Parallel Algorithms for Logic Synthesis using the MIS Approach

Kaushik De[†] John A. Chandy[‡] Sumit Roy[‡] Steven Parkes[§] Prithviraj Banerjee[‡]

[†] LSI Logic Corporation
Milpitas, CA 95035
kaushik@lsil.com

[‡] University of Illinois
Urbana, IL 61801
{jchandy,sroy,banerjee}@crhc.uiuc.edu

[§] Sierra Vista Research
Santa Clara, CA 95055-3758
parkes@sierravista.com

Abstract

Combinational logic synthesis is a very important but computationally expensive phase of VLSI system design. Parallel processing offers an attractive solution to reduce this design cycle time. In this paper, we describe `ProperMIS`*, a portable parallel algorithm for logic synthesis based on the* `MIS` *multi-level logic synthesis system. As part of this work, we have developed novel parallel algorithms for the different logic transformations of the* `MIS` *system. Our algorithm uses an asynchronous message-driven computing model with no synchronizing barriers separating phases of parallel computation. The algorithm is portable across a wide variety of parallel architectures, and is built around a well-defined sequential algorithm interface, so that we can benefit from future expansion of the sequential algorithm. We present results on several MCNC and ISCAS benchmark circuits for a variety of shared memory and distributed processing architectures. Our implementation produces speedups of an average of 4 on 8 processors.*

1 Introduction

Combinational logic synthesis is the optimization of a logic design to realize a specific combinational function in either two level or multilevel form, and typically optimizes the area or delay of the resultant circuit. Efficient algorithms for two-level logic minimization include `ESPRESSO` [1] and for multilevel logic optimization, `SOCRATES` [2], `MIS` [3], `SYLON-XTRANS` [4], and `BOLD` [5].

Since logic synthesis is very compute intensive, parallel processing is fast becoming a desirable solution to reduce the large amounts of time spent in VLSI circuit design. This has been recognized by several researchers in VLSI CAD, as many have started to investigate parallel algorithms for problems in logic synthesis and verification [6, 7, 8, 9].

We recently developed a portable parallel algorithm for the transduction method [4] of logic synthesis [10], and results of the parallel algorithm were presented for a variety of parallel platforms. Even though we obtained reasonably good speedups using that algorithm, the original sequential algorithm using the transduction method has very large run times. The more popular logic synthesis algorithm is `MIS-II`, which is based on iterative factoring and simplification of nodes in a Boolean network. This algorithm forms the core of numerous university and industrial logic synthesis systems. A previous attempt to parallelize `MIS-II` resulted in poor speedups and significant loss in quality, because the `MIS-II` algorithm is inherently sequential in nature and extremely hard to parallelize [7].

In this paper, we therefore present `ProperMIS`, a new parallel `MIS-II` based algorithm for logic synthesis that uses an asynchronous message-driven computing model with no synchronizing barriers separating phases of parallel computation. Using the ProperCAD II system, the algorithm is portable across a wide variety of parallel architectures.

2 ProperCAD II Overview

Much of the work in parallel CAD reported to date suffers from a major limitation in that these proposed parallel algorithms are designed with a specific underlying architecture in mind. As a result, these applications perform poorly on architectures other than the one for which they were designed. Just as importantly, incompatibilities in programming environments make it difficult to port these programs across different parallel architectures. This limitation has serious consequences, since a parallel algorithm needs to be developed afresh for every target MIMD architecture.

One of the primary concerns of the ProperCAD project [11] is to address this portability problem by designing algorithms to run on a range of parallel machines including shared memory multiprocessors, distributed memory multicomputers, and networks of workstations. The ProperCAD approach to the design of parallel CAD algorithms is illustrated in Figure 1. A parallel algorithm is designed around an existing uniprocessor algorithm by identifying modules in the uniprocessor code and designing a well-defined interface between the parallel and sequential code.

The project has undergone two distinct phases, the first of which, ProperCAD I, involved the use of the C-based Charm language and runtime system [12]. The second phase, ProperCAD II [13, 14], entailed the creation of a C++ library which provided an object–oriented parallel interface based on the actor model of concurrent object–oriented computing.

The ProperCAD II library provides the mechanisms necessary for parallel execution in through the use of a fundamental object called an actor [15]. An actor object consists of a thread of control that communicates with other actors by sending messages, and all actor actions are in response to these messages. Specific actor

This research was supported in part by the National Science Foundation under grant MIP-9320854, the Semiconductor Research Corporation under grant SRC 94-DP-109, and the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research Office.

