space, and use the triple that identifies the node to also determine where the reference count is stored.

4.2 Index Problems

Some of the problems with doing reordering in our case are the same as when doing breath-first BDD manipulation, as they spring from having the index on the in-going pointer, rather than in the node itself, and from wanting to keep nodes with the same index located on the same pages in memory.

The problem is that during reordering, nodes change indexes and they change in a way that is dependent on the contents of the node. There are basically two ways to deal with this problem, either we can use forward pointers, where, in the old location of the node, you leave a forward marked with the new index; or you can add the index to the node, and update all pointers with the correct index at the end of reordering.

4.2.1 Forwarding Schemes

There are basically two different ways of using forward pointers. After a swap, nodes with indexes corresponding to the higher of the levels before the swap will have either their old index or the index of the other level involved in the swap. When choosing a forward scheme, we can assume one of the two cases, and nodes for which the assumption does not hold, will then be replaced by a forward pointer.

If the assumption we make is that nodes do not change index, we get the scheme described in [9]. Figure 11 shows a simple example of this. The advantage is here that there is no change in indexes of pointers, and new nodes can be allocated on the pages where they should end.

The scheme has some problems with keeping a canonical representation of nodes. At first glance it would seem that all nodes will be accessing the new node via the forward pointer, but during the swap process new nodes are created and it can happen that a node identical to the node forwarded to is created, in which case we end up with both direct and indirect access to the new node. In turn this can lead to having two different (non-forward)

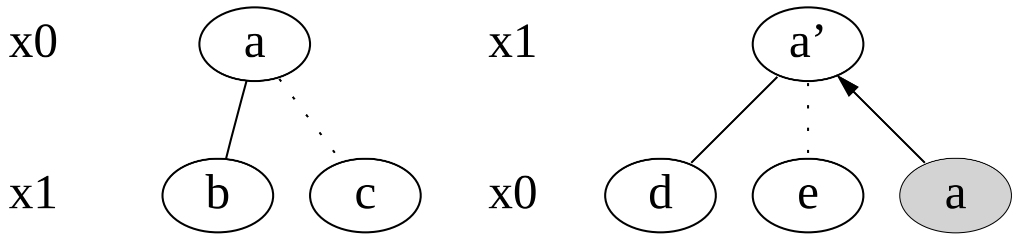


Figure 11: The result of a swap using the first forwarding sceme.

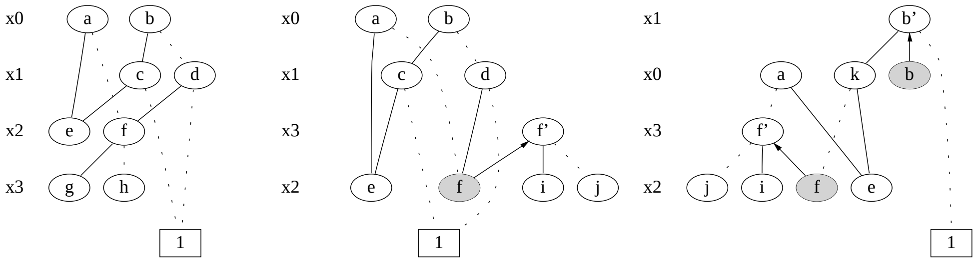


Figure 12: Example of how updating the children of a node (node a in the last swap) when swapped down can result in two nodes (a and k) representing the same function. The first BDD is the initial BDD, the two next is the BDD after swapping x2 with x3, and x0 with x1 respectively. Note that nodes that do not change labels, do not change physical location, and the children only change if they are shown. Nodes without children shown (that are not forward nodes), are assumed to have children further down in the BDD.

nodes represent the same function. To solve this [9] suggests that whenever a node is swapped down, any children that may be forward pointers are followed to the real node in the end. This, however, creates a new problem (which is overlooked in [9]), as is illustrated in figure 12. Solving the problem requires ensuring all nodes involved in the swap have children with a canonical representation. For the most part this requires looking at the children of the nodes, and following any forward pointers to the end, in general a fairly costly operation because of the poor locality of the BDD. For references crossing the boundaries introduced in 4.1, no new references have been created across the boundary, so the old pointer will still be canonical. Hence it is still possible to do swaps without needing to send information across boundaries.

The other possible forwarding scheme, where wa assume that nodes alway change index, have some slightly different properties. Since it is infeasible to actually change the index on all the pointers, we say that after the swap both indexes correspond to the same level, and that they are actually just

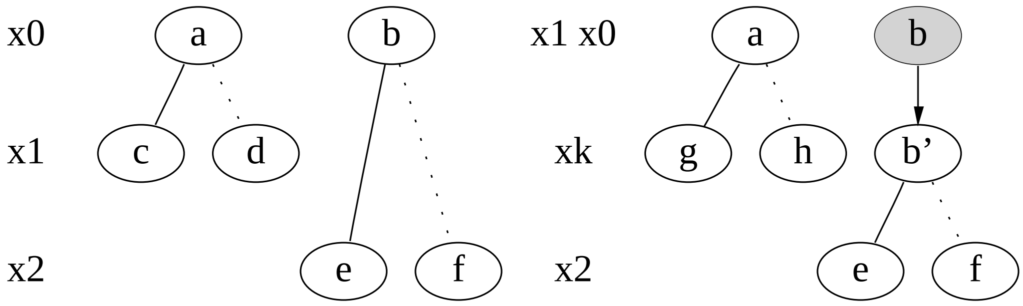


Figure 13: The swap process using the 2nd forwarding scheme

different representations of the same thing. This means that the lower level after the swap needs a new index, but this is straightforward, since by default all nodes that had references to that level now have references to the level above. The level thus only consists of nodes created during the swap, and it is simple to give these references the new index.

As with the other scheme, we need to consider what happens when a forward node is involved in a swap. To prevent the same function from being represented by different "real" nodes, it is sufficient (but not necessary) that across a division at any level there is a canonical representation of the nodes below. This property can be maintained at each swap by the following 2 rules:

- A forward node being swapped down is replaced by the node forwarded to if and only if the node forwarded to is located on the level being swapped up.

- Both cofactors of a forward node being swapped up is equal to the node forwarded to.

To see that this preserves the property, observe that no nodes except on the two levels being swapped are read, so only canonical representation of nodes below can be accessible. No nodes above the swap are changed, so these reference remain canonical. After the swap all references crossing the division between the two levels in the swap either go to the levels below the swap or directly from the upper to the lower level, but in either case they are still canonical.

The difference from the first forward scheme is that all nodes created are only referenced from the level directly above, so the problem, where a node is created that already has a forward pointer to it from somewhere else in the BDD, is avoided.

