PARALLEL ALGORITHMS FOR LOGIC SYNTHESIS

BY

KAUSHIK DE

B.Tech., Indian Institute of Technology, Kharagpur, 1987
M.S., University of Illinois at Urbana-Champaign, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

PARALLEL ALGORITHMS FOR LOGIC SYNTHESIS

Kaushik De, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1994
Prithviraj Banerjee, Advisor

The size of the VLSI circuit is increasing at a very rapid pace, and soon the sequential algorithms running on a uniprocessor will be inadequate to handle such large circuits. Parallel processing can be used to reduce the computation time considerably with almost no degradation in the quality. Most of the parallel algorithms devloped for VLSI CAD applications, however, are designed for one specific parallel architecture. As a result, considerable effort and expense are needed to port them to different parallel machines. The ongoing ProperCAD project at the University of Illinois offers a bright solution to that problem by allowing the user to develop parallel algorithms on the top of a portable framework such that the programs developed will run *unchanged* on a variety of parallel machines, both *shared* and *distributed* memory machines. In this thesis, parallel algorithms for combinational logic synthesis using two approaches are developed: 1) the Transduction method, which uses the concept of *a set of permissible functions* to perform various logic transformations to reduce the size of logic circuit, and 2) the MIS approach, which uses *algebraic factoring* and *node simplification* for the purpose of logic minimization. The parallel algorithms developed in this thesis offer three major contributions. First, the parallel algorithms use an *asynchronous*, message-driven computing model with no synchronizing barriers separating phases of parallel computation. Second, these algorithms are *portable* across a wide variety of parallel architectures, shared memory machines such as Encore Multimax and Sequent Symmetry, distributed memory machines such as Intel/860, and networks of workstations. Finally, these algorithms are built around well defined *sequential algorithm interfaces*, so that the parallel algorithms can benefit from the future improvements and expansions of the sequential algorithms. Very large circuits, however, can not be handled as a whole by any synthesis algorithm. Those circuits are partitioned, and then the partitions are synthesized independently in parallel. A parallel synthesis system based on the partitioning approach is also described in this thesis.

DEDICATION

TO MY PARENTS

ACKNOWLEDGMENTS

I would like to thank by advisor, Professor Prithviraj Banerjee, for his encouragement, advice, and assistance during this research. I would like to express my appreciation to Professors W. Kent Fuchs, Ibrahim Hajj, Saburo Muroga and Janak Patel for serving on my thesis commitee.

I am grateful to Balkrishna Ramkumar for numerous discussions on parallel algorithms. I thank Sungho Kim and Chieng-Fai Lim for discussions and help regarding CAD algorithms. I am indebted to all of my friends and colleagues at the Center for Reliable and High-Performance Computing and the Coordinated Science Laboratory for their friendship and support.

I would like to express my thanks to Professor L. V. Kale for letting me use the CHARM system, Argonne National Laboratory for letting me use their parallel machines, and the researchers at the University of California, Berkeley, for access to their MIS software.

Finally, I must thank my family for their love and support during my years as a graduate student.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

With the increasing complexity of VLSI circuits, existing Computer-Aided Design (CAD) algorithms for VLSI running on uniprocessors will be inadequate for handling large circuits in a reasonable amount of time. During the last decade, the number of components in VLSI chips has doubled almost each year, and this growth in chip size is expected to continue for another decade due to continued advancements in the fabrication process. At the end of this decade, the number of transistors in a single VLSI chip is expected to reach 100 million.

The computational speed of uniprocessors has not grown at that high rate. Most of the CAD algorithms for VLSI applications have exponential computational complexities. Many of the heuristics algorithms, designed to speed the computation by sacrificing the quality of the solution, have computational complexities which are more than linear in nature. Hence, to cope with the future computational needs, the uniprocessor speed has to grow at a rate which is greater than the rate of growth of the number of components in a VLSI chip. The uniprocessor speed is not projected to grow at that rapid pace, and soon the uniprocessors will not be adequate to handle the large circuits for VLSI applications.

Parallel processing is fast becoming an attractive solution to reduce the large computation time spent in VLSI circuit design. This has been recognized by several researchers in VLSI CAD as is evident in the recent literature [1]. Parallel algorithms have been developed for various areas of VLSI CAD, such as placement and routing [2, 3, 4, 5, 6], circuit extraction [7], circuit simulation [8, 9], logic simulation [10, 11], and test generation and fault simulation [12, 13].

Combinational logic synthesis deals with the optimization of logic to realize a specific combinational function in either two-level or multilevel form, and typically optimizes the area or delay of the resultant circuit. Many efficient algorithms for two-level logic minimization, such as ESPRESSO [14], and for multilevel logic optimization, such as SOCRATES [15], MIS [16], SYLON-XTRANS [17], and BOLD [18], have been developed recently. Logic synthesis for medium-sized circuits requires large computing times and memory. For larger circuits of the future, sequential algorithms will not be able to address their complexity.

Due to the very high computational complexity, researchers have started to investigate parallel algorithms for the problems in logic synthesis and verification. Galivanche and Reddy [19] have proposed a parallel implementation of two-level network synthesis based on ESPRESSO. The authors presented parallel versions of the procedures in two-level synthesis, such as reduction, expansion and complementation of cubes, and reported speedups of about 7 on an 8-processor shared memory multiprocessor. Zipfel has proposed a parallel multilevel synthesis algorithm using algebraic factorization procedures of MIS in [20]. He has developed parallel algorithms for kernel extraction and cube extraction procedures within the MIS synthesis program; however, the results of speedups on an 8-processor shared memory multiprocessor were less than 2 on 8 processors. Hachtel and Moceyunas [21] have proposed a parallel tautology checking algorithm with application to logic verification. Ma et al. [22] have proposed parallel logic verification algorithms based on implicit enumeration. Both of the above works on parallel logic verification were proposed for shared memory multiprocessors and reported good speedups. Lim et al. have recently proposed a parallel logic synthesis algorithm using the Transduction method [23], [24], suitable for execution on a shared memory multiprocessor. Speedups of about 6 were reported on an 8-processor Encore multiprocessor.