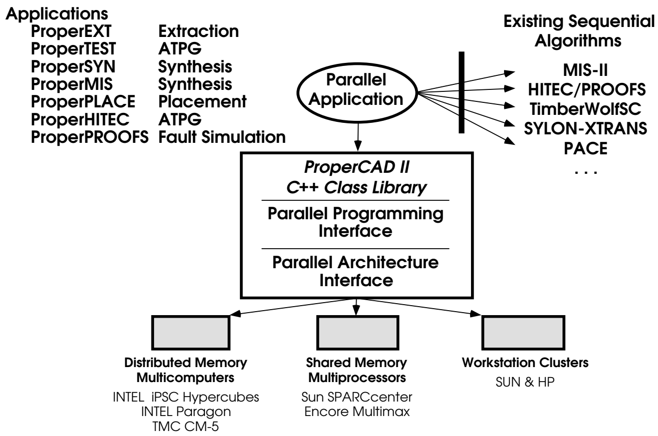

Figure 1: An overview of the ProperCAD project.

methods are invoked to process each type of message, and actors are not allowed to block or explicitly make receive requests from other processors. The runtime system on each processor picks the next available actor thread with some priority and that thread is then allowed to run to completion without interruption.

As part of ProperCAD, a suite of parallel applications have been developed that address the most significant tasks in VLSI design automation including circuit extraction, test generation, fault simulation, cell placement, and logic synthesis.

3 `MIS-II` Overview

3.1 Definitions

In this section, we will review some basic definitions as given in [3] to be used later in this paper.

A *variable* is a symbol representing a single coordinate of the Boolean space.

A *literal* is a variable or its negation.

A *cube* is a set C of literals such that $x \in C$ implies $\overline{x} \notin C$.

An *expression* is a set f of cubes.

The *support* of an expression f is the set of literals $sup(f)$ which are present in the sum-of-product expression of f.

An expression f is *cube-free* if no cube divides f evenly.

The *primary divisors* of an expression f form a set of expressions $D(f) = \{f/C | C \text{ is a cube}\}$.

The *kernels* of an expression f are the expressions $K(f) = \{g | g \in D(f) \text{ and } g \text{ is cube-free}\}$. In other words, the kernels of an expression f are the cube-free primary divisors of f.

The cube C used to obtain kernel $k = f/C$ is called the *co-kernel* of k.

A Boolean network is a directed acyclic graph (DAG) where each node i is associated with 1) a variable y_i and 2) a representation f_i of a logic function. In the graph, an arc connects node i to node j if $y_i \in sup(f_j)$. The *fan-in* of a node i is the set of all of the nodes pointing to node i. The *fan-out* of a node i is the set of all nodes which node i points to.

The *satisfiability don't care* set of a node i is defined to be $DSAT_i = y_i\overline{f_i} + \overline{y_i}f_i = y_i \bigoplus f_i$, where y_i is the variable representing the node i and f_i is the logic function for that node.

3.2 Types of Logic Transformations

The aim of *kernel extraction* is to search for multiple-cube common divisors and extract those divisors. There exists a multiple-cube common divisor δ for l nodes $\eta_1, \eta_2, \ldots, \eta_l$ if $\exists\ k_{\eta_1} \in K(\eta_1), k_{\eta_2} \in K(\eta_2), \ldots k_{\eta_l} \in K(\eta_l)$, such that $k_{\eta_1} \cap k_{\eta_2} \ldots \cap k_{\eta_l} = \delta$. For example, given the equations in a Boolean network

$$F = af + bf + ag + cg + ade + bde + cde,$$

$$G = af + bf + ace + bce, \quad H = ade + cde \tag{1}$$

the best multiple-cube divisor that can be found is $a + b$. After creating a new node X for the common divisor in the network, the equations in the network become

$$F = deX + fX + ag + cg + cde,$$

$$G = ceX + fX, \quad H = ade + cde, \quad X = a + b \tag{2}$$

The original network had 33 literals, and the modified network after the kernel extraction has 25 literals.

Cube extraction searches for single-cube common divisors and extracts those divisors. There exists a single cube common divisor ζ for m nodes, $\eta_1, \eta_2, \ldots, \eta_m$ if $c_1 \in F(\eta_1), c_2 \in F(\eta_2), \ldots, c_m \in F(\eta_m)$, such that $c_1 \cap c_2 \ldots \cap c_m = \zeta$, where $F(\eta_i)$ represents the Boolean expression for the node η_i. For example, given the following Boolean network equations

$$F = abc + abd + eg, \quad G = abfg \tag{3}$$

the best single cube common divisor that can be found is ab. After creating a new node X in the network, the equations in the modified network become

$$F = Xc + Xd + eg, \quad G = Xfg, \quad X = ab \tag{4}$$

This transformation reduces the number of literals from 12 to 11.