An additional advantage is that find/create of nodes is done only at the new levels generated, and thus once a level has been created the unique table is no longer needed, and the space can be freed. A disadvantage of this scheme is that a new index is used for each swap, and since there is a limited number of indexes available (normally 2^{16}), there is a limit to the number of swaps that can be done without cleaning up.

4.2.2 Adding Indexes to Nodes

The alternative to the forwarding schemes is to add the index to the nodes, and change the index as required. There are some complications with this as well, and though these have more to do with implementation details we will still discuss them here.

The index can be added the same way we added the reference count. We can either create a new structure for it, or have the one created for reference counts also contain indexes. For simplicity we store it with the reference count.

One problem has to do with the way we identify nodes, and the unique table. Since node identifiers are only unique within each index, the index on the in-going pointer is still needed to identify a node, even if the index on the pointer no longer corresponds to the index of the node. As mentioned earlier each node contains a next pointer for linking it in the unique table. Because there is a different table for each index, the next pointer does not include the index. As the in-going pointer now can have any index, the smaller next pointer is not sufficient to link the nodes in the unique table.

There are basically two solutions to this: either add an index to the next pointer, or make it possible to identify nodes with the space available for the next pointer (2 shorts). Due to time constraints we have only tried adding the index. The next pointer index is added together with the reference count and the node index. This solution has one serious drawback, namely to follow links in the hash table, a reference to two different memory locations are needed, and again because of the poor locality of the BDD this may slow lookups in the unique table down considerably.

Another problem with adding the index to the node is that it complicates moving levels between nodes during reordering, as to allow swaps across divisions. Though in the end we have decided against implementing this, we have tried not to rule out the option.

4.2.3 Comparing Strategies

In [9] the authors compared the first forwarding scheme to adding indexes to the nodes, and concluded that the latter was more efficient. However, because of the problems with the triple representation, the picture may look different in our implementation.

We encountered an entirely different problem that led to the abandoning of both forwarding schemes. The problem has to do with memory usage. One of the properties of the forward schemes is that they keep nodes with same index on same pages. This becomes a problem when many indexes at some part of the reordering pass by the fat part of the BDD. These levels get a lot of memory allocated, and even when the indexes decrease in size, the result is sparsely populated indexes and thus a high memory use (and even worse locality). We did not examine whether the live nodes tended to be on the same pages, and thus by keeping count on the number of live nodes on each page, it would be possible to reclaim pages that died totally, but we see nothing that would indicate that such a behavior is to be expected in general. The second of the two forwarding schemes is even worse than the first, since all new nodes are allocated with new indexes, and thus on new pages, so the only way memory may be reused is by reclaiming whole pages.

Because of the poor memory performance of the forwarding schemes, we have choosen the solution of adding indexes to the nodes.

4.3 Representation

In general it is interesting whether the way nodes are identified should be changed during reordering. Though the triple representation allow us to store different parts of the node at different locations, but still use the same identifier, it adds computational costs when adding the index to the nodes, and it is part of the reason for the high memory overhead of forwarding schemes. Some of the problem is that the index is used as part of the address; alternatively we could have it uniquely determined by the address (the last two coordinates of the triple), at least during reordering. Because of time constraints we have not tried other representations, but we will nevertheless discuss some alternatives.

One possible solution is making the address globally unique all the time. This would require that some central administration of available page numbers be kept. The amount of administration required is probably not big, but it limits the total number of nodes that can be represented. Assuming a short is 2 bytes, the number of nodes that can be represented is 2^{16} times the number of nodes per page (currently 2^{12}). If the address is unique per index this is the limit for each index, where as when addresses are made globally unique, it becomes the limit on the total number of nodes. Of course the nodes per page may be increased at the cost of some memory overhead, so it is not completely clear if the limit is too constraining. This solution could very well be integrated with the virtual memory system, but then it is beginning to resemble a DSM system, and it may be beneficial to actually use a DSM system to implement it. It may also be possible to get rid of the index on the pointer, and instead maintain a table of page to index mappings.

Making addresses unique within each processor cannot be done in general, if we want to be able to move levels without updating the entire BDD. However, during reordering we already have to update all nodes in both blocks if we want to do a swap across a division. Since there will be nodes in other parts of the BDD with references to the nodes we want to change, we would like to make the changes in a way that given the old address (and knowing it is old) we can still find the new one. An obvious solution is just to leave forward pointers at all the old locations, or at least allocate a structure containing forward addresses, where the old identifier can be used in the same way we added reference counts and indexes to nodes. This, however, has a high memory overhead. A better solution is to just concatenate all the page tables, into one new big page table. This would just require storing for each index the location in the new page table, the old page table starts, and converting an old address to a new just requires adding this number to the page number. As with making addresses globally unique, this does limit the number of nodes that can be represented, but it may be more tolerable here since it is only a limit per processor. It may not be possible here to get rid of the index on the pointers, as they are still needed to determine whether a reference is local or not. For the reorder solution where indexes are added to nodes, this is probably a better representation than the current triple representation.

4.4 Cleaning Up

The simple fact that we require pointers to carry the index of the node pointed to, require that we do some kind of cleanup at the end of the reorder phase. There are two approaches to doing cleanup, either we do only the necessary clean up, or we clean up as much as possible. We have chosen the latter as this makes handling the BDD operations simpler, and it allows to reclaim more memory.

Doing a thorough cleanup requires moving all nodes to a new structure. Updating the references as they change is infeasible as it requires going through the entire BDD to find the references. This means that the new addresses have to be stored somehow, and this results in a memory overhead. It should be possible to keep this overhead to a minimum, as it is only needed at the time where memory from dead nodes can be reclaimed, and if there is a significant node reduction as result of the reordering, the net use of memory will not increase. It may however be that there is no decrease in the size, in which case there will be a memory overhead.

If care is not taken, the memory use may end up doubling, by representing both the old and the new BDD structure at the same time. This can be avoided by moving nodes one page at at time, and then freeing the page, while just keeping information of the new location of the nodes.

When cleaning up we have to look at how new locations are sent to other processors. We know exactly which nodes are externally referenced, since we needed it for reference counts anyway, so we need only send updates of these nodes. We do not know from which processors they are referenced though. Keeping with the principle of communication with the neighbors only, we have chosen only to send updates to the processor directly above. It is then the responsibility of the processor above to keep track of which of these nodes are referenced further up, and pass these updates on. The reasoning for this is that references that cross all the levels are rare, so keeping track of them is not too expensive, and most of the nodes referenced across a processor will also be referenced from the processor it crosses. It also simplifies the update procedure as each processor will receive updates from only one place. The format of how updates are sent also leaves a choice to be made. We can either send both information about new and old addresses, or since the processor above knows exactly what nodes it is referencing, we can just send the new addresses in a well defined order. The choice is between communication and either memory use or computation. Because of the way we handle external references we are storing the information anyway, so we clearly only send the new addresses.

