Parallel Construction Algorithms for BDDs

Jer-Sheng Chen and Prithviraj Banerjee

Technical Report No. CPDC-TR-9801-006

Center for Parallel and Distributed Computing

Department of Electrical & Computer Engineering

Northwestern University

Technological Institute

2145 Sheridan Road, Evanston IL-60208

Parallel Construction Algorithms for BDDs *

Jer-Sheng Chen and Prithviraj Banerjee

Department of Electrical and Computer Engineering

Northwestern University

Evanston, IL 60208

{chenjs,banerjee}@ece.nwu.edu

Abstract

Binary Decision Diagrams (BDDs) provide very efficient representations of Boolean functions and have been widely used in various computer-aided design of VLSI systems. As the construction time of BDDs varies with applications and is often large in some complex circuits, it would be useful to design parallel algorithms to construct BDDs. In this paper, we propose two parallel algorithms for constructing the BDDs of logic circuits. We first present a shared memory parallel algorithm using locks. To achieve more parallelism, we present another algorithm using replication of fanin cones that is suitable for execution on distributed memory multicomputers. We have implemented our parallel algorithms on two shared memory multiprocessors and one distributed memory multicomputer. Experimental results showing speedups for some ISCAS benchmark circuits are reported on an 8 processor IBM J-40 shared memory multiprocessor, an 8 processor SGI Origin shared memory multiprocessor, and a 16 processor IBM SP-2 distributed memory multicomputer.

Key words: Binary Decision Diagrams (BDDs), parallel algorithms, fanin cones, shared memory multiprocessors, distributed memory multicomputers.

*This research has been supported by DARPA under contract DAAH-94-G-0273.

1 Introduction

The manipulation of Boolean functions is one of the most essential operations in various applications of Computer-Aided Design (CAD) of VLSI systems, such as verification, synthesis, logic minimization, simulation and testing. To manipulate Boolean functions efficiently, various representations of Boolean functions have been proposed in the past years. One of the most state-of-the-art representations is the Binary Decision Diagram (BDD) [1]. BDDs are canonical representations and provide symbolic manipulations of Boolean functions. However, for some large-scale circuits, the constructions of their BDDs can be very time-consuming and the memory needed can be prohibitively large.

To solve large-scale problems, parallel processing has been used effectively either by distributing computational loads among multiple processors or by utilizing the large memory in parallel machines. For circuits with large BDDs, parallel processing can be used both to reduce the construction time and to extend the memory limitation in uniprocessors.

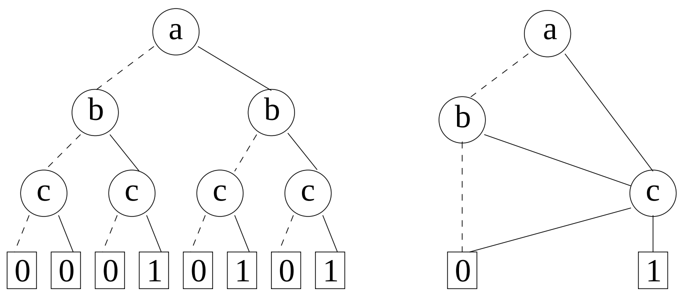
In this paper, we propose two parallel algorithms for constructing BDDs of logic circuits. One is used in shared memory multiprocessor systems and the other in distributed memory multicomputer systems. Our approach is based on the depth-first BDD algorithm and the fanin-cone partition. For each output in a circuit, we form its fanin cone and construct its corresponding BDD. The circuit is then partitioned according to the fanin cones. The resulting fanin cones can be assigned to different processors where the BDDs are actually constructed.

We have implemented our parallel algorithms on both shared memory and distributed memory systems. The target machines we used are an 8 processor IBM J40 shared memory multiprocessor, an 8 processor SGI Origin 2000 distributed shared memory multiprocessor, and a 16 processor IBM SP-2 distributed memory multicomputer. For the experiments, we constructed the BDDs for the ISCAS benchmark circuits [2][3].