Resubstitution is used to check if an existing function itself is a divisor of another function. For example, consider the network:

$$x = ac + ad + bc + bd + e \ \text{ and } \ y = a + b \tag{5}$$

The function y itself is a divisor of the function x. Therefore, it can be used to simplify the function x, which can be rewritten as:

$$x = y(c + d) + e \tag{6}$$

Each node of a Boolean network is a Boolean function, expressed in the sum-of-products format (two-level logic). Hence, two-level minimization algorithms such as *espresso* [1] can be used to minimize each node of the Boolean network. That process is called *simplification*.

4 Parallelization Methodology

An immediately apparent approach to parallel logic synthesis is to divide the logic circuit into several partitions and then synthesize those partitions independently in parallel [16]. Unfortunately, that results in loss in quality of the synthesized circuit, because of the lack of global information. Instead of using such a naive method, in `ProperMIS`, we partition the circuit for the purpose of division of work among different processors, however, we do not synthesize those partitions independently. Logic minimization is performed on the entire network, so as not to lose the quality of the synthesized logic. Partitioning is used only for distribution of work among different processors, so it does not have any effect on the quality of the synthesized circuit.

After reading in the circuit, a very simple strategy based on the input cones of the primary outputs is used to create the partitions. For each partition π, an actor object denoted as $partition(\pi)$ is created. We create many small partitions such that load balancing is good, but we make sure that the amount of computation required by each partition is roughly an order of magnitude higher than that of communication time for sending a message between objects. The ProperCAD II run-time system automatically distributes these objects to different processors.

Each $partition(\pi)$ actor then initiates the optimization procedure by starting with the simplification procedure which is detailed in Section 4.3. When all of the nodes in a partition π are simplified, the $partition(\pi)$ actor is sent a message to generate kernels for the nodes in π. At this point the other transformations, kernel extraction, cube extraction, and resubstitution are begun as explained in Sections 4.1 and 4.2. We assign priorities to different transformations, with initial priorities ordered as follows (highest to lowest): simplification, kernel extraction, cube extraction and resubstitution. For example, a processor will not pick up any message regarding kernel extraction if some other message regarding simplification is waiting to be processed in that processor. These priorities are used to guide the synthesis process such that it avoids local minima.

In order to allow different processors to simultaneously perform conflicting optimizations on the network, we provide coherence among the parallel applications of these optimizations by using version numbers for the nodes. Every node in the Boolean network has a version number that is incremented by one whenever the functionality of the node changes due to some transformation. Therefore, when a processor finds a possible transformation on a node η, it asks permission from a designated *master* processor to make the transformation and it provides the version number of η in the request. The *master* processor checks if the version number provided in the request is the same as the current version number of η. If so, permission is granted because the functionality of η has not changed since it was checked for the possible transformation. Otherwise, permission is denied.

4.1 Parallel Algorithm for Extraction

In Section 3.2, we defined the serial kernel extraction process. In our parallel algorithm, all of the possible kernels for each node are generated concurrently. Using the kernel generation algorithm described in [1], each $partition$ actor will in parallel generate the kernels for all the nodes in its partition, and will then broadcast that information to all of the processors.

Consider the equations given in Eq. 1. The kernels (co-kernels) of the equation F are $de + f + g\ (a)$, $de + f\ (b)$, $a + b + c\ (de)$, $a + b\ (f)$, $de + g\ (c)$ and $a + c\ (g)$. The kernels of G are $ce + f\ (a, b)$, $a + b\ (f, ce)$, and the only kernel of H is $a + c\ (de)$. Finding useful intersections of kernels is facilitated with a data structure called the *co-kernel cube* matrix, as described in [17]. A row in this matrix corresponds to a kernel (and its associated co-kernel), and a column corresponds to a cube which is present in some kernel. The entry at position (i, j) is nonzero if kernel i contains the cube j. For reference, the cubes of the original expressions in Eq. 1 are numbered from 1 to 13.

			a	b	c	de	f	g	ce
			1	2	3	4	5	6	7
F	a	1	.	.	.	5	1	3	.
F	b	2	.	.	.	6	2	.	.
F	de	3	5	6	7	.	.	.	.
F	f	4	1	2	.	.	.	.	.
F	c	5	.	.	.	7	.	4	.
F	g	6	3	.	4	.	.	.	.
G	a	7	.	.	.	.	8	.	10
G	b	8	.	.	.	.	9	.	11
G	ce	9	10	11	.	.	.	.	.
G	f	10	8	9	.	.	.	.	.
H	de	11	12	.	13	.	.	.	.

(a)

			ce	f	a	b	c	de	g
			1	2	3	4	5	6	7
G	a	1	10	8	.	.	.	.	.
G	b	2	11	9	.	.	.	.	.
G	ce	3	.	.	10	11	.	.	.
G	f	4	.	.	8	9	.	.	.
H	de	5	.	.	12	.	13	.	.
F	a	6	.	1	.	.	.	5	3
F	b	7	.	2	.	.	.	6	.
F	de	8	.	.	5	6	7	.	.
F	f	9	.	.	1	2	.	.	.
F	c	10	.	.	.	.	.	7	4
F	g	11	.	.	3	.	4	.	.