4.5 Swapping Across Divisions

Throughout the project we have tried to keep the option open that we might want to implement ways of swapping across divisions. In the end we decided against this because of the complexity, and because there is an alternative, as we will describe below. It may be that some of the designs could have been done better if swaps across divisions are ruled out from the beginning.

Not allowing swaps across certain boundaries seems too restrictive, so that is not a good alternative. However before reordering is started we can move the levels around, so the levels we want to swap are on the same machine. This will probably not give enough freedom in general, but the solution lies in doing the reordering in more than one pass, and move levels between passes. This would mean a reordering algorithm would have to have the following format.

1. define blocks
2. do swaps within blocks
3. cleanup

This allows arbitrary sequences of swaps, even if each block is limited to two variables, because we can just define the blocks so the two levels we want to swap are in the same block.

The disadvantage of having to do cleanup and redefine blocks to do swaps across divisions is that cleanup requires synchronizing all the processors. The cost of the cleanup operation is also fairly high, and again this has to be done at all the processors.

4.6 Parallel Reordering Algorithm

As described we have made it possible to do several independent swaps in parallel as long as they stay within same block, and it is possible to redefine blocks along the way, at the cost of cleaning up BDD.

An interesting problem is the need of an algorithm that takes advantage of the fact that several independent swaps can be done in parallel. Searching for such an algorithm is outside the scope of this work.

One suggested method for reordering is block-restricted sifting [5]. This allows individual blocks to be reordered in parallel, and we have implemented this algorithm. Though block-restricted sifting fits very well in the requirements for a parallel algorithm, the number of block is normally low (1-5) and the bulk of the work is often placed in one or two blocks.

4.7 Summary

When reordering we add a modified reference count to the nodes. This is done by dividing the BDD into blocks of variables, where reference count are kept within each block and nodes referenced from outside the block are just marked as such. No information flows between blocks as long as only swaps within the block are done.

The way indexes of nodes change require either the use of forward nodes, or adding the index to the nodes. After trying both we have chosen to add the index, because of poor memory use in the forwarding schemes.

We have discussed different representations, but have not tried anything other than the triple representation (index,page,number). The triple representation require the index to be added to the next pointer, which degrades the performance of the unique table

After reordering is done, the BDD is cleaned up to reclaim the memory that has been freed by the reduction in nodes. We clean up as much as possible, to make further manipulation of the BDD easier.

We have not made it possible to swap variables from different blocks. Swaps between block can be accomplished by stopping the reordering, cleaning up, redefining the blocks and then restarting the reordering.

A good algorithm that exploits the possibility of doing several swaps in parallel is needed before the reordering in BDDNOW can become efficient. We have implemented the block restricted sifting algorithm, but the amount of obvious parallelism in that is too small to make it an efficient parallel algorithm.

5 Experimental Results

Experimental results were obtained from running the package on a cluster of 16 PCs with 266MHz Pentium II processors, running FreeBSD, connected by a high speed network consisting of 32-bit LANai 4.1 boards and two M2M-dual 8 port SAN switches. Each pc has 128MB of RAM. We use a standard PVM implementation running on top of TCP/IP.

Due to some operating system constraints we were only able to get 64MB of memory on each machine, except on one machine, so all runs with 2 or more machines only have 64MB per machine available. Even without the OS memory limit, there was only about 100MB of physical memory available for user processes.

The testbed is close to what we can expect possible users to have available. 16 machines is slightly higher the the normal number of workstations available in a cluster, but we only used 8 of them in the test. The network is somewhat faster than most typical commodity networks, and it would have been better to run the test on a more common network type like an ATM net, but when using PVM on top of TCP/IP there may not be much difference in effective transfer rates.

5.1 Combinational Circuits

To give the customary benchmark example, we ran a simple combinational circuit comparison program on some ISCAS'85 circuits (comparing redundant and non-redundant versions, when available) and compared the execution times with our parallel BDD package to those with the CUDD package, which is a highly-efficient sequential BDD package.

Times are in seconds, number of recursive calls in millions. CUDD does not count the number of recursive calls, and even if it did, the number would be different as BDDNOW tests base case and does cache lookup before recursing. If we wanted to compare the number of calls in the two packages, we should have CUDD count the number of times the recursive function makes it past the cache lookup step.

Circuit	c1355		c1908		c880		c3540	
Nodes	199859		233471		1370138		3448095	
P	time	calls	time	calls	time	calls	time	calls
seq	1.44		2.36		11.33		52.67	
1	5.31	0.71	4.44	0.48	16.5	1.92	91.1	11.99
2	5.75	0.67	5.37	0.44	16.2	1.87	74.6	10.89
4	4.87	0.44	6.62	0.46	15.7	1.61	68.9	9.85
6	4.79	0.40	7.41	0.45	15.3	1.60	65.3	9.15
8	4.82	0.37	7.84	0.45	17.8	1.61	65.1	8.78

The runtime results are obtained as an average of 3 runs, the number of calls were just calculated for one of them (the program always calculate them). There was some variation in both times and number of calls, but no significant variation.

These results are not good, as we failed to achieve any speedup over the sequential code. However building the BDD for combinational circuits performs many operations on small BDDs. Upon closer examination, we found that although the processors do about the same amount of work, they spend considerable time idling, which suggests that superscalar operation may help keep all the processors busy a greater part of the time, and thus reduce execution time. Another property to be observed from the results is that although the parallel package performs poorly on small circuits, it gets better as the BDDs involved increase in size.

We did not find an ISCAS'85 circuit that fit in 100MB but not in 64MB, so we cannot give a fair comparison (i.e., against non-thrashing sequential code) requiring the parallel package to combine memory from several machines. Obviously, the parallel package vastly outperforms the sequential code once the sequential code runs out of physical memory.

5.2 Big BDDs

Recall that our primary design goal was handling large BDDs. To get something scalable, we build BDDs for the n functions $g_0, \ldots, g_{n-1}$, where

$$g_i = \bigwedge_{j=0}^{n-1} (a_i \vee b_{i+j \bmod n})$$

This gives an exponential size BDD, and we can set n as we need to get a appropriate size BDD. For $n = 20$, the BDD has 5.25M nodes and the CUDD package uses 93MB of memory. Times are in seconds.