Much of the work in parallel algorithms for CAD reported to date, however, suffers from a major limitation, namely, that they are designed with a specific underlying parallel architecture in mind. Moreover, incompatibilities in programming environments also make it difficult to port these programs across different parallel machines. As a result, these programs perform poorly on architectures other than the one for which they were designed. For example, none of the above parallel algorithms for logic synthesis and verification can be extended on a distributed memory multicomputer such as Intel hypercubes or Ametek meshes.

This limitation has serious consequences. A parallel algorithm has to be developed afresh for every target parallel architecture. This is compounded by the length of the software development cycle, which, for parallel programs, is considerably longer than for sequential programs. As a result, parallel programs are considerably costlier to develop than sequential programs. Hence, when a new and better parallel machine comes on the market, a program developed for a different architecture can not be ported easily to that new machine without spending a considerable amount of time and money.

Hence, we plan to develop parallel algorithms for logic synthesis in such a fashion that they will be portable across a variety of parallel machines. As a result, the programs developed will run unchanged on those machines, and, if a parallel machine is upgraded to a better machine, the programs can be ported to the new machine almost without any effort. To achieve that goal, we plan to develop our algorithms as a part of the ongoing ProperCAD project at the University of Illinois [25]. In the next subsection, we will provide a brief overview of the ProperCAD project.

The two most popular approaches for combinational logic synthesis are 1) the Transduction method [17] and 2) the MIS approach [16]. In this thesis, we will develop parallel algorithms for both of the approaches using the CHARM runtime system such that they are portable across a variety of parallel machines. We will develop asynchronous message-driven parallel algorithms, with no synchronizing barriers between various phases of computation. This feature will enhance the scalability of the algorithms and will produce better speedups.

However, but very large circuits cannot be handled by most existing combinational synthesis algorithms due to excessive run times and prohibitive memory requirements.

For those circuits, the only solution is to partition them and synthesize each partition independently. We plan to synthesize the partitions in parallel. However, the quality of the synthesized circuit is not optimal because the partitions are synthesized independently. Hence, the partitioning algorithm has a big impact on the quality of the synthesized circuit. Also, the completion time for parallel synthesis is bounded by the longest completion time among all of the partitions. Thus, the partitioning algorithm also has a large effect on the run time. In this thesis, we will investigate the development of a partitioning algorithm which can produce a good quality synthesized circuit and also reduce the run time by minimizing the maximum synthesis time among all of the partitions. We will also develop a parallel logic synthesis system using this partitioning approach.

1.2 Overview of the ProperCAD Project

There are two goals of the ProperCAD project [25] (Portable object-oriented parallel environment for CAD applications). The first goal is to develop an environment for writing portable parallel programs for VLSI CAD applications that will run on a range of parallel machines including shared memory multiprocessors, such as the Encore Multimax and Sequent Symmetry, distributed memory multicomputers, such as the Intel iPSC/860, and networks of workstations. This is being accomplished in two phases. The first phase involves the use of the CHARM language and run-time system, which is based on extensions to the C language [26, 27] on top of which we developed CAD application-specific libraries to support the development of parallel CAD applications. The second phase involves the integration of the CAD libraries and the machine-independent run-time support in C++ in a truly object-oriented manner. The other goal of ProperCAD is to develop new parallel algorithms for CAD around existing sequential algorithms with a strict interface defining the interactions between the two. This will enable the parallel algorithms to benefit from future improvements in the sequential algorithms.

The ProperCAD approach to the design of parallel CAD algorithms is illustrated in Figure 1.1. 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 codes.

Parallel execution in the ProperCAD environment, based on the CHARM run-time system [26, 27], follows a coarse-grained data-flow model and is message driven. Conceptually, a pool of messages is created in which each message may either request the creation of a new *object* or be destined for an existing *object*. An *object* can only create new messages or be awakened by messages sent to it at designated entry points. Processes are not allowed to block or explicitly make receive requests from other processors.

Every message thus represents some work. The machine-specific code (see Figure 1.1) is responsible for dynamically distributing the messages across the available processors. A choice of queuing strategies is also available to the programmer to influence the order in which messages in the pool are processed. The strategy may be depth-first, breadth-first, or priority-based. In the priority-based strategies, the programmer provides priorities with every message, which determines the order in which the messages in the pool are processed.

The ProperCAD approach has already been applied to the following CAD applications: Circuit Extraction [28], Test Generation [29], and Placement [30].

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 describes the portable parallel algorithm for logic synthesis using the Transduction method [17]. The chapter first provides a brief overview of the Transduction method, then describes the parallel algorithm, and finally gives the experimental results obtained. Chapter 3 describes the portable parallel algorithm we have developed for algebraic factoring and node simplification in logic synthesis, namely, the MIS approach [16]. The chapter also provides a brief overview of the MIS approach. Chaper 4 describes the partitioning approach of parallel logic synthesis. In this chapter, we provide a new partitioning algorithm which uses a heuristics method to make the synthesis time of all of the partitions approximately equal. We