The rest of this paper is organized as follows. Section 2 gives an overview of BDDs and describes the serial BDD construction algorithm. Related work for the parallel BDD construction is summarized in Section 3. Section 4 describes the shared memory parallel algorithm, and Section 5 describes the distributed memory algorithm. Finally, Section 6 points out some limitations to our algorithms and addresses some improvements

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a)



(b) (c)

Figure 1: An Boolean function represented with (a) a Truth Table, (b) a Binary Decision Tree, and (c) a Binary Decision Diagram.

to them.

2 BDD Overview

A Boolean function can be represented by a complete binary tree called a *binary decision tree*, which is based on the truth table of the function. As an example, Figure 1(a) shows the truth table of the Boolean function, $f = \overline{a} \cdot c + b \cdot c$, and Figure 1(b) shows the corresponding binary decision tree. Each *nonterminal* node is labeled by a variable and has two edges directed toward two children nodes. The 0-branch (shown as a dashed line) represents the case when the variable is assigned 0 and the 1-branch (shown as a solid line) represents the case when the variable is assigned 1. Each *terminal* node is labeled 0 or 1. Each path from the root to a terminal node corresponds to a truth assignment of a Boolean function, and the value of the function for this assignment is determined by the terminal node label.

As can be seen in Figure 1(b), there are some redundant nodes and duplicate subgraphs in the binary decision tree. If we eliminate all the redundant nodes and duplicate subgraphs, we end up with what is known as a *binary decision diagram* (BDD). A BDD is no longer a binary tree but a *directed acyclic graph* (DAG). Figure 1(c) shows the BDD representation of the binary decision tree in Figure 1(b). In most cases, a BDD contains less nodes than its corresponding binary decision tree representation.

For a BDD representing a Boolean function, if we put a restriction that variables occur in the same order on all paths from the root to the terminal nodes and each variable occurs at most once on each path, then the BDD for the function is called the *ordered binary decision diagram* (OBDD). Figure 1(c) shows an OBDD with variable ordering $a < b < c$. [1] has shown that the OBDD is a canonical form for a Boolean function; that is, for any function, there is *exactly one* OBDD representing it. Note that the ordering of variables chosen for a BDD can have a great impact on the size of the BDD. The BDD size of one variable ordering could be exponentially more compact than the size of another ordering. For the rest of the paper, only OBDDs are considered and for briefness we will refer to these graphs as BDDs.

2.1 Basic BDD Construction

Consider a Boolean function $f(x_1, x_2, \ldots, x_n)$ and assume the input variables are ordered according to the index $(x_1 < x_2 < \ldots < x_n)$. The BDD representing f can be constructed by applying the Shannon expansion rule as follows:

$$f(x_1, x_2, \ldots, x_n) = x_1 \cdot f_{x_1=1} + \overline{x_1} \cdot f_{x_1=0} \tag{1}$$

where $f_{x_1=1}$ and $f_{x_1=0}$ are the *positive* and *negative cofactors* of f with respect to x_1, and $\cdot$ and $+$ are logical AND and OR. If F is the BDD of f, x_1 will be the variable in F's root node and the cofactor $f_{x_1=1}$ and $f_{x_1=0}$ are the subgraphs of F's 1-branch and 0-branch, respectively. The expansion can be recursively applied to the two cofactors using x_2 as the splitting variable, and repeated for all variables. If the constructed BDD contains identical subgraphs or redundant nodes at the end of the expansion, a reduction step [1] to eliminate these subgraphs and nodes will be performed in the reverse order starting from the terminal nodes.

2.2 Serial Algorithm for BDD Construction

In this section, we describe a depth-first based serial BDD construction algorithm proposed in [1][4][5]. The depth-first BDD algorithm described here forms the basis of our parallel algorithms.