(b)

Figure 2: Two co-kernel cube matrices for Eq. 1

In a parallel environment, the creation of the co-kernel cube matrix is a more involved process. We want to create the co-kernel cube matrix in all of the processors, and they have to be the same in all processors. In the sequential algorithm, when a new kernel (co-kernel) is generated, a new row is assigned for that kernel, and the row number corresponding to that kernel is noted in a table. Similarly, when a unique kernel-cube is generated, a new column is assigned for that cube and the column number corresponding to that cube is noted in a table.

Since kernels are generated in parallel and then broadcast to all of the processors, the order in which the kernels are received may vary in different processors. For example, one processor may receive the kernels in the order F, G and H. In that case, the co-kernel cube matrix will be the same as that given in Figure 2(a). However, if another processor receives the kernels in the order G, H and F, the co-kernel cube matrix in that processor will resemble the one given in Figure 2(b), which has different labeling for the rows and the columns.

To keep the row and column labeling consistent across all processors, each node is assigned a distinct interval of labels. For example, we can assign the intervals [1–1000], [1001–2000] and [2001–3000] to the nodes F, G and H respectively. So any kernel

			a	b	c	de	f	g	ce
			1	2	3	4	5	6	1003
F	a	1	.	.	.	5	1	3	.
F	b	2	.	.	.	6	2	.	.
F	de	3	**5**	**6**	7	.	.	.	.
F	f	4	**1**	**2**	.	.	.	.	.
F	c	5	.	.	.	7	.	4	.
F	g	6	3	.	4	.	.	.	.
G	a	1001	.	.	.	.	8	.	10
G	b	1002	.	.	.	.	9	.	11
G	ce	1003	**10**	**11**	.	.	.	.	.
G	f	1004	**8**	**9**	.	.	.	.	.
H	de	2001	12	.	13	.	.	.	.

Figure 3: Consistent co-kernel cube matrix for Eq. 1

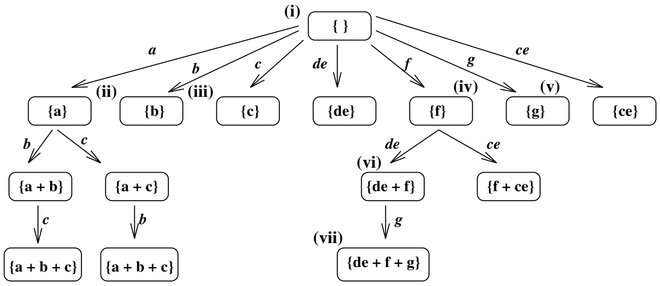

Figure 4: Conceptual search space for finding the maximum-valued rectangle

(co-kernel) generated for the node G will be labeled as a row in the co-kernel cube matrix with a number starting from 1001. Hence, the labeling of the rows will be consistent in all of the processors irrespective of the order in which the kernels are received. Using a similar strategy for column labeling will produce a co-kernel cube matrix as in Figure 3.

A rectangle of the co-kernel cube matrix identifies an intersection of kernels, that is, a common subexpression in the network. The columns of the rectangle identify the cubes in the subexpression, and the rows of the rectangle identify the particular functions that the subexpression divides. For example, the prime rectangle ({3, 4, 1003, 1004}, $\{1, 2\}$) in Figure 3 (shown in bold) identifies the subexpression $a + b$ which divides the function F and G as in Eq. 2. The cubes numbered 1, 2, 5, 6, 8, 9, 10, and 11 from the original set of functions are covered by this rectangle.

The *value* of a rectangle measures the reduction of the number of literals in the network if the particular rectangle is selected. Rectangle-covering is performed by generating all possible rectangles and then selecting the maximum valued rectangle. The search for the maximum-valued rectangle can be performed in parallel as conceptually illustrated in Figure 4 and illustrated with the help of a *co-kernel cube matrix* in Figure 5. After the generation of all of the kernels, a designated *master* processor creates an actor called *kernel_extract* to compute the maximum valued rectangle by generating all of the prime rectangles.

When a new *kernel_extract* actor is created, the following information is passed: the submatrix of the co-kernel cube matrix (M), the rectangle generated thus far ($rect$) and the index of the column from which it should search onwards for new prime rectangles ($index$). For the root *kernel_extract* actor, M is the co-kernel cube matrix itself, $rect$ is a empty rectangle and the value of $index$ is 0 as shown in Figure 5(i). The responsibility of this *kernel_extract* actor is to generate all of the prime rectangles with fewer rows but more columns than *rect*. That is performed by adding to *rect* columns with labels more than or equal to *index*. Conceptually, it is equivalent to generating all of the possible subexpressions which can be multiple cube common divisors to the given set of equations by adding more kernel cubes (columns of M) to the given subexpression (represented by *rect*). The *kernel_extract* actor also computes the value of *rect* and records the information if it is the best rectangle seen by this actor.