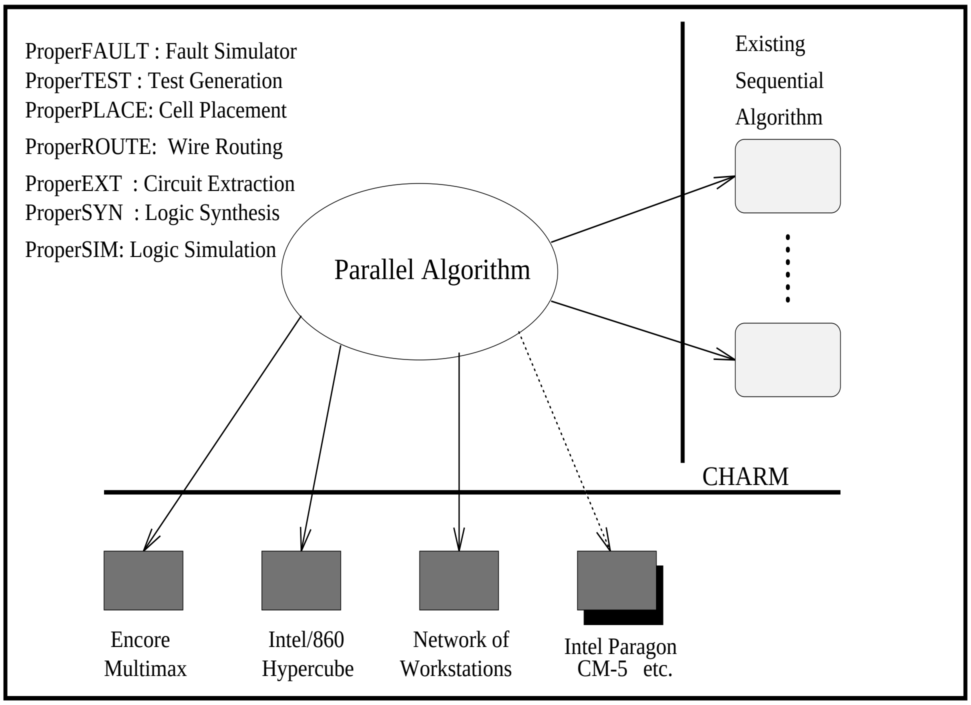

Figure 1.1 An overview of the ProperCAD project, listing the application under development

also described an iterative method to improve the quality of the synthesized circuit. In Chapter 5, we summarize the work presented in this thesis and give directions for future research in this area.

CHAPTER 2

PARALLEL ALGORITHM FOR THE TRANSDUCTION METHOD

2.1 Introduction

In this chapter, we present a new parallel algorithm, named ProperSYN, for logic synthesis using the Transduction method for combinational logic synthesis [17] as a part of the ProperCAD project, based on the CHARM system [26, 27]. The successful implementation of SYLON-XTRANS [31] and its variants [32] has proven that the Transduction method is a powerful tool in the synthesis of multilevel networks. However, the Transduction method has the disadvantage of large execution times for the large networks. We believe that an efficient parallel algorithm for logic synthesis using the transduction method will make the run times manageably small.

The parallel algorithm for logic synthesis in the ProperSYN system is novel and different from the parallel algorithm described by Lim et al. [23], [24]. Lim's algorithm is suitable for shared memory multiprocessors and uses a shared circuit description over all processors and explicit barrier synchronization between phases of transformations. Our algorithm uses an asynchronous message-driven computational model with no synchronizing barriers separating phases of parallel computation, and is therefore inherently more scalable to a larger number of processors. Scalability also depends on other important aspects such as *load balancing* and *scheduling*. To achieve good load balancing and scheduling, we create an order of magnitude more tasks than the number of processors available. Those tasks are automatically distributed by the CHARM run-time system. Since the number of tasks is large and the size of each task is small, the average work performed by each processor tends to be the same. While creating tasks, we took into consideration the communication-to-computation ratio to make sure that the communication overhead remains low. Also, our algorithm is portable across a wide variety of parallel architectures, both *shared* and *distributed* memory architectures. Finally, our algorithm is built around a well-defined sequential algorithm interface, so that we can benefit from future expansion of the sequential algorithm. The preliminary results of this research were published in [33], and the complete work has been submitted for publication [34].

The rest of the chapter is organized as follows. Section 2.2 provides an overview of the Transduction method of logic synthesis. Section 2.3 describes the portable parallel algorithm for logic synthesis based on the Transduction method developed in the ProperSYN system. Section 2.4 describes the experimental results obtained by applying ProperSYN on a variety of parallel machines.

2.2 Overview of the Transduction Method

2.2.1 Terminology and notations

The Transduction method considers only cycle-free multilevel circuits consisting of AND, OR, NAND, NOR and NOT gates. Let n be the number of primary inputs, m be the number of primary outputs, and g be the number of gates in a multilevel circuit. Let $X = \{x_1, x_2, \ldots, x_n\}$ be the set of *input variables* and $Z = \{z_1, z_2, \ldots, z_m\}$ be the set of *output variables* of the circuit. In addition, let $V = \{v_1, v_2, \ldots, v_g\}$ be the set of gates in the circuit, and $C = \{c_{ij}\}$ be the set of connections where c_{ij} connects the output of gate v_i to the input of gate v_j.

A gate v_i is an *immediate predecessor* of v_j if there exists a connection c_{ij}. Conversely, v_j is an *immediate successor* of v_i if c_{ij} exists. Let $IP(v_i)$ and $IS(v_i)$ be the set of all immediate predecessors and immediate successors of the gate v_i, respectively. When there is a sequence of gates v_{k_1}, v_{k_2}, ..., v_{k_t} such that $v_{k_{b+1}} \in IS(v_{k_b})$ for all $b = 1, 2, \ldots, t-1$, then v_{k_t} is a *successor* of v_{k_1}. Similarly defined, v_{k_1} is a *predecessor* of v_{k_t}. Let $P(v_i)$ and $S(v_i)$ denote the set of predecessors and successors of the gate v_i, respectively. A gate v_i is said to have a *reconvergent fanout* (or is *reconvergent*) if there exist two distinct gates v_{k_1}, $v_{k_2} \in IS(v_i)$ such that $S(v_{k_1}) \cap S(v_{k_2}) \notin \emptyset$.

An *output function* at a gate is the set of values output by a gate in a circuit for all possible combinations of input variables. This is also very frequently referred to as the function at the gate for short. The function at a gate v_i, $f(v_i)$, can be expressed as a vector of Boolean values. For example, if n = 3 and v_i is an AND gate with x_1, x_2 and x_3 as its inputs, where $f(x_1) = (01010101)$, $f(x_2) = (00110011)$, and $f(x_3) = (00001111)$, then $f(v_i) = (00000001)$. Also, if G is a Boolean vector, let $G^{(d)}$ be the dth value in G. The value of d can range from 1 to 2^n inclusive. In the Transduction Method, connections are often treated as gates. Hence, the function at a gate is also extended to cover that of a connection, which is defined by $f(c_{ij}) = f(v_i)$.

The function at a gate v_i can sometimes be expressed not only in terms of the input variables, but also as the function at some other gate v_k in the circuit. This is denoted by $f(v_i|v_k)$ and is called the function at v_i *with respect to* v_k. In this case, the gate v_k is treated as if it were an input variable, ignoring the functions at its input connections.