Given two BDDs F and G representing the Boolean functions f and g that have the same variable ordering (e.g., $x_1 < x_2 < \ldots < x_n$) and a Boolean operation $< op >$ (e.g., AND or OR), the BDD of the operation $f < op > g$ can be constructed by applying the $Apply(op, F, G)$ operation proposed in [1]. The operation *Apply* can be recursively called based on the Shannon expansion rule as follows:

$$Apply(op, F, G) = x_i \cdot Apply(op, F_{x_i=1}, G_{x_i=1}) + \overline{x_i} \cdot Apply(op, F_{x_i=0}, G_{x_i=0}) \tag{2}$$

If we start from the roots of the two BDDs, we can then construct the result BDD by recursively constructing the 0- and the 1-branches and then generate the new root from these. Based on the recursive expansion, a depth-first BDD construction algorithm is shown in Figure 2.

The algorithm takes an operator ($< op >$) and its two BDD operands (F and G) as inputs and returns a BDD for the result of the operation $f < op > g$. Instead of building the BDD and then reducing it to a canonical form, this algorithm guarantees to maintain *canonicity* while constructing the BDD by using a hash table. The hash table, called the *unique table*, records all existing BDD nodes and a collision chain is used to resolve collisions. To avoid exponential blow-up recursive calls and reduce memory usage, a hash-based software cache, called *computed table*, is used to store previously computed results. The use of the computed table can greatly eliminate unnecessary recursive calls and thus reduce the construction time. The recursive steps in the algorithm end when a terminal case is encountered (line 1-2) or when the operation is found in the computed table (line 4). The terminal case is when the result of the operation can be trivially evaluated. For each non-redundant result node, it is first checked against the unique table (line 12). If the node is not in the unique table, it is inserted into the unique table and the computed table as well (line 13-15). If a node is redundant, it is eliminated (line 9) and its child node is inserted into the computed table as the computed result.

```
Apply(op, F, G) // assume that F and G have the same variable ordering
    if terminal case
        evaluate (op, F, G) and return the result
    // Check up the cache
    if the operation (op, F, G) is in the computed table, return the result found
    // Depth-first traversal
    x ← the variable with the smallest index in F and G
    T ← Apply(op, F_{x=1}, G_{x=1})
    E ← Apply(op, F_{x=0}, G_{x=0})
    if (T == E), result ← T // redundant node
    else
        // Check up the hash table before creating a new node
        result ← look up unique table with key (x, T, E)
        if (result == NULL) // a new node
            result ← create a BDD node with key (x, T, E)
            insert result into the unique table
    // Insert the computed result into the cache
    insert the operation (op, F, G) and result into the computed table
    return result
```

Figure 2: Depth-first based BDD algorithm.

3 Related Work

Since the use of BDDs was proposed to manipulate Boolean functions, many parallel algorithms have been developed to speed up the construction of BDDs. In this section, we briefly describe some of them.

Kimura and Clarke [6] first developed a parallel algorithm based on the level of logic operations. Their algorithm treated BDDs as minimal finite automata and constructed the BDDs through building and minimizing the product automaton. A speedup of 10 was reported on a 10-bit multiplier on a 16-processor shared memory machine. The work was extended in [7] by using the *Apply* algorithm. A dynamic expansion of logic operations was also proposed to gain more parallelism. A speedup of 14 was reported on a 10-bit multiplier on a 25-processor shared memory machine.

A vectorized parallel BDD algorithm was proposed by Ochi et al. in [8]. Instead of using depth-first algorithm, their algorithm was based on breadth-first manipulation of BDDs, which requires expansion and reduction phases. Speedups have been reported on several ISCAS-85 benchmark circuits.

Parasuram et al. [9] have proposed an algorithm on a massively parallel machine. Data such as nodes and hash table were arbitrary distributed and shared among processors using a *Distributed Shared Memory*

(DSM) abstraction. A distributed stack was used to achieve fine-grained parallelism. Speedups from 20-32 were reported on some ISCAS-85 circuits on a CM-5 machine.

A data parallel BDD algorithm on a SIMD machine was proposed by Gai et al. [10]. BDD nodes and hash table were distributed on a MasPar MP-2 with 16K PEs. CPU times for constructing BDDs for some ISCAS circuits were reported.