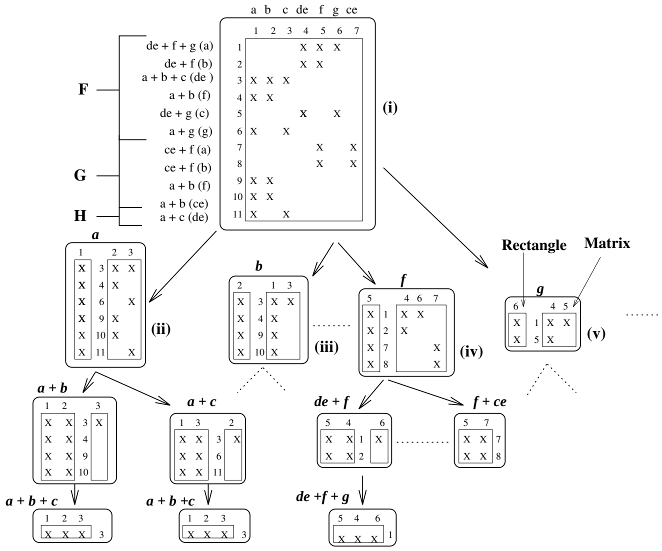

Figure 5: *Kernel_extract* actor search tree and associated matrices

If the number of columns in the submatrix M is less than a user-defined threshold, all of the possible prime rectangles are generated by applying the sequential algorithm described in [17]. Otherwise, new child actors are created and the search space is divided among those children. Each column c with a label greater than *index* is examined as a column to include in the rectangle. The submatrix $M1$ of the original matrix M is created by selecting only the rows in which column c has a nonzero value, and a new rectangle *rect1* is formed from the columns of the old rectangle *rect* and the rows for which c has a nonzero value. At this point, a new child *kernel_extract* actor is created with the following arguments: the submatrix $M1$, the new rectangle *rect1* and the index c. For example, by adding column 1 to the empty rectangle in the root object, we create a new rectangle *rect1* with one column and six rows $(3, 4, 6, 9, 10, 11)$ as shown in Figure 5(ii). This actor receives the submatrix $M1$, the rectangle $rect1$ as shown in Figure 5(ii) and the value of the parameter *index* will be 1. Conceptually, it is equivalent to adding the kernel cube a to the null subexpression in the root object as shown in Figure 4(ii).

The leaf nodes of the search tree generated prime rectangles by using the sequential algorithm and then reports the best rectangle seen to its parent after the computation is performed. Non leaf nodes, which created child actors, must wait for the best rectangles seen by its children before it can then forward the best rectangle

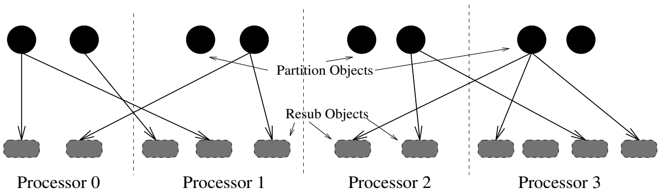

Figure 6: Creation of objects for resubstitution

to its parent. The root *kernel_extract* actor finally reports the best rectangle information to the *master* processor.

When the *master* processor receives the information regarding the maximum-valued rectangle, it computes the subexpression corresponding to that rectangle from the co-kernel cube matrix, and broadcasts that information to the other processors, which then remove the kernels of the affected expressions from the co-kernel cube matrix. The *version numbers* of the affected nodes are incremented by 1. A new root *kernel_extract* actor is created to process the new co-kernel cube matrix, and the process continues until no more positive valued rectangles can be found.

The parallel algorithm for *cube extraction* is similar to that for *kernel extraction*, and is not described further for lack of space. More details are available in [18].

4.2 Parallel Algorithm for Resubstitution

As described in Section 3.2, the resubstitution transformation is used to check if an existing expression itself is a divisor of other expressions. The resubstitution operation can be performed in parallel by creating for each node η *resub*(η) actors which are then distributed across all of the processors by the ProperCAD II run-time system as shown in Figure 6.

The responsibility of *resub*(η) is to search for possible resubstitution by other nodes in the node η. The search is restricted to algebraic divisors as the search for Boolean division is very expensive. A node ν can be an algebraic divisor of the node η if the support of ν is a subset of the support of η. Hence, *resub*(η) restricts its search for divisors to the nodes whose supports are subsets of the support of η. Anytime it finds a divisor ν and the division is going to reduce the literal count of the network, permission for the transformation must be requested from the *master* processor. The *master* processor verifies the currentness of the version number and, if granted, increments the version number. Since the functionality of the node η has changed, the *master* processor creates a new *resub* actor for the node η.