Very frequently, the function at a gate can be changed without affecting the function at the primary outputs. A *permissible function* at a gate is a function which the output of the gate can take for this purpose. For example, in Figure 2.1, for n= 3, $f(x_1)$ = (01010101), $f(x_2) = (00110011)$, $f(x_3) = (00001111)$, $f(v_1) = (11101110)$ and $z_1 = f(v_2)$ = (10111011). However, if $f(v_1)$ is changed to (01101110), z_1 is still unchanged. Hence, (01101110) is a permissible function of v_1.

The vector (01100110) is another permissible function of v_1. To present these two permissible functions effectively, a don't care value "*" is used. This is used to mean either a "0" or "1" value. Hence, (0110*110) represents both (01101110) and (01100110). A collection of permissible functions is known as a *set of permissible functions (SPF)*, which is discussed in the next subsections.

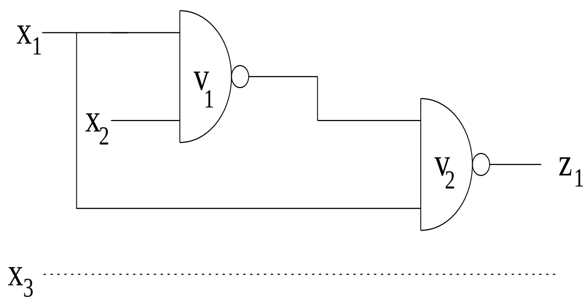

Figure 2.1 An example of a permissible function

2.2.2 Maximum set of permissible functions

As the name suggests, the *maximum set of permissible functions (MSPF)* of a gate in a circuit is the set that contains all possible permissible functions of the gate. The procedure of computing the MSPFs of gates and connections in a multilevel circuit containing only NOR gates is described in [35] and it is extended to OR, AND, NAND and NOT gates in [17].

Before formally describing the methods of computing MSPFs, the concept is explained with an example. From Figure 2.1, we have $f(x_1) = (01010101)$, $f(v_1) = (11101110)$, and $f(v_2) = (10111011)$. Since the output $z_1 = f(v_2)$ must remain constant, $MSPF(v_2)$ $= f(v_2) = (10111011)$. Let the first bit of the vectors to be the leftmost bit. Considering these bits, $x_1 = 0$ and $f(v_1) = 1$. Since v_2 is a NAND gate and x_1 is 0, $f(v_2)$ is always 1 regardless of the value of $f(v_1)$. Hence, the first bit of the MSPF of v_1 is "*". Similarly, the rest of the MSPF bits of v_1 can be computed, and this vector is found to be (*1*0*1*0).

First, we will describe the procedure of computing the MSPF of a connection, c_{ij}. Consider a portion of a circuit which contains c_{ij} as shown in Figure 2.2.

Suppose that the functions at all of the connections c_{xj} for $1 \leq x \leq k$ and at v_j are known. Let the MSPF of v_j be $MSPF(v_j)$ and suppose that it is known too. If v_j is a NOR gate, the dth bit of the vector $MSPF(c_{ij})$, $MSPF^{(d)}(c_{ij})$ is then given by

$$MSPF^{(d)}(c_{ij}) \;=\; F^{(d)} \#_{NOR} MSPF^{(d)}(v_j) \tag{2.1}$$

where the operator $\#_{NOR}$ is defined in Table A in Figure 2.3, F $= \cup_{1 \leq x \leq k, x \neq i} f(c_{xj})$, and $\cup$ is the normal Boolean OR operator. Similarly, for the cases in which v_j is an OR, AND

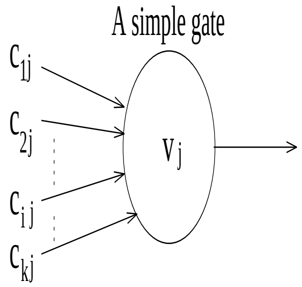

Figure 2.2 Computation of the MSPF of a connection

or NAND gate, $MSPF^{(d)}(c_{ij})$ is given by Equations (2.2), (2.3) and (2.4) respectively. The function G in Equations (2.3) and (2.4) is defined as

$$G = \cap_{1 \le x \le k, x \ne i} f(c_{xj})$$

where $\cap$ is the Boolean AND operator and the operators $\#_{OR}$, $\#_{NAND}$ and $\#_{AND}$ are given in Tables B, C and D in Figure 2.3, respectively. The "-" sign in these tables means that those situations will never be encountered.

$$MSPF^{(d)}(c_{ij}) \;=\; F^{(d)} \#_{OR} MSPF^{(d)}(v_j) \tag{2.2}$$

$$MSPF^{(d)}(c_{ij}) \;=\; G^{(d)} \#_{AND} MSPF^{(d)}(v_j) \tag{2.3}$$

$$MSPF^{(d)}(c_{ij}) \;=\; G^{(d)} \#_{NAND} MSPF^{(d)}(v_j) \tag{2.4}$$

A special case arises when v_j is a NOT gate; $MSPF^{(d)}(c_{ij})$ is then computed simply by Equation (2.5), where "˜" is the COMPLEMENT operator.

$$MSPF^{(d)}(c_{ij}) \;=\; \tilde{\ } MSPF^{(d)}(v_j) \tag{2.5}$$

The evaluation of the MSPF of a gate is slightly more complicated. Consider a gate v_i as shown in Figure 2.4 with c_{i1}, c_{i2}. …, c_{ik} as its output connections. If the gate v_i is not a reconvergent gate, its MSPF is given by Equation (2.6), where $* \cap 1 = 1 \cap * = 1$ and $* \cap 0 = 0 \cap * = 0$ in addition to the normal properties of $\cap$ on the domain $\{0,1\}$.