Sotrnetta and Brewer [11] proposed a parallel BDD package for a distributed environment such as a *network of workstations* (NOW) or a distributed memory multicomputer. Different forms of parallelism based on depth-first BDD algorithm were exploited. BDD nodes and hash table were distributed among processors using a two-level hash function. Simultaneous threads of computation were executed through the use of the work queue on each processor. CPU times on some ISCAS-85 circuits were reported on Meiko CS-2 with 64 processor nodes.

A breadth-first algorithm for manipulating BDDs has been reported by Ranjan et al. [12]. BDDs were partitioned on a network of workstations such that BDD nodes for a set of consecutive variables are assigned to the same workstation. After a workstation has finished processing its nodes, it then passes them to the workstation which has the next set of variables.

Arunachalam et al. [13] proposed a distributed BDD package for a cluster of workstations. Their goal was to verify large-scale circuits that can not fit into the memory in one single machine. BDD nodes were distributed among the workstations and a distributed pointer pointing to a node was used if the node is allocated on other machine. Dynamic memory load balancing was also used when the memory use of one machine exceeds a predefined threshold value.

In this paper, we propose a parallel BDD algorithm based on the depth-first BDD construction. For each output in a circuit, we first generate its fanin cone. These fanin cones can be distributed to processors on which their corresponding BDDs are constructed. A lock based shared memory parallel algorithm is described in Section 4. More parallelism can be achieved by replicating fanin cones among processors. We describe such an algorithm in detail in Section 5.

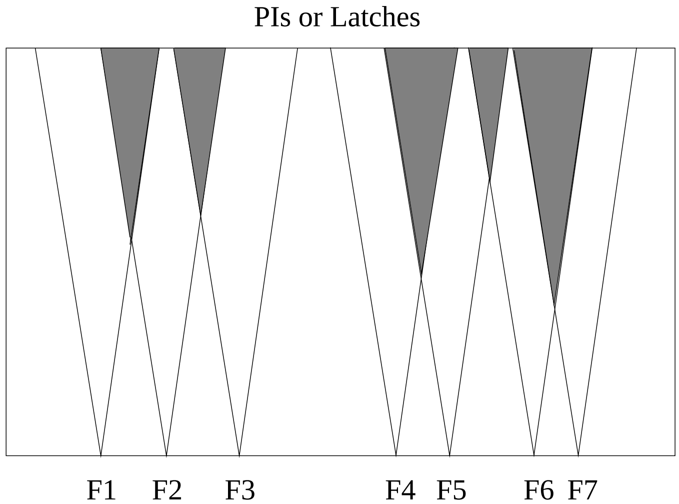


Figure 3: Fanin cones example. Overlap regions are shown in the shaded cones.

4 Shared Memory Parallel Algorithm

In this section, we describe a parallel BDD construction algorithm for shared memory multiprocessors. In a shared memory system, each processor can access the shared memory independently but needs to synchronize with each other when updating shared data.

The basis of our algorithm is the depth-first *Apply* algorithm described in Section 2.2 and the fanin cone of each output in a circuit. A fanin cone is generated for each primary output (PO) or latch input in a circuit. This can be done by traversing depth first from each PO or latch input back to primary inputs (PIs) or latch outputs. Each fanin cone represents a logic function in the circuit and can be assigned to a processor where its corresponding BDD is constructed. As an example, Figure 3 shows a possible fanin cone configuration in a typical circuit. In this figure, each PO corresponds to a fanin cone. The overlap regions (shown in shaded cones) are shared among fanin cones. Processors need to synchronize when accessing these shared regions. A work queue is maintained to keep track of all the fanin cones that are not processed. At the beginning, all the fanin cones are added into the work queue. During the execution, each processor removes one cone from the work queue and constructs the BDD. Whenever a processor completes constructing the BDD, it takes the next available cone from the work queue and continues its work.