4.3 Parallel Algorithm for Simplification

Two-level minimization is a much more developed science than multilevel minimization, and very efficient algorithms exist for finding minimal two-level representations of Boolean functions. In the context of a multilevel network, two-level minimization can be made more powerful by providing the minimizer with a set of functions known as don't-care sets. These sets are derived from the structure of the network as well, and its size can determine how thorough and how fast the minimization process will be. In a sequential algorithm for simplification, since only one node is simplified at a time, the don't-care set of any other node can be used in the simplification process without error.

Since we are performing node simplifications in parallel, there must be constraints on which don't-cares can be used to ensure the results are correct. For example, consider the set of expressions

$$X = a\overline{b} + \overline{a}b \text{ and } Y = ab + \overline{a}\overline{b}$$

If both of the nodes are simplified concurrently and they use each other's satisfiability don't-care sets, then the result could be

$X = \overline{Y}$ and $Y = \overline{X}$

which is wrong. But if they are simplified one at a time, that will never happen. Then we would have

$X = a\overline{b} + \overline{a}b$ and $Y = \overline{X}$ or $X = \overline{Y}$ and $Y = ab + \overline{a}\overline{b}$

This simple example shows that two nodes being simplified concurrently should not use each other's satisfiability don't-care sets.

To make concurrent node simplification possible, we thus have to introduce the concept of *locking* a node. A node is said to be *algebraically locked* if no change in functionality is allowed to that node. A node is said to be *Boolean locked* if simplification of that node is not allowed, but all algebraic operations such as kernel extraction, cube extraction and resubstitution are allowed.

Conceptually, the satisfiability don't care of any node can be used for simplification of another node as long as it does not form a cycle in the network. To perform the simplification of nodes in parallel, we must find the largest set of nodes such that simplification of those nodes can be performed concurrently. Since this problem is NP-hard, we use a heuristic as described below.

Each $partition$ actor creates for each node η in its partition a *simp*(η) actor that is responsible for simplification of that node. As with the *resub* actors, the *simp* actors are distributed across different processors by the ProperCAD II run-time system.

Upon creation, each *simp*(η) actor asks for permission from the *master* processor for the simplification of η. Initially all of the nodes are unlocked. When the *master* processor picks up the permission request, it checks if η is Boolean locked. If it is, the *master* processor denies permission and asks *simp*(η) to try later. Otherwise, the don't-care set is checked to determine if any of the nodes are are algebraically locked. If so, depending on the type of don't-care set used, the *master* processor will deny permission or simply supply a reduced don't-care set. If permission is granted, again depending on the don't-care set, the nodes in the set are Boolean or algebraically locked. After simplification is complete, the other processors are notified and the *master* processor will unlock the appropriate locks.

It is very apparent from the description of the simplification procedure that the first node to receive permission can use all of the don't-care sets it asks for. Therefore, for better simplification, we assigned different priorities proportional to the size of the nodes (in terms of the literal count) to different *simp* actors. If two *simp* actors ask permission from the *master* processor simultaneously, the larger node will be processed first. With such a procedure, the don't-care sets of those nodes will be available for use, potentially improving the quality of node simplification.

5 Results and Discussion

In this section, we will discuss results obtained by applying `ProperMIS` on various MCNC and ISCAS benchmark circuits.

In Table 1, we compare the quality of circuits obtained by `MIS-II` with that of the `ProperMIS` algorithm in a uniproces-

Table 1: Comparison between `MIS-II` and `ProperMIS`

Circuit	Literal Count			Run Time (sec)	
	Initial	`MIS`	`Proper MIS`	`MIS`	`Proper MIS`
bw	424	178	186	30.2	32.4
C499	616	558	598	12.8	28.6
duke2	1746	453	461	40.7	71.4
i8	4626	1354	1152	84.4	81.1
i9	1453	614	614	24.4	17.8
misex3	1661	563	445	97.7	247.9
misex3c	1694	506	522	90.3	99.6
pair	2667	2180	1881	89.7	75.2
x3	1816	862	839	61.0	70.8

Table 2: Results for the Intel Paragon

Circuit	1 Processor		4 Processors		8 Processors	
	Lit	Time	Lit	Spd	Lit	Spd
bw	175	27.0	183	5.4	184	10.1
C499	598	23.3	600	4.3	599	6.9
duke2	468	58.1	516	2.1	539	2.8
i8	1155	61.6	1471	2.3	1414	2.6
i9	614	12.8	624	2.3	623	2.3
misex3	528	75.1	527	1.8	554	3.4
misex3c	522	87.2	566	1.7	560	2.0
pair	1881	52.3	1903	1.5	1922	1.8
x3	857	38.0	908	1.4	904	1.8