Table A

$\#_{NOR}$		$MSPF^{(d)}$ 0	1	*
$F^{(d)}$	0	1	0	*
	1	*	-	*

Table B

$\#_{OR}$		$MSPF^{(d)}$ 0	1	*
$F^{(d)}$	0	0	1	*
	1	-	*	*

Table C

$\#_{NAND}$		$MSPF^{(d)}$ 0	1	*
$G^{(d)}$	0	-	*	*
	1	1	0	*

Table D

$\#_{AND}$		$MSPF^{(d)}$ 0	1	*
$G^{(d)}$	0	*	-	*
	1	0	1	*

Figure 2.3 The tables for computing MSPF for simple gates

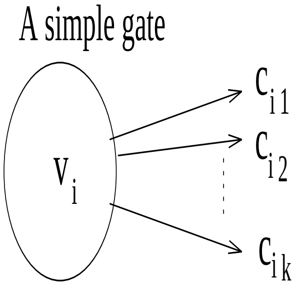

Figure 2.4 Computation of the MSPF of a gate

$$MSPF^{(d)}(v_i) \;=\; \cap_{1 \leq x \leq k} MSPF^{(d)}(c_{ix}) \tag{2.6}$$

However, if v_i is reconvergent, then the procedure of computation of the MSPF becomes more complex. The gate v_i is then treated as an input variable, and all of the primary outputs are evaluated with respect to v_i. Using Shannon's expansion, the function at every output z_j can then be expressed as

$$f(z_j|v_i) \;=\; f(v_i) \cap P_j \cup \tilde{}f(v_i) \cap Q_j$$

where P_j and Q_j are some functions expressed in terms of primary inputs only. The MSPF of v_i due to z_j, $MSPF_j^{(d)}(v_i)$, can then be recomputed using the following algorithm:

if $f(v_i)^{(d)} = 1$ and $Q_j^{(d)} = 0$, then $MSPF_j^{(d)}(v_i) = 1$
else if $f(v_i)^{(d)} = 0$ and $Q_j^{(d)} = 1$, then $MSPF_j^{(d)}(v_i) = 1$
else if $f(v_i)^{(d)} = 1$ and $P_j^{(d)} = 0$, then $MSPF_j^{(d)}(v_i) = 0$
else if $f(v_i)^{(d)} = 0$ and $P_j^{(d)} = 1$, then $MSPF_j^{(d)}(v_i) = 0$
else $MSPF_j^{(d)}(v_i) = *$.

The final value for $MSPF^{(d)}(v_i)$ is then the intersection of $MSPF_j^{(d)}(v_i)$ for $1 \leq$ j $\leq$ m.

To explain the correctness of this algorithm, suppose $f^{(d)}(v_i) = 1$ and $Q_j^{(d)} = 0$. Hence, $f^{(d)}(z_j|v_i) = P_j^{(d)}$, which could be either either 1 or 0. Therefore, $MSPF_j^{(d)}(v_i)$ must be 1 so as to allow $P_j^{(d)}$ to propagate to $f^{(d)}(z_j|v_i)$. Similar arguments can be presented for the other three cases.

The MSPFs of all of the gates and connections in a circuit can be computed using the information presented in this subsection by first setting the MSPF of each primary output to be the same as the output function of the gate driving that primary output, and then computing the rest of the MSPFs from the primary outputs towards primary inputs.

2.2.3 Compatible set of permissible functions

As the MSPF of a gate contains the largest set of permissible functions associated with it, this set also contains the largest observability don't-care set for the gate. The observability don't care set of a gate is the set of input values for which the gate's output is not observable through the primary outputs. However, as explained in the last subsection, the computation of MSPF could be time-consuming, especially when a circuit has many reconvergent gates. In addition, the MSPFs of all of the gates and connections in a circuit have to be recomputed each time the circuit is transformed and reduced. Therefore, to reduce the amount of processing time required, compatible sets of permissible functions (CSPF) for gates and connections are more frequently used.

A *compatible set of permissible functions* (CSPF) is a subset of the MSPF and is computed based on some ordering of the connections in a circuit. Although the don't-care set associated with a CSPF is often smaller than that with the MSPF, the quality of the resulting circuit minimized using a CSPF usually does not suffer too badly and the processing time is drastically reduced.

The computation of the CSPFs is similar to the computation of the MSPFs. Referring again to Figure 2.2, the CSPF for the connection c_{ij} is given by Equations (2.7), (2.8), (2.9) or (2.10) depending on whether v_j is a NOR, OR, AND or NAND gate, respectively.

$$CSPF^{(d)}(c_{ij}) = F'^{(d)} \#_{NOR} CSPF^{(d)}(v_j) \tag{2.7}$$

$$CSPF^{(d)}(c_{ij}) = F'^{(d)} \#_{OR} CSPF^{(d)}(v_j) \tag{2.8}$$

$$CSPF^{(d)}(c_{ij}) = G'^{(d)} \#_{AND} CSPF^{(d)}(v_j) \tag{2.9}$$

$$CSPF^{(d)}(c_{ij}) = G'^{(d)} \#_{NAND} CSPF^{(d)}(v_j) \tag{2.10}$$

The functions F' and G' are slightly different from F and G described in the previous subsection and are given by

$$F' = \cup_{x<i} f(c_{xj})$$

$$G' = \cap_{x<i} f(c_{xj})$$

As can be seen, F' and G' depend on how the connections are ordered. If a connection is placed towards the beginning of the ordered list (i.e., with a smaller "i" value), the size of its don't-care set associated with its CSPF is smaller. Some heuristics are proposed in [35] to order the connections in a circuit.

The computation of the CSPF of a gate is performed exactly as the case of computing the MSPF of a nonreconvergent gate. Formally,

$$CSPF^{(d)}(v_i) = \cap_{1 \le x \le k} CSPF^{(d)}(c_{ix}) \tag{2.11}$$

The CSPF of an output gate is the same as its output function. The CSPFs are computed from the primary outputs to the primary inputs as is done for the case of MSPFs.

Since the CSPF of a reconvergent gate can also be evaluated using Equation (2.11) as opposed to computing the output functions of all of the primary outputs (as done for the case of MSPF), the use of CSPF can reduce the computation time. In addition, as

CSPFs are based on a partial ordering of the connections, they need not be recomputed each time the circuit is transformed.

2.2.4 Pruning

The pruning procedure removes the redundant connections in a circuit. Pruning can be performed based on either MSPF or CSPF. To detect the redundant connections, the MSPFs or the CSPFs of all of the connections have to be computed. A connection c_{ij} can be declared redundant if one of the following conditions is true:

(1) If the gate v_j is a NOR or OR gate and $SPF^{(d)}(c_{ij}) = 0$ or * for all $1 \leq$ d $\leq 2^n$

(2) If the gate v_j is a NAND or AND gate and $SPF^{(d)}(c_{ij}) = 1$ or * for all $1 \leq$ d $\leq 2^n$

(3) If the gate v_j is a NOT gate and $SPF^{(d)}(c_{ij}) = *$ for all $1 \leq$ d $\leq 2^n$

To understand the correctness of the redundancy conditions, let us consider a connection c_{ij} connected to an AND gate v_j. If $SPF^{(d)}(c_{ij}) = 1$ or * for all $1 \leq$ d $\leq 2^n$, then c_{ij} can be set to logical 1 without changing the output function of the primary outputs. A logical 1 is a redundant connection at the input of an AND gate. Hence, c_{ij} is redundant and can be removed. The other conditions for redundancy can be explained similarly.