Because the unique table and the computed table are shared by all processors, any part of codes that

```
Read in a circuit
Generate a fanin cone for each primary output or latch input
Build up the BDDs for the primary inputs or latch outputs
Create a work queue, wq, and add each fanin cone to it
queueSize ← the number of fanin cones in the circuit

for all processors in PARALLEL do
while TRUE do
    lock(wq.lock)
    // No more work left in the queue
    if (queueSize == 0)
        unlock(wq.lock)
        exit
    queueSize ← queueSize − 1
    // Take the next available work
    myWork ← the first work in wq
    unlock(wq.lock)
    // Perform BDD operations on the fanin cone
    perform myWork
end while
End
```

Figure 4: Shared memory parallel algorithm.

updates these two tables is the *critical region*. To avoid race condition, *locks* and *unlocks* are used when multiple processors want to enter the critical region at the same time. If one processor is already in the critical region, no other processor can enter the critical region before the lock to the region is released. However, it would be very inefficient if too many processors just wait for the lock before they can enter the same critical region. The goal of our algorithm is to minimize the idle time imposed on processors due to waiting for the locks. To achieve the goal, each entry in the unique table and the computed table is associated with a unique lock such that processors working on different nodes at the same time can still access these tables without being locked out. A processor will be blocked only when the processor wants to access the shaded region which is being locked by other processor. The work queue is implemented as a centralized linked list and is associated with a global lock. Figure 4 shows the outline of the parallel algorithm. Each $myWork$ function in the algorithm is responsible for building up the BDD of a fanin cone.

Table 1: Elapsed time (seconds) for constructing BDDs on the IBM J-40.

Circuits	Outputs	1 Processor	2 Processors	4 Processors	8 Processors
C1355	32	8.84	6.73	5.63	8.78
C1908	25	1.90	1.67	1.55	4.64
C7552	108	1.37	1.06	0.97	2.52
C499	32	3.67	2.99	2.45	8.02
C880	26	1.15	0.86	0.80	1.75
C3540	22	20.18	13.72	10.92	16.81
s5378	213	0.50	0.36	0.29	0.87
s9234	250	0.90	0.61	0.45	1.22

Table 2: Elapsed time (seconds) for constructing BDDs on the SGI Origin 2000.

Circuits	Outputs	1 Processor	2 Processors	4 Processors	8 Processors
C1355	32	7.36	5.53	4.60	7.12
C1908	25	1.58	1.18	0.98	1.50
C7552	108	1.14	0.85	0.71	1.12
C499	32	3.06	2.30	1.92	6.88
C880	26	0.88	0.66	0.58	0.87
C3540	22	15.70	11.73	9.80	14.47
s5378	213	0.41	0.29	0.22	0.56
s9234	250	0.70	0.51	0.38	0.98

4.1 Experimental Results

We have implemented the above algorithm on the IBM J-40 and SGI Origin 2000 shared memory multiprocessors. Currently each of these two machines is configured as 8 processors with 1GB shared memory. The test circuits used are from the ISCAS-85 combinational and ISCAS-89 sequential benchmarks [2][3]. For sequential circuits, combinational circuits are extracted by replacing latches with PIs and POs. The BDDs representing each PO for the test circuits are constructed and the elapsed times are recorded. The algorithm in [17] is used to determine the variable ordering for all test circuits. In our experiment, all output functions of each circuit use the same variable ordering.

Table 1 shows the elapsed time for building BDDs with different number of processors on the IBM J40. Table 2 shows the results on the SGI Origin 2000, using the same test circuits and variable orderings. As can be seen from these tables, there are some speedup gains when the processor number is increased from 1 to 4. However, the performance degrades as the processor number is increased to 8.

Table 3: The percentage of logic overlap among fanin cones in ISCAS circuits.