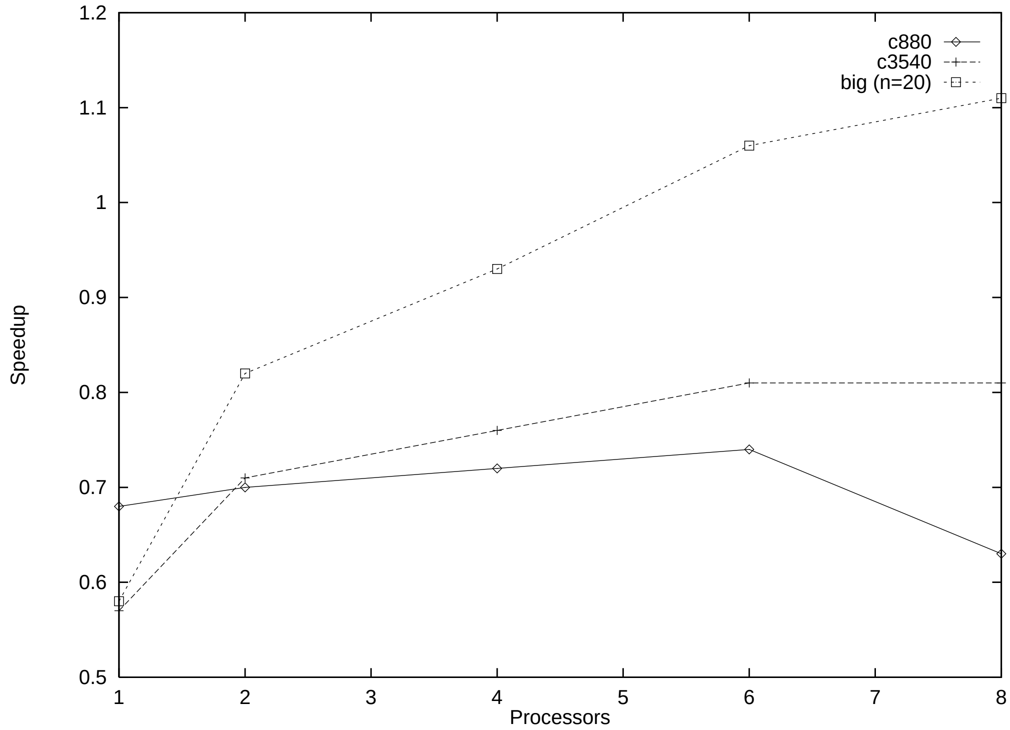


Figure 14: Absolute speedups for the two large circuits, and for the big BDD.

P	time	calls
seq.	69.5	
1	119.7	1.76
2	84.62	1.55
4	74.2	1.39
6	65.0	1.35
8	62.8	1.35

We believe this is the first reported speed-up of a distributed-memory parallel BDD package over efficient sequential code running in main memory (not thrashing).

We also ran with $n = 21$. The BDD had 10.5M nodes, and thus requires around 200MB. Due to the granularity of the memory balancing, we were not able to fit this BDD on 4 machines, but we did build it using 6 machines, taking 172 seconds and doing 34 million recursive calls. The sequential package obviously cannot build this BDD on a single machine.

5.3 Cache Effects

Some of the speed-up achieved is due to the fact that each processor allocates as much cache as one processor in the sequential version, so the parallel version has more total cache available.

All the tests were run with a computed cache size of 100000. This is the default limit in CUDD. The CUDD documentation suggests that we start with a smaller cache, and the package then increases the size, if hit-rates suggest that this would be beneficial. We have chosen to start at the maximum size for two reasons. We have used the same measure in the parallel package to determine when the cache should be resized, and this may not be optimal, especially with the overloading of the cache functionality. The other reason is that if for some reason the cache is much too small, the entries may in general not live long enough to be used, causing the program to decide against making the cache bigger. We actually experienced this, though it was a result of reducing the size after reorder, thinking it would be resized correctly.

We discovered another effect related to the cache. To get an idea of how the speedup is attributed to the extra cache we made some runs with twice the cache space available. One interesting result came from this. For one example, when we doubled the cache and ran it on one processor, we generated more recursive calls than with two processors with the old cache size. Normally we would expect the one processor version to have less calls, because it has the same available cache as the sum of the two, and it is less restricted in the use of the cache space. We think this is because the two processor run has a fairly bad load distribution, with the top processor doing only about a quarter of the work. This in turn gives more cache per call in the higher part of the computation DAG, and in this part, cache hits are more valuable. In general it seems that not much attention has been paid to how to structure the cache, and what eviction policies to use. Researching cache effects would be interesting for parallel as well as sequential BDD manipulation.

5.4 Optimizing Code

It is clear that the parallel version running on one processor is quite a bit slower than the sequential code. This slowdown may be overhead from the necessary changes needed to parallelize it, but it may also be from inefficiencies in the code. We know that some of the operations done could be implemented more efficiently , so working a bit on optimizing the code should increase the speed of the parallel execution. How much the speed would increase depends on the extend of the communication overhead.

5.5 Reordering

To see how effective the parallel reordering is, we ran some timings on it. The tests were not run on the very final version of the package, but nothing that affects the reordering part has changed since they were run.

The times are the times it take to build the BDD, and then run the reordering algorithm on it. The block restricted sifting (BRS) algorithm is in general faster than normal sifting. We wanted to see how much slower the parallel code is on one machine, and since CUDD does not support BRS (yet), we forced all variables into one block, giving the same effect as running sifting. We only ran BRS with as many processors as there were blocks.

Here are the results. The times are in seconds, the sizes given are the final number of BDD nodes in thousands.

Circuit	c1355			c1908			c3540		
Nodes	196238			134938			3135272		
P	time	size	swaps	time	size	swaps	time	size	swaps
seq	52.0	129	1797	22.1	29	1127	547	135	2752
1 (sift)	124.8	131	2641	55.1	29	1677	871	135	3882
1 (BRS)	60.3	130	1361	27.8	56	763	1955	1412	2321
2	46.9	130	1305	22.9	56	699	1310	1259	2225
3				23.7	56	699			

We will not go into great detail with analysis of these results, but there are a few things we will comment on. The sequential code does less swaps in sifting. This is because the sequential code has some optimizations that we have not yet implemented. There is only one place where the parallel case is faster, and this it mainly because the BRS algorithm does less swaps. The reason why one and two processors get a different number of swaps, is that there is a limit of 20% on how much the BDD can grow during reordering. In the parallel case the 20% growth is based on the number of nodes stored on the processor. With fewer nodes than the entire BDD, the allowed absolute growth is less, so the variable cannot be sifted quite as far away from the starting position, resulting in fewer swaps. In some cases this could lead to the parallel version not finding a different order.

6 Future Work and Conclusion

The package described in this work is still mainly a proof-of-concept implementation, so much work remains to be done. Some obvious next steps include improving the robustness of the code, to ready it for distribution, and implementing more of the CUDD API.

