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.

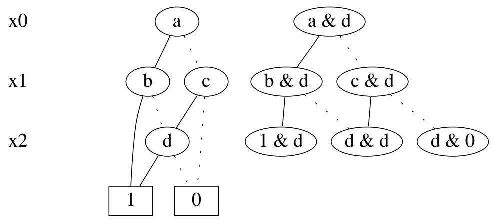

Fig. 1. A BDD and the resulting computation DAG for an operation on nodes a and d.

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 uniquly 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.