Circuits	Inputs	Outputs	BDD nodes	Overlap Percentage
C1355	41	32	27,867	78%
C1908	33	25	7,571	60%
C7552	207	108	2,839	27%
C499	41	32	27,867	78%
C880	60	26	4,071	30%
C3540	50	22	34,735	32%
s5378	199	213	2,116	43%
s9234	247	250	3,307	51%

4.2 Summary

The results shown in Table 1 and Table 2 are not very scalable in terms of the number of processors. We have observed that the overlap regions among fanin cones in some of the benchmark circuits are very large. Table 3 shows the overlap percentage in these benchmark circuits. Locks and unlocks need to be used when processors enter these shared regions. We have noticed that the larger the overlap regions are, the more the contentions to the shared data among processors will occur. For a system with small number of processors, the cost in the contentions can still be compensated by running multiple processes in parallel. However, the cost due to the contentions become a dominant factor after the number of processors exceeds a certain point. Table 1 and Table 2 show that the performance become degraded as the number of processors increases from 4 to 8.

Our algorithm using locks has a bottleneck due to the contention among processors. One way around the problem of the contention is to replicate the fanin cones to each processor and have all the processors perform BDD construction in parallel. In the next section, we will describe such a parallel algorithm.

5 Parallel Algorithm Using Replication of Fanin Cones

The algorithm first broadcasts the circuit copy to all processors. By replicating the whole circuit, each processor can construct its own BDDs in parallel without any contention. Before any BDD can be constructed, each processor reads in the circuit and determines the number of PIs and the level for each fanin cone.

```
Read in circuit
numProcs ← the number of processors
myId ← processor id
Broadcast circuit copy to all processors
For all processors in PARALLEL do
    Generate a fanin cone, fin, for each primary output or latch input
    Determine the level of each fanin cone
    numFanin ← the number of fanin cones
    Sort the fanin cones according to the level as fin[0], fin[1], ..., fin[numFanin − 1]
    Build up the BDDs for each primary input or latch output
    for i = myId to numFanin
        Build up the BDDs for each fin[i]
        i ← i + numProcs
End
```

Figure 5: Parallel BDD algorithm using replication of fanin cones.

The level is the recursive depth traversing from a PO back to its PIs. After the fanin cones are generated, the fanin cones are sorted by their levels decreasingly. By sorting the fanin cones, we can use interleaved scheduling to distribute the workload among processors. The idea of sorting the fanin cones is to get a better load balancing. An outline of the parallel algorithm is shown in Figure 5.

5.1 Experimental Results

We have implemented the algorithm on the two machines (J-40 and Origin 200) and a distributed memory machine, IBM SP-2, using the *Message Passing Interface* (MPI) library. The IBM SP-2 is currently configured with 16 processors and each processor has a 128MB local memory. For this experiment, we use the same benchmark circuits and the same variable ordering described in Section 4. Table 4–Table 6 show the elapsed time for building BDDs with different number of processors, and Table 7–Table 9 show the speedups of the parallel implementation over a single processor on these three systems, respectively.

5.2 Summary

The algorithm can be used in a distributed memory system. Note that the communication overhead among processors is very small. The only disadvantage of this algorithm is that the processors perform some redundant work. The redundant work comes from the logic overlaps mentioned above. As an example shown

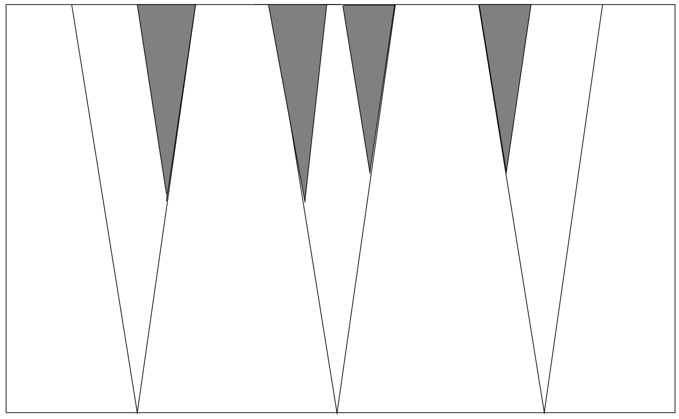


Figure 6: Replication of fanin cones. The redundant work performed by each processor is shown in the shaded regions.

in Figure 6, the shaded regions are the redundant work if the fanin cones are replicated to 3 processors. In this figure, processor 1 (P1) and processor 3 (P3) perform the redundant work (the shaded regions) that is shared with processor 2 (P2), and P2 performs the redundant work that is shared with P1 and P3. As a result, the speedup numbers shown in Table 7–Table 9 are limited.