Implementing true superscalar execution as described in section 3.10 would be a practical way to improve performance on certain applications.

This BDD package is an efficient and portable BDD package that demonstrates speed-up over optimized sequential code and effectively uses the memory of networked workstations. It will be an excellent platform for further research.

The test were run on commodity PCs, but with a high speed network. Running test on a more commodity type network, as well as multiprocessors with native PVM implementation is recommended to get a better idea how the package performs on other setups.

Dynamic variable reordering remains a challenging problem. Although we were able to parallelize block-restricted sifting, sometimes there are not enough blocks to gain significant speed-up. Also, block-restricted sifting does not always find a good variable order. More research is needed into variable reordering algorithms that are parallelizable.

The package is nearly as memory-efficient as CUDD [12], the optimized sequential package on which it is based. With reordering, some additional memory overhead is incurred, similar to the technique Ranjan *et al.* [9] used to add reordering to the breadth-first CAL package [11].

We discovered that there may be effects in the use of a computed cache that is not fully understood, and more research into this should be valuable for both parallel and sequential BDD packages.

Finally, the package is designed for portability: the API is a subset of the CUDD API to simplify porting applications between our package and the popular CUDD package, and the parallel code uses the widely supported and freely available PVM package [3] for machine independence.

The use of PVM allows the use of heterogeneous setup of machines, and it would be interesting to investigate how to make effective use of a heterogeneous collection of machines.

The scalability is limited by the fact that the work on a given level cannot be shared among several processors. An obvious way to extend the scalability is to examine the feasibility of parallelizing the individual levels.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[2] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical Report CMU-CS-92-160, Carnegie Mellon University, July 1992. A version of this report was published in *Computing Surveys*, Vol. 24, No. 3, September 1992, p. 293–318.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. See also `http://www.epm.ornl.gov/pvm/pvm_home.html`.

[4] A. J. Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*. PhD thesis, Stanford University, January 1996. Published as Stanford University Tech Report STAN-CS-TR-95-1561.

[5] C. Meinel and A. Slobodová. Speeding up variable reordering of OBDDs. In *International Conference on Computer Design*, pages 338–343. IEEE, October 1997.

[6] K. Milvang-Jensen and A. J. Hu. BDDNOW: A parallel BDD package. In *Formal Methods in Computer-Aided Design*, 1998. To Appear.

[7] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *International Conference on Computer-Aided Design*, pages 48–55. IEEE, 1993.

[8] Y. Parasuram, E. Stabler, and S.-K. Chin. Parallel implementation of BDD algorithms using a distributed shared memory. In *27th Hawaii International Conference on System Sciences*, pages 16–25. IEEE, 1994.

[9] R. K. Ranjan, W. Gosti, R. K. Brayton, and A. Sangiovanni-Vincentelli. Dynamic reordering in a breadth-first manipulation based BDD package: Challenges and solutions. In *International Conference on Computer Design*, pages 344–351. IEEE, October 1997.

[10] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 42–47. IEEE, 1993.

[11] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. High performance BDD package by exploiting memory hierarchy. In *33th Design Automation Conference*, pages 635–640. ACM/IEEE, 1996.

[12] F. Somenzi. CUDD: CU decision diagram package. Available from `ftp://vlsi.colorado.edu/pub/`.

[13] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *33th Design Automation Conference*, pages 641–644. ACM/IEEE, 1996.

A Workload Plots

Note that these results are not obtained with the final version of the package.

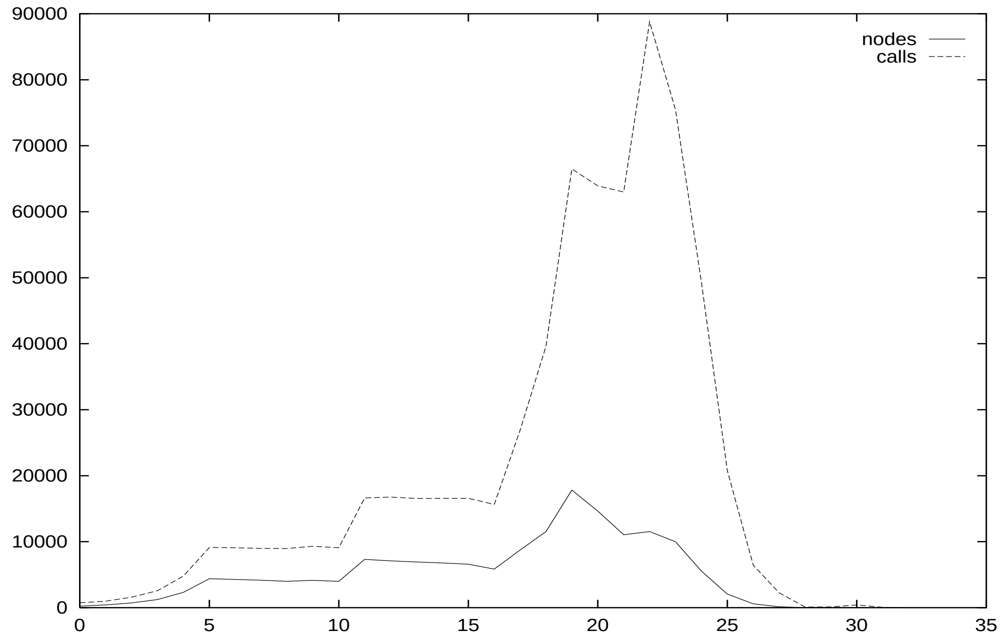


Figure 15: Workload and node distribution for c1908. The x-axis denotes the variable level.