Table 3: Results for the Encore Multimax

Circuit	1 Processor		4 Processors		8 Processors	
	Lit	Time	Lit	Spd	Lit	Spd
bw	182	176.1	190	3.5	199	4.2
C499	598	182.4	598	6.8	598	6.1
duke2	467	171.0	496	2.2	596	7.0
i8	1151	280.5	1172	2.8	1159	4.3
i9	614	57.0	643	2.3	751	2.3
misex3	440	1590.6	501	1.5	481	3.8
misex3c	526	425.1	534	2.2	590	3.4
pair	1881	106.7	1912	2.4	1896	2.6
x3	840	220.0	864	4.8	940	11.0

sor environment. The parameters and don't-care sets supplied to `MIS-II` and `ProperMIS` are identical for any circuit. The first 3 columns compare the literal counts of the initial circuits, the circuits produced by `MIS-II` and the circuits produced by `ProperMIS`, respectively. The quality of the circuits produced by `ProperMIS` is almost the same (sometimes better) as that of circuits produced by `MIS-II`. The last two columns compare the runtimes for `MIS-II` and `ProperMIS` for a Sun 4/690MP (runtimes are given in seconds). Again, in most cases the runtimes are comparable. The runtimes and qualities are not identical due to the nondeterminism in the order of applying various transformations.

Tables 2, 3 and 5 present the results obtained on a distributed memory MIMD machine, the Intel Paragon, as well as two shared memory machines, an 8-processor Encore Multimax and a 4-processor Sun 4/690MP server. Table 4 shows the results for a network of SPARCstation 5 workstations. For different numbers of processors, the quality of the synthesized circuits in terms of the literal count and the run times in seconds and speedups are shown for some benchmark circuits.

From the tables it is clear that the `ProperMIS` algorithm produces good speedups, and maintains quality comparable to that of the `MIS-II` approach. Some of the superlinear speedups are due to anomalies in parallel search techniques. One can observe that the maximum degradation of quality with multiple processors is generally less than 10%. The quality is sometimes improved with a larger number of processors, again, due to nondeterminism. Measurements of the parallel characteristics of `ProperMIS` are shown in Table 6 for *C499* on the Sun 4/690MP. While the user time to system time ratio is good, load balancing, however, is poor, and we will be addressing that issue in future research.

6 Conclusions

In this paper, we have presented an asynchronous portable parallel algorithm for logic synthesis, called `ProperMIS`, implemented as part of the ProperCAD project. As part of this work, we have developed novel parallel algorithms for the *kernel extraction*, *cube extraction*, *resubstitution*, and *node simplification* procedures of the `MIS-II` system. `ProperMIS` uses an asynchronous message-driven actor model of computation; no explicit synchronizing barriers are allowed in the algorithm. We have implemented the algorithm and have evaluated it on various MCNC and ISCAS synthesis benchmark circuits. The results, reported on a variety of parallel architectures, show good speedups with almost no degradation in the quality of the synthesized network over the uniprocessor algorithm.

By retaining about 80% of the sequential code from `MIS-II`, we were easily able to update `ProperMIS` to the latest version of `MIS`, now part of `SIS 1.2` [19]. Preliminary results using the new version are shown in Table 7. We are in the process of evaluating the reasons for the difference in results between `ProperMIS` and `ProperSIS`. We are also currently investigating parallelizing other aspects of logic synthesis present in `SIS 1.2`.

Overly large circuits can not be handled by `ProperMIS` due to excessive memory requirements. In light of this, we have conducted research to partition those circuits and then synthesize each partition separately in parallel. In doing so, we have designed innovative partitioning and redistribution methods that do not sacrifice the quality of the synthesized logic [16].

7 Acknowledgement

We are grateful to Dr. Balkrishna Ramkumar for various discussions on the development of parallel algorithms, and to John G. Holm for his assistance in characterizing the parallel behavior of `ProperMIS`. We are also thankful to the San Diego Supercomputing Center for granting us access to their Intel Paragon.

References

[1] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, "ESPRESSO-II: A new logic minimizer for program-

Table 4: Results for a cluster of SPARCstation 5s

Circuit	1 Processor		2 Processors		4 Processors	
	Lit	Time	Lit	Spd	Lit	Spd
bw	186	18.3	191	1.2	187	4.3
C499	598	16.7	598	1.3	599	2.4
duke2	461	46.3	531	0.9	521	1.4
i8	1153	49.1	1244	1.0	1273	1.1
i9	614	11.4	617	1.3	620	1.4
misex3	445	137.9	481	1.1	508	1.8
misex3c	522	60.4	572	0.8	687	0.8
pair	1881	47.9	1982	0.6	1905	0.9
x3	839	31.4	865	0.9	893	0.8

Table 5: Results for a Sun 4/690MP

Circuit	1 Processor		2 Processors		4 Processors	
	Lit	Time	Lit	Spd	Lit	Spd
bw	186	32.4	191	1.1	186	5.0
C499	598	28.6	598	1.4	600	4.3
duke2	461	71.4	512	1.1	519	1.6
i8	1152	81.1	1246	1.5	1281	2.4
i9	614	17.8	619	1.9	622	2.3
misex3	445	247.9	480	1.0	508	2.3
misex3c	522	99.6	588	1.4	565	1.9
pair	1881	75.2	1900	1.4	1967	3.9
x3	839	70.8	872	2.2	896	6.1

Table 6: Characteristics of `ProperMIS`.