One possible solution to reduce the redundant work is to partition the circuit into a set of logic partitions by the fanin cones [15]. The idea is that the partitions are assigned to processors such that the redundant work performed by each processor is as small as possible. Each processor only has to process its own partitions and can work independently.

Table 4: Elapsed time (seconds) for constructing BDDs on the IBM J-40 using fanin-cone replication.

Circuits	Outputs	1 Processor	2 Processors	4 Processors	8 Processors
C1355	32	4.31	2.55	1.94	1.80
C1908	25	0.85	0.64	0.53	0.51
C7552	108	0.65	0.49	0.33	0.30
C499	32	1.55	0.85	0.84	0.46
C880	26	0.52	0.39	0.31	0.27
C3540	22	10.21	9.02	8.34	8.18
s5378	213	0.27	0.21	0.18	0.15
s9234	250	0.48	0.32	0.27	0.23

Table 5: Elapsed time (seconds) for constructing BDDs on the SGI Origin 2000 using fanin-cone replication.

Circuits	Outputs	1 Processor	2 Processors	4 Processors	8 Processors
C1355	32	3.52	2.12	1.48	1.37
C1908	25	0.73	0.52	0.41	0.39
C7552	108	0.56	0.39	0.27	0.23
C499	32	1.16	0.72	0.71	0.39
C880	26	0.45	0.30	0.24	0.22
C3540	22	8.54	6.92	6.63	6.10
s5378	213	0.23	0.17	0.15	0.11
s9234	250	0.40	0.28	0.23	0.21

Table 6: Elapsed time (seconds) for constructing BDDs on the IBM SP-2 using fanin-cone replication.

Circuits	Outputs	1 Processor	2 Processors	4 Processors	8 Processors	16 Processors
C1355	32	3.78	2.34	1.64	1.49	1.27
C1908	25	0.81	0.60	0.43	0.41	0.35
C7552	108	0.67	0.44	0.33	0.28	0.21
C499	32	1.31	0.79	0.75	0.39	0.33
C880	26	0.50	0.34	0.25	0.23	0.22
C3540	22	9.09	7.24	6.42	6.13	6.11
s5378	213	0.35	0.27	0.24	0.23	0.22
s9234	250	0.55	0.40	0.35	0.33	0.31

Table 7: Speedups for constructing BDDs on the IBM J-40 using fanin-cone replication.

Circuits	2 Processors	4 Processors	8 Processors
C1355	1.70	2.22	2.40
C1908	1.33	1.61	1.67
C7552	1.33	1.97	2.17
C499	1.82	1.85	3.37
C880	1.33	1.68	1.93
C3540	1.13	1.23	1.25
s5378	1.29	1.50	1.80
s9234	1.50	1.70	2.09

Table 8: Speedups for constructing BDDs on the SGI Origin 2000 using fanin-cone replication.

Circuits	2 Processors	4 Processors	8 Processors
C1355	1.66	2.37	2.57
C1908	1.40	1.78	1.87
C7552	1.44	2.07	2.43
C499	1.61	1.63	2.97
C880	1.50	1.88	2.05
C3540	1.23	1.23	1.40
s5378	1.35	1.53	2.10
s9234	1.43	1.74	1.90

Table 9: Speedups for constructing BDDs on the IBM SP-2 using fanin-cone replication.

Circuits	2 Processors	4 Processors	8 Processors	16 Processors
C1355	1.62	2.30	2.54	2.98
C1908	1.35	1.88	1.98	2.31
C2670	1.03	1.09	1.55	1.70
C7552	1.52	2.03	2.39	3.20
C499	1.66	1.75	3.33	3.87
C880	1.47	2.00	2.17	2.27
C3540	1.26	1.42	1.48	1.49
s5378	1.30	1.46	1.52	1.60
s9234	1.38	1.57	1.67	1.77

6 Conclusions and Future Work

In this paper, we have presented two parallel programs for BDD construction. Our algorithms are based on the depth-first BDD algorithm and fanin-cone partition. One algorithm is a shared memory algorithm using locks. We have implemented this algorithm on two shared memory multiprocessors. To get more parallelism, an algorithm using replication of fanin cones is also proposed. We have implemented this algorithm both on shared memory and distributed memory systems.