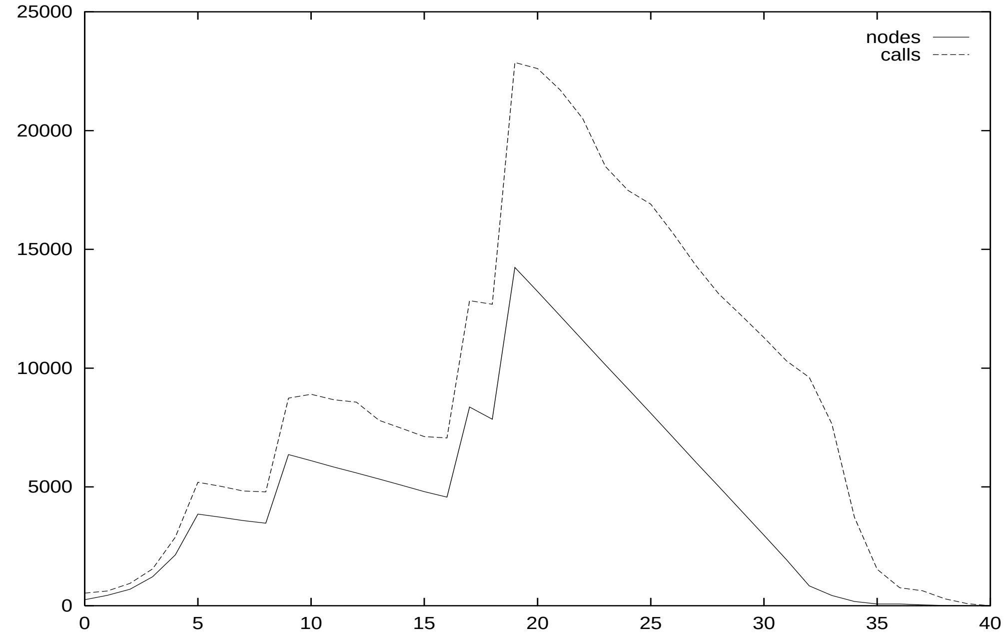


Figure 16: Workload and node distribution for c1355. The x-axis denotes the variable level.

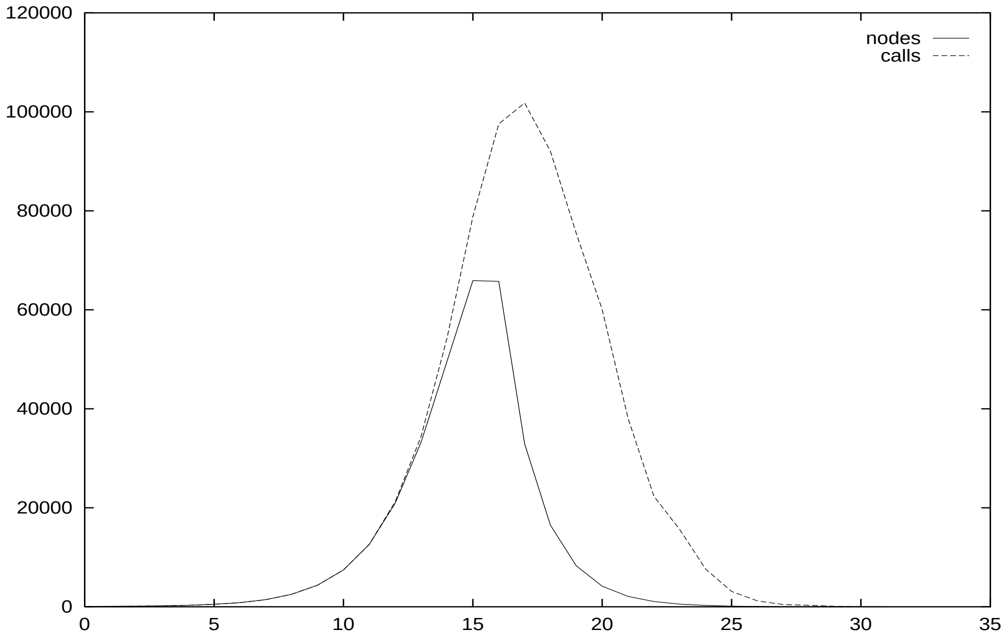


Figure 17: Workload and node distribution for the big BDD with n=16. The x-axis denotes the variable level.

B Tool Paper (to appear in FMCAD98)

BDDNOW: A Parallel BDD Package ⋆

Kim Milvang-Jensen[1] and Alan J. Hu[2]

[1] Dept. of Computer Science, Univ. of Copenhagen (DIKU)
lordtime@diku.dk, http://www.diku.dk/students/lordtime/
[2] Dept. of Computer Science, Univ. of British Columbia
ajh@cs.ubc.ca, http://www.cs.ubc.ca/spider/ajh/

Abstract. BDDs (binary decision diagrams) are ubiquitous in formal verification tools, and the time and memory used by the BDD package is frequently the constraint that prevents application of formal verification. Accordingly, several researchers have investigated using parallel processing for BDDs. In this paper, we present a parallel BDD package with several novel features. The parallelization scheme strives for minimal communication overhead, so we are able to demonstrate speed-up even running on networked commodity PC workstations. Average memory utilization per node is comparable to that of efficient sequential packages. In addition, the package supports dynamic variable reordering, and simultaneous computation of multiple BDD operations. Finally, the package is designed for portability – providing a subset of the CUDD API for the application programmer, and running on the widely available PVM package.

1 Introduction

Binary decision diagrams (BDD) [1] have become the dominant data structure for representing propositional logic formulas in formal verification and CAD. In practice, the memory required by large BDDs is typically the limiting factor for formal verification tools. In some cases, especially with the advent of dynamic variable reordering [5], run time is also a limiting factor. Accordingly, considerable research has been invested in more efficient BDD implementations.

An obvious line of research is to use parallel processing to accelerate BDD operations, but BDDs are not easily parallelizable. Nevertheless, several groups have produced parallel BDD packages for various different architectures. For space reasons, we discuss here only the work of Stornetta and Brewer [8], which is the prior work most similar to ours in that we both target distributed memory machines. We refer the interested reader to their paper for references to other work.

Networked workstations are currently the most widely available parallel computing resource, so we designed our package for this platform. Especially important for BDDs is the use of the pooled RAM of the networked workstations to work on BDDs too large to fit into the memory of any one workstation. Stornetta and Brewer also mention networks of workstations as a promising platform for their package. However, their package is more suitable for a parallel distributed memory supercomputer, as their scheme of

⋆ This work was performed while the first author was visiting the University of British Columbia. The second author was partially supported by an NSERC research grant.

randomly distributing nodes to processors (which gives excellent load balancing) results in very high communication overhead. Indeed, their reported results using a 32-node Meiko CS-2 showed speed-up over the sequential code only when the sequential code had exhausted its physical memory and was swapping to disk. In contrast, our approach trades off inferior load-balancing for lower communication. We are able to show speed-up over optimized sequential code with as few as four nodes, running on networked PCs. Both their package and ours can use the combined memory of networked workstations to avoid thrashing.

Our package has several other features. The package supports limited dynamic variable reordering — adjacent variables that are assigned to the same processor can be swapped reasonably efficiently, so we have implemented a parallel version of block-restricted sifting [3]. When not using reordering, the package is nearly as memory-efficient as CUDD [7], the optimized sequential package on which it is based. With reordering, some additional memory overhead is incurred, similar to the technique Ranjan *et al.* [4] used to add reordering to the breadth-first CAL package [6]. Another feature is the ability to perform several different BDD operations in parallel. Finally, the package is designed for portability: the API is a subset of the CUDD API, to simplify porting applications between our package and the popular CUDD package, and the parallel code uses the widely supported and freely available PVM package [2] for machine independence.