If CSPF is used for detecting redundant connections, then all of the identified redundant connections can be removed without recomputing the CSPFs. But if MSPF is used, the MSPFs have to be recomputed after one redundant connection is removed.

2.2.5 Gate substitution

In a gate substitution procedure, a gate v_i in a circuit is selected and the other existing gates are compared with the gate v_i to determine if any one of them can replace v_i without changing the output function of the primary outputs. This is illustrated in Figure 2.5.

To determine if a gate v_j can replace another gate v_i, either the MSPF or the CSPF of v_i can be used. However, the use of MSPF is very time-consuming. Hence, CSPF is used

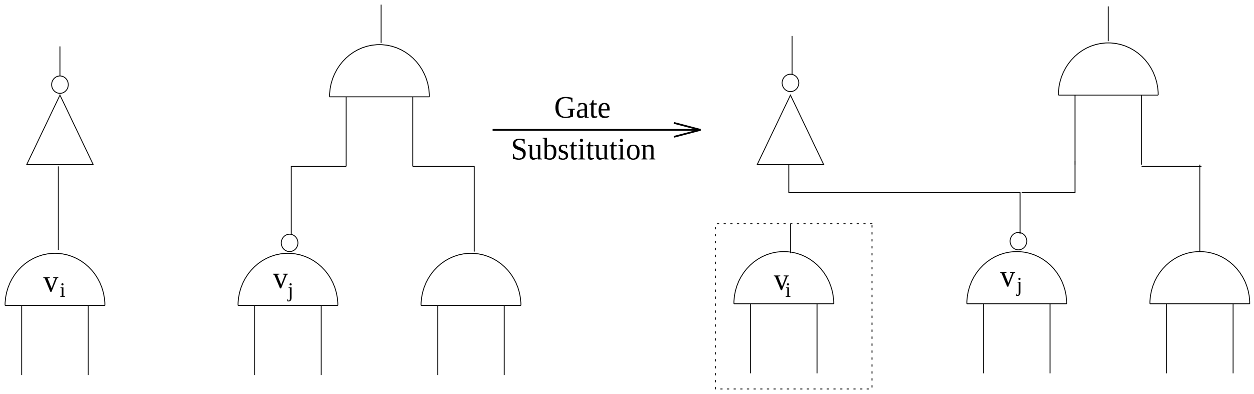

Figure 2.5 An example of gate substitution

for gate substitution and all other transformations. The condition for v_j to be able to replace v_i is $f^{(d)}(v_j) \in CSPF^{(d)}(v_i)$ for all $1 \leq \mathrm{d} \leq 2^n$. The correctness of this condition follows straight from the definition of CSPF. If $f(v_j)$ is an element of $CSPF(v_i)$, the output functions at the output connections of v_i can be changed to the output function of v_j without changing the primary outputs. Hence, each of the output connections of v_i can be connected to the output of v_j instead of v_i, and v_i can be removed from the network.

2.2.6 Gate merging

The gate merging procedure is slightly more complicated that the gate substitution procedure described in the last subsection. The basic idea of this procedure is to select two gates, v_i and v_j, and determine if a third gate v_k can be synthesized with its inputs connected to a subset of the existing gates in the circuit such that v_k can replace both the gates v_i and v_j. This results in removal of a gate from the circuit and is shown in Figure 2.6.

Two gates, v_i and v_j, are *possible to merge* if the intersection of their CSPFs is not NULL. Then one must synthesize a new gate, v_k, whose output function is a subset of the intersection of the CSPFs of v_i and v_j. The condition of gate merging is formally defined as follows.

$$output\_function(v_k) \subseteq CSPF(v_i) \cap CSPF(v_j)$$

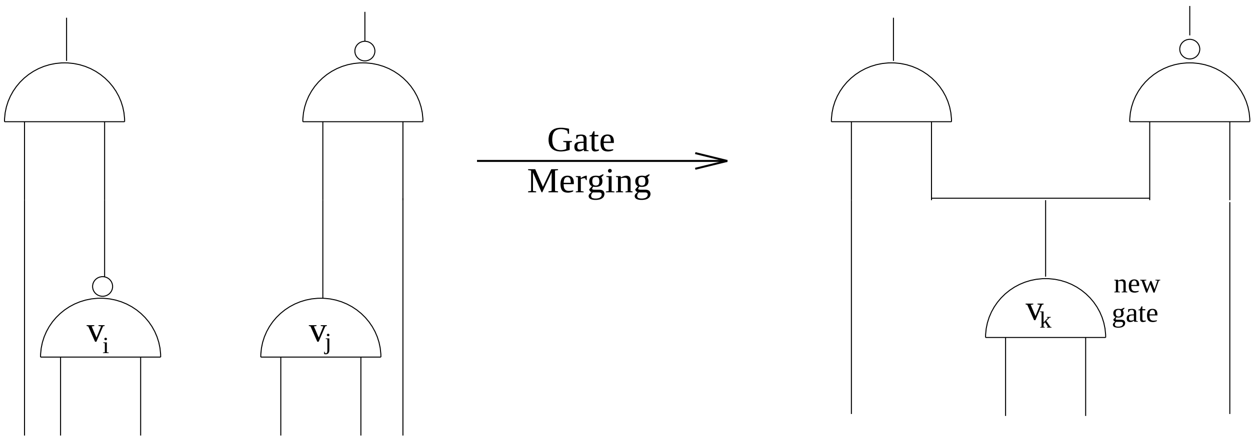

Figure 2.6 An example of gate merging

The new gate v_k is formed by connecting some existing gates in the circuit to the input of v_k. A gate v_p is said to be *connectable* to another gate v_q if the following conditions apply:

(1) If v_q is a NOR gate and there does not exist a value for d between 1 and 2^n such that $CSPF^{(d)}(v_q) = 1$ and $f^{(d)}(v_p) = 1$.