	1 PE	2 PEs		4 PEs			
No. Actors	1732	1681		1779			
No. Msgs	4423	5876		6402			
Grain Size (ms)	6.56	3.75		3.76			
	P0	P0	P1	P0	P1	P2	P3
Idle Time (%)	4	55	12	50	19	27	35
System Time (%)	3	4	3	9	5	5	4
User Time (%)	93	41	85	41	76	68	60

Table 7: `ProperSIS` Results for a Sun 4/690MP

Circuit	1 Processor		2 Processors		4 Processors	
	Lit	Time	Lit	Spd	Lit	Spd
bw	182	39.1	186	1.4	180	4.5
C499	598	34.5	600	1.8	601	4.1
duke2	459	63.6	512	1.2	513	1.9
i8	1158	81.3	1225	1.6	1240	2.2
i9	614	20.6	629	1.8	622	2.2
misex3	449	241.1	469	1.1	516	1.9
misex3c	522	94.9	514	1.2	568	1.8
pair	1891	74.2	1931	1.4	1912	1.6
x3	907	44.2	957	1.2	1136	1.2

mable logic arrays," in *Proceedings of the Custom Integrated Circuits Conference*, pp. 370–376, June 1984.

[2] A. J. de Geus and W. Cohen, "A rule-based system for optimizing combinational logic," *IEEE Design & Test of Computers*, pp. 22–32, Aug. 1985.

[3] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1062–1081, Nov. 1987.

[4] X.Xiang, "Multilevel logic network synthesis systems, SYLON-XTRANS." Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1990.

[5] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft, "The Boulder Optimal Logic Design system," in *Digest of Papers, International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 62–65, Nov. 1987.

[6] R. Galivanche and S. M. Reddy, "A parallel PLA minimization program," in *Proceedings of the Design Automation Conference*, (Miami Beach, FL), pp. 600–607, June 1987.

[7] G. G. Zipfel, "A parallel algorithm for algebraic factorization with application to multi-level logic synthesis." M.S. thesis, University of Illinois at Urbana-Champaign, June 1991. Tech. Rep. CRHC–91–24/UILU–ENG–91–2234.

[8] G. D. Hachtel and P. H. Moceyunas, "Parallel algorithms for boolean tautology checking," in *Digest of Papers, International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 422–425, Nov. 1987.

[9] H.-K. T. Ma, S. Devadas, and A. Sangiovanni-Vincentelli, "Logic verification algorithms and their parallel implementations," in *Proceedings of the Design Automation Conference*, (Miami Beach, FL), pp. 283–290, June 1987.

[10] K. De, B. Ramkumar, and P. Banerjee, "A portable parallel algorithm for logic synthesis using transduction," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 566–580, May 1994.

[11] B. Ramkumar and P. Banerjee, "ProperCAD: A portable object-oriented parallel environment for VLSI CAD," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 829–842, July 1994.

[12] L. V. Kalé, "The Chare Kernel parallel programming system," in *Proceedings of the International Conference on Parallel Processing*, (St. Charles, IL), pp. 17–25, Aug. 1990.

[13] S. Parkes, J. A. Chandy, and P. Banerjee, "A library-based approach to portable, parallel, object-oriented programming: Interface, implementation, and application," in *Supercomputing '94*, (Washington, DC), pp. 69–78, Nov. 1994.

[14] S. M. Parkes, "A class library approach to concurrent object-oriented programming with applications to VLSI CAD." Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Sept. 1994. Tech. Rep. CRHC-94-20/UILU-ENG-94-2235.

[15] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: The MIT Press, 1986.

[16] K. De and P. Banerjee, "Parallel logic synthesis using partitioning," in *Proceedings of the International Conference on Parallel Processing*, (St. Charles, IL), pp. III:135–142, Aug. 1994.

[17] R. L. Rudell, "Logic synthesis for VLSI design." Ph.D. Dissertation, University of California, Berkeley, 1989.

[18] K. De, "Parallel algorithms for logic synthesis." Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Sept. 1993. Tech. Rep. CRHC-93-20/UILU-ENG-93-2235.

[19] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Tech. Rep. UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, May 1992.