2 Design Principles

Our parallel BDD package was designed according to three main criteria: (1) Optimize for large BDDs, not small ones, (2) target networks of workstations as the platform, and (3) make the package portable.

The package runs on multiple machines using PVM, but the user program is just a sequential program, using the package exactly as it would use any other BDD package. The user can simply recompile the program with our package instead of the CUDD package, provided that only the supported subset of operations is used. For our experimental results, we used the same test program compiled with either our package or CUDD.

2.1 Workload Distribution

Manipulating BDDs typically consists of applying the following steps recursively:

1. Check for base cases.
2. Do a result-cache lookup to see if this operation has been done already.
3. Recurse on the left and right children.
4. Otherwise, find/create the node *ite(x,left_result,right_result)*.

This computation can be represented as a DAG, where the nodes represent operations and the edges represent recursive calls and cache hits. The base cases are the sinks. See Figure 1. This DAG can be traversed in any order, which allows doing many operations in parallel. The hard part lies in the large shared data structures (e.g., result cache and unique table). Since we want good memory utillization, data replication must be minimized.

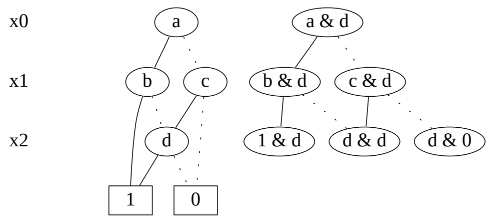


Fig. 1. A BDD and the resulting computation DAG for an operation on nodes a and d.

Because we are targeting the package at networks of workstations, we expect communication to be expensive, so we want to exploit any locality of reference we can find. As explained in [6], the part of the unique table that will be accessed by an operation can be determined by the topmost variable of the BDDs involved. The cache can also be divided into parts storing only results for where the topmost variable falls in a certain range. Therefore, each machine stores the nodes for some variable indices, and each machine does all operations where the topmost variable is one of the indices it has responsibility for. The advantage of this scheme is that all memory reference become local, and only the work moves between processors. To further limit communication, we have chosen to have each processor manage a contiguous range of indices. We experimented briefly with assigning indices in other ways, but did not achieve good results.

When a processor finds out that a recursive call has to be done on another machine, it sends the job, and goes on doing other work. This means that unlike in a depth-first traversal, the program needs to keep track of what work is in progress, and what state it is in. In other words, the computation DAG must be represented somehow. Fortunately, only the parts of the DAG where computation has not yet completed need to be represented, and it is possible to limit the size of that part by having a processor stop and wait for the jobs it has sent to other machines to complete, before going on to more work.

The main problem with this approach is the lack of efficient ways to balance the work. The only flexibility is the choice of what processors have responsibility for what indices. The package does load balancing whenever the size of the BDD doubles (starting at 100000 nodes). The indices are redistributed so that each processor gets approximately the same number of nodes, keeping neighbouring indices together. This is suboptimal: we have seen that the most work is not done where the most nodes are, but rather at indices slightly later in the order. A better strategy should take into account both where the work is done, as well as the amount of memory available at each processor. Our current choice does, however, give good distribution of memory use, and for very large BDDs, memory distribution is more important than work distribution.

2.2 Memory Usage

The CUDD package has fairly low memory usage, and we have largely maintained this in our package. To be able to determine the topmost variable in a call, we add the index to all pointers, just as is done in breadth-first BDD operations. This increases the size of each node slightly. Offsetting this increase, we eliminate the reference count. Keeping track of reference counts in a distributed graph is costly, and garbage collection should be done by using mark and sweep instead.

Normally a BDD is just a pointer to a BDD node. However, in a parallel implementation, having a pointer to memory on a different machine is not good design and may cause problems running on heterogeneous machines. It also makes moving indices from one machine to another costly, as you cannot guarantee that a node will get the same address on the new machine as it had on the old. Alternatively, one could modify the virtual memory system to create one linear address space over all the machines, but this too would be difficult on heterogeneous setups. To avoid these problems, we introduce a different addressing scheme. Within each index we create an array of arrays of nodes. By numbering the nodes for each index in an array, we get machine-independent addressing. The double indirection structure gives finer granularity over memory allocation. Each node is uniqly identified by the triple (index,array number,number within array). There is some waste of space, because node space is allocated per level, and an extra memory reference is required for each node access, but this reference will be to the small array of array addresses. The nice features of this scheme are that it is clean, moving indices is easy, and nodes require only 8 shorts to represent.

While the package does use memory of all the processors well, the size of the BDDs that can be handled is still limited by the fact that the index with the most nodes has to fit on a single machine.

2.3 Dynamic Variable Reordering

Dynamic variable reordering [5] is becoming increasingly more important, and a good BDD package must support it. The basic operation in reordering is the swapping of two adjecent variables. Though doing each swap parallelized would be interesting, we have concentrated our efforts in doing the individual swaps locally, but making it possible to do several independent swaps at the same time.

Some of the problems with doing reordering in our case are the same as when doing breadth-first BDD manipulation, as they spring from having the index on the ingoing pointer, rather then in the node itself, and from wanting to keep nodes with the same index located on the same pages in memory. Just as in [4], after trying different schemes for using forwarding pointers, we concluded that the best way is to change representation during reordering, and after reordering move the nodes so they are located where you want them. Unfortunately, moving nodes results in a memory overhead.

There are also challenges in reordering unique to the distributed storing of the BDD. Exactly what they are and how we have overcome them is beyond the scope of this paper. We will just note that they do result in some memory overhead, but to a large degree, we can use the same space as is needed for moving nodes, so the total memory overhead is not that much bigger.

A more interesting problem is the need of an algorithm that takes advantage of the fact that several independant swaps can be done in parallel. For example, block-restricted sifting [3], which was originally suggested as a faster sequential reordering algorithm, allows individual blocks to be reordered in parallel, so we have implemented this algorithm. The current implementation can swap variables only within a single machine, so more complicated reordering algorithms will have to iterate: (1) define blocks, (2) swap within blocks, and (3) cleanup. Currently, cleanup requires synchronization of all the machines, and the cleanup operation is itself fairly expensive, so depending on the algorithm used it may be nessecary to include support for redefining blocks for a certain part of the BDD without involving the rest.

2.4 Multiple Independent BDD Operations

The way the user program sends tasks to the processes running the BDD package is essentially no different from the way the processes send tasks to each other. Since internally many jobs are done at the same time, it is possible for the user process to start new BDD operations before the old ones have completed. This is currently implemented.