The experimental results show that our algorithm for shared memory systems is not very scalable in terms of processor numbers. However, for the distributed memory systems, some speedups and scalability can be achieved. We have found that some of the benchmark circuits have large logic overlaps among fanin cones. In our shared memory algorithm, the overlaps can cause serious contention among processors. In the distributed memory algorithm, processors will perform some redundant work due to the overlaps. In the view of that, we believe that a good BDD partition scheme with minimal overlap and communication overhead is needed to improve the efficiency of our algorithms.

As for the future work, we plan to find a better partition algorithm, such as the partitioned BDDs proposed in [16]. By using the partition scheme, we hope that the whole work can be scheduled to different processors with minimal redundancy and communication overhead.

References

[1] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers,* vol. C-35, pp. 8:677-691, 1986

[2] F. Brglez, and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and A Target Translator in Fortran," *IEEE International Symposium on Circuits and Systems,* vol. 3, June 1985

[3] F. Brglez, D. Bryant, K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *IEEE International Symposium on Circuits and Systems,* pp. 1929-1934, May 1989

[4] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Surveys,* vol. 24, no. 3, Sept. 1992

[5] K. Brace, R. Rudell, and R. E. Bryant, "Efficient Implementation of A BDD Package," *Proc. of the International Conference on Computer-Aided Design,* pp. 40-45, June 1990

[6] S. Kimura and E. M. Clarke, "A Parallel Algorithm for Constructing Binary Decision Diagrams," *IEEE Proc. of the International Conference on Computer Design,* pp. 220-223, Sept. 1990

[7] S. Kimura, T. Igaki, and H. Haneda, "Parallel Binary Decision Diagram Manipulation," *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences,* vol. E75-A. no. 10, pp. 1255-1266, Oct. 1992

[8] H. Ochi, K. Yasuoka, and S. Yajima, "Breadth-first Manipulation of SBDD of Boolean Functions for Vector Processing," *Proc. of the 28th ACM/IEEE Design Automation Conference,* pp. 413-416, June 1991

[9] Y. Parasuram, E. Stabler, and S. K. Chin, "Parallel Implementation of BDD Algorithms Using A Distributed Shared Memory," *Proc. of 27th Hawaii International Conference on Systems Sciences,* pp. 16-25, Jan. 1994

[10] S. Gai, M. Rebaudengo, and M. S. Recorda, "A Data Parallel Algorithm for Boolean Function Manipulation," *Proc. of Fifth Symposium on Frontiers of Massively Parallel computation,* pp. 28-34, Feb. 1995

[11] T. Sotrnetta and F. Brewer, "Implementation of An Efficient Parallel BDD Package," *Proc. of the 33rd ACM/IEEE Design Automation Conference,* pp. 641-644, June 1996

[12] R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Decision Diagrams on Network of Workstations," *IEEE Proc. of the International Conference on Computer Design,* Oct. 1996

[13] P. Arunachalam, C. Chase, and D. Moundanos, "Distributed Binary Decision Diagrams for Verification of Large Circuits," *IEEE Proc. of the International Conference on Computer Design,* pp. 365-370, Oct. 1996

[14] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in A Logic Synthesis Environment," *Proc. of the International Conference on Computer-Aided Design,* pp. 6-9, Nov. 1988

[15] T. Chan, "Distributed Automatic Test Pattern Generation," *Proc. Application Specific Integrated Circuits Conference (ASIC-92),* Sept. 1992

[16] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli, "Partitioned ROBDDs–A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions," *Proc. of the International Conference on Computer-Aided Design,* pp. 547-554 1996

[17] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *Proc. of the International Conference on Computer-Aided Design,* pp. 42-47, 1993