(2) If v_q is a OR gate and there does not exist a value for d between 1 and 2^n such that $CSPF^{(d)}(v_q) = 0$ and $f^{(d)}(v_p) = 1$.

(3) If v_q is a AND gate and there does not exist a value for d between 1 and 2^n such that $CSPF^{(d)}(v_q) = 1$ and $f^{(d)}(v_p) = 0$.

(4) If v_q is a NAND gate and there does not exist a value for d between 1 and 2^n such that $CSPF^{(d)}(v_q) = 0$ and $f^{(d)}(v_p) = 0$.

2.2.7 Generalized gate substitution

As the name suggests, generalized gate substitution is a more general form of gate substitution. In gate substitution, a gate is checked to see if all of its output connections can be replaced by the output of another gate in the circuit. In generalized gate substitution, each of the gate's output connections is checked if it can be replaced by the output of some other gate instead. Hence, in generalized gate substitution, a gate is substituted by more than one gate.

2.2.8 Gate input reduction

The last transformation available in the Transduction method is gate input reduction. In this transformation, a new gate v_j is synthesized to replace a target gate v_i such that the number of inputs of v_j is less than that of v_i. After a successful gate input reduction transformation, the total number of connections in the circuit is reduced.

To perform this transformation, a more stringent form of the connectability condition, namely, the effectively connectable condition, is needed. A gate v_j is said to be *effectively connectable* to v_i if one of the following four conditions is true in addition to the connectability condition.

- If v_i is a NOR gate, there must be some value of d between 1 and 2^n inclusive such that $CSPF^{(d)}(v_i) = 0$ and $f^{(d)}(v_j) = 1$.
- If v_i is an OR gate, there must be some value of d between 1 and 2^n inclusive such that $CSPF^{(d)}(v_i) = 1$ and $f^{(d)}(v_j) = 1$.
- If v_i is an AND gate, there must be some value of d between 1 and 2^n inclusive such that $CSPF^{(d)}(v_i) = 0$ and $f^{(d)}(v_j) = 0$.
- If v_i is a NAND gate, there must be some value of d between 1 and 2^n inclusive such that $CSPF^{(d)}(v_i) = 1$ and $f^{(d)}(v_j) = 0$.

2.3 Parallel Algorithms for Transduction Method

An obvious way of extracting significant speedup out of the logic synthesis application is to generate a large number of partitions and to synthesize each partition independently. The results of the individual partitions can then be merged back. Unfortunately, such an approach has the problem that with an increasing number of partitions, the quality of the overall network degrades. Hence, we do not use that approach in the ProperSYN algorithm. However, really large circuits can not be synthesized as whole due to the excessive run times and/or memory requirements. For those circuits, the partitioning approach of parallel synthesis is very suitable. This approach is discussed in detail in Chapter 4 of this thesis.

2.3.1 Work decomposition using partitioning

In our ProperSYN algorithms, the logic network is divided into a set of nonoverlapping partitions. However, it should be noted that the partitioning is for *parallelization purposes only*; we *do not perform synthesis on those partitions separately*. The logic minimization is performed on the entire network as a whole.

We used a very simple partitioning strategy. The user specifies the maximum number of gates per partition, *max_gates*. Initially, all of the primary outputs of the given circuit are put in a list called *cone_tips*. Then, a node η is picked from the list *cone_tips*, and the transitive fan-ins of η which do not yet have their partition assigned are put into a new partition Π in a breadth-first manner until the number of nodes in Π reaches *max_gates*, after which a new partition Π' is formed. Then all of the fan-in nodes to nodes in Π are put at the end of *cone_tips*. This procedure continues until all of the nodes in the network are assigned to some partition. The algorithm is shown in Figure 2.7.

For each partition, one *object* is created. Processes that perform various operations on these objects are distributed to different processors using some load-balancing methods supported by the CHARM run-time system, as depicted in Figure 2.8. Typically, 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 the communication time for sending a message between objects. We will denote the object for a partition p as *partition_object* p. Each processor has a copy of the network, but has the responsibility for the shaded regions of the network. Effectively, we can say that each processor "owns" some parts of the network.

Algorithm: Network Partition (network, max_gates)
/* Algorithm for partitioning the network */

```
cone_tips = list_of_partitions = ∅
foreach primary output (network, PO)
    insert PO in cone_tips.
While (some nodes are not assigned a partition yet) {
    η = Extract head of cone_tips.
    While(η is assigned partition)
       η = Extract head of cone_tips.
    Π = temp_list = ∅.
    Put η in temp_list.
    While((||Π|| < max_gates) && (temp_list ≠ ∅)) {
       μ = Extract head of temp_list.
       Put μ in Π.
       For each fan-in(μ, ν)
          If(ν is not assigned partition)
             Put ν at the end of temp_list.
    }
    Add Π to the list_of_partitions.
    For each item(temp_list, μ)
       Put μ at the end of cone_tips.
}
```

Figure 2.7 The partitioning algorithm

The quality of the resultant circuit does not vary with the partitioning strategy beyond the statistical variation shown in Section 2.4.2 because the partitioning is used only for the purpose of division of work. The evaluation time for the output function and the CSPF depends on the partitioning strategy. But determining a good partitioning strategy for the minimization of the evaluation times for the output function and the CSPF is a nontrivial procedure and can increase the overall run time of any synthesis algorithm. The evaluation times for the output functions and the CSPFs are a small part of the total run time of the algorithm. Hence, we decided to use a simple partitioning strategy.