Ideally, the user would just send a sequence of operations to the BDD package, and the package would then figure out which ones did not depend on the results of the others, and have several operations executing at the same time. User calls to the BDD package would be non-blocking — the application program would continue to execute, doing useful work, and perhaps generating additional calls to the BDD package. The user program would not need to know about the parallel execution. The only change to the user program would be that BDD comparison would be done through a call to the BDD package that guaranteed that both operands were done before the pointers were compared. Intuitively, the BDD package would behave like a superscalar processor, dynamically extracting parallelism from sequential code.[1] Doing such superscalar execution is beyond the scope of our project, but integrating it with the current package, which does support multiple simultaneous BDD operations, should be simple.

3 Experimental Results

Experimental results were obtained from running the package on a cluster of 16 266MHz Pentium II computers, each with 128MB of RAM running FreeBSD, connected by a high speed network consisting of 32-bit LANai 4.1 boards and two M2M-dual 8 port SAN switches. We use a standard PVM implementation running on top of TCP/IP.

Due to some operating system constraints we were only able to get 64MB of memory on each machine, exept on one machine, so all runs with 2 or more machines only have 64MB per machine available. Even without the OS memory limit, there was only about 100MB of physical memory available for user processes.

3.1 Combinational Circuits

To give the customary benchmark example, we ran a simple combinational circuit comparison program on some ISCAS'85 circuits (comparing redundant and non-redundant

[1] Sanghavi *et al.* [6] present a similar idea under the monikers "pipelining" and "superscalarity" in the context of their breadth-first sequential BDD package. By introducing blocking comparisons, we propose dynamically extracting parallelism from an unmodified application program, whereas their method requires specially written code to use. By analogy, their approach resembles a statically scheduled VLIW processor more than a superscalar processor. Furthermore, in our parallel BDD package, we could actually take advantage of the parallelism, whereas they were simply trying to reduce page faults.

versions, when available) and compared the execution times with our parallel BDD package to those with the CUDD package[2], which is a highly-efficient sequential BDD package. Times are in seconds.

Circuit	Nodes	Operations	Sequential Exec. Time	Parallel Exec. Time w/ n Procs. 1	2	4	6	8
c1355	199859	583970	2.5	4.7	6.3	8.5		13.0
c1908	233471	752402	3.3	6.1	8.3	10.9		16.0
c880	1370138	1798896	10.5	14.9	22.2	23.5	22.4	
c3540	3448095	14286323	57.9	101.8	105.2	90.5	88.9	94.5

These results are not good, but building the BDD for combinational circuits performs many operations on small BDDs. Upon closer examination, we found that although the processors do about the same amount of work, they spend considerable time idling, which suggests that superscalar operation may help. Also, although the parallel package performs poorly on small circuits, it gets better as the BDDs involved increase in size. We did not find an ISCAS'85 circuit that fit in 100MB but not in 64MB, so we cannot give a fair comparison (i.e., against non-thrashing sequential code) requiring the parallel package to combine memory from several machines. Obviously, the parallel package vastly outperforms the sequential code once the sequential code runs out of physical memory.

3.2 Big BDDs

Recall that our primary design goal was handling large BDDs. To get something scalable, we build BDDs for the n functions $g_0, \ldots, g_{n-1}$, where $g_i = \bigwedge_{j=0}^{n-1}(a_i \vee b_{i+j \bmod n})$.

This gives an exponential size BDD, and we can set n as we need to get a appropriate size BDD. For $n = 20$, the BDD has 5.25M nodes and the CUDD package uses 93MB of memory. Times are in seconds.

Sequential Exec. Time	Parallel Exec. Time w/ n Procs. 1	2	4	6	8
70.5	108.8	77.9	68.1	61.0	57.6

We believe this is the first reported speed-up of a distributed-memory parallel BDD package over efficient sequential code running in main memory (not thrashing).

We also ran with $n = 21$. The BDD had 10.5M nodes, and thus requires around 200MB. Due to the granularity of the memory balancing, we were not able to fit this BDD on 4 machines, but we did build it using 6 machines, taking 151 seconds. The sequential package obviously cannot build this BDD on a single machine, demonstrating how the parallel package can harness the combined memory of several workstations.

Some of the speed-up is achieved because each processor in the parallel version allocates as much cache as a single processor in the sequential version, so the parallel version has more total cache available. Also, the parallel version running on one processor

[2] Results for CUDD reported are for version 2.1.2, which was the most current available and the code on which our BDDNOW package is based. By publication time, CUDD 2.2.0 had become available. Our techniques should be equally applicable to parallelizing the new version.

is clearly much slower than the sequential code. Some of this slow-down may be from necessary parallelization overhead, and some may be from insufficient code tuning in our preliminary implementation.

4 Future Work

The package described in this paper is still a preliminary proof-of-concept implementation. Some obvious next steps include ruggedizing the code to ready it for distribution, and implementing more of the CUDD API. More interesting future research involves load balancing and communication overhead. As we mentioned earlier, the package sacrifices some load balancing in order to reduce communication overhead. Unfortunately, in extreme cases, the workload can become so unbalanced that no useful parallel work is performed. Moreover, substantial time is still spent on communication. Further research is definitely warranted.

Implementing true superscalar execution as described in Section 2.4 would be a practical way to improve performance on certain applications.

Dynamic variable reordering remains a challenging problem. Although we were able to parallelize block-resticted sifting, sometimes there are not enough blocks to gain significant speed-up. Also, block-restricted sifting does not always find a good variable order. More research is needed into variable reordering algorithms that are parallelizable.

In sum, this BDD package is an efficient and portable BDD package that demonstrates speed-up over optimized sequential code and effectively uses the memory of networked workstations. It will be an excellent platform for further research.

References

1. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
2. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. See also `http://www.epm.ornl.gov/pvm/pvm_home.html`.
3. C. Meinel and A. Slobodová. Speeding up variable reordering of OBDDs. In *International Conference on Computer Design*, pages 338–343. IEEE, October 1997.
4. R. K. Ranjan, W. Gosti, R. K. Brayton, and A. Sangiovanni-Vincentelli. Dynamic reordering in a breadth-first manipulation based BDD package: Challenges and solutions. In *International Conference on Computer Design*, pages 344–351. IEEE, October 1997.
5. R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 42–47. IEEE, 1993.
6. J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. High performance BDD package by exploiting memory hierarchy. In *33th Design Automation Conference*, pages 635–640. ACM/IEEE, 1996.
7. F. Somenzi. CUDD: CU decision diagram package. Available from `ftp://vlsi.colorado.edu/pub/`.
8. T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *33th Design Automation Conference*, pages 641–644. ACM/IEEE, 1996.