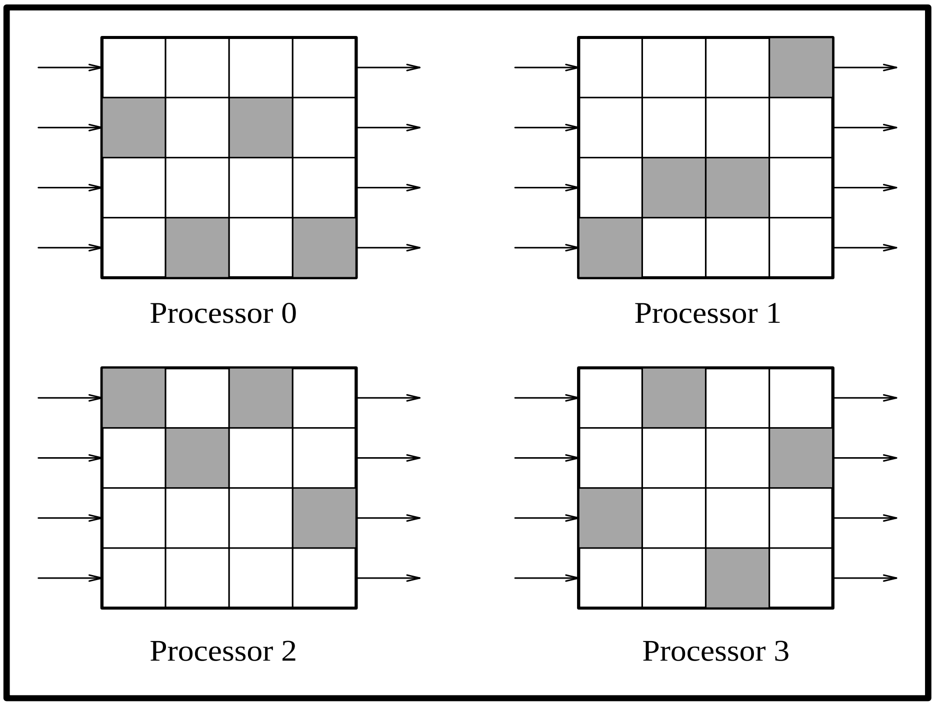

Figure 2.8 Partitions of a network and the "ownership" of each processor

2.3.2 Parallel evaluation of output function

For optimization purposes, we have to evaluate the output functions at each gate and connection in the network. The output functions have to be expressed in terms of the primary inputs of the network. We compute output functions using binary decision diagrams (BDD) [36].

A major problem we have to deal with is the fact that since different processes perform different optimizations simultaneously on the network, the output functions and the CSPFs keep changing. It is important to provide some coherence mechanism among various applications of these optimizations in parallel. To do that with each BDD, we keep a tag, called the *version*. Also, with each gate and connection, we keep the current version number of the output function and the CSPF. A BDD is "valid" if its version number is current. Any transformation in the network changes the output functions and the CSPFs in some parts of the network. The version numbers are used to prevent any "illegal" transformation done using an "invalid" BDD. Initially, the versions of output functions and the CSPF are set to 1. If, due to any transformation, the output function or CSPF of a gate or connection becomes "invalid," the corresponding version number attached to it is incremented by 1.

The evaluation of output functions starts from the primary inputs of the network. The output function of a gate can be evaluated if "valid" output functions of its fan-ins are available. Any gate whose fan-ins are primary inputs only can be evaluated without any constraints. The gates which have fan-ins from other partitions can not be evaluated unless the output function of those fan-ins arrives from the other partitions.

After the output function of a gate ζ is evaluated in a partition, it is checked if its output function is needed in any other partition. If the other partition that needs the output function is in the same processor, that *partition_object* is informed about the availability of the output function of ζ. But if it is in a different processor, then the current *partition_object* invokes the *work manager* in the current processor, which in turn transmits the BDD to the *work manager* of the destination processor. Then the *work manager* in the destination processor sends a message to the corresponding processes in that processor about the availability of the output function of ζ. We use the *work manager* to ensure that only one copy of the BDD is sent across the processor boundary. The procedure is illustrated in Figure 2.9.

Whenever a *partition_object* receives a message about the availability of the output function of a gate ζ, it checks to make sure that it is a "valid" BDD. It then checks for the gates in this partition which are waiting for the availability of the output function of ζ. Those gates which have all of the output functions of its fan-ins available can now be evaluated.

It is clear from the above description that the parallel evaluation of the output function is performed using an asynchronous message-driven data-flow computational model. We do not have any synchronizing barrier for levels of gates as is done in parallel logic simulation algorithms, and hence our parallel algorithm is expected to scale well with a large number of processors.

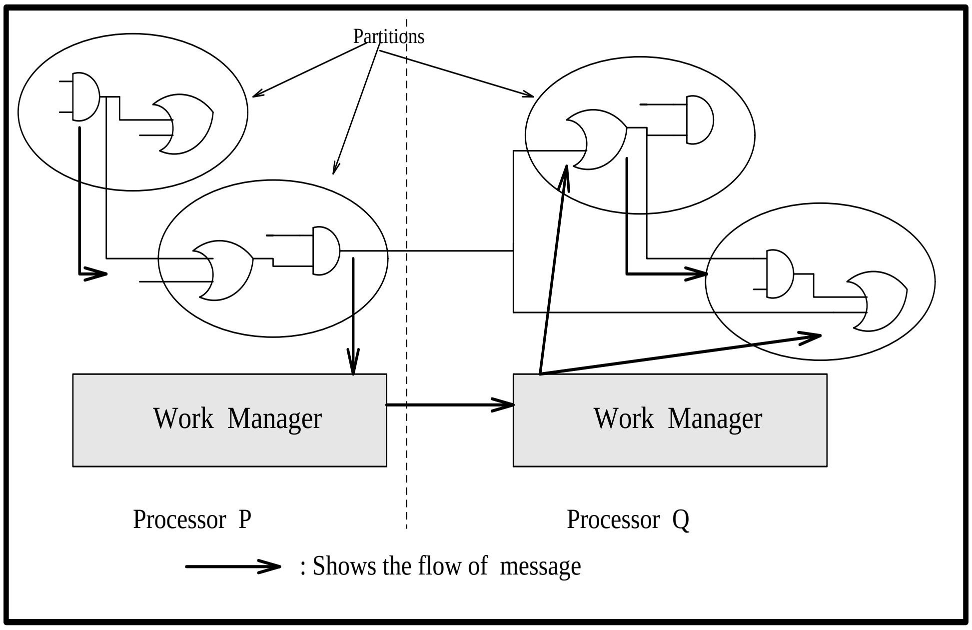

Figure 2.9 Evaluation of output functions and the flow of messages

2.3.3 Parallel evaluation of CSPF

The flow of evaluation of CSPF moves from the primary outputs to the primary inputs. If the output function of a primary output is available, its CSPF can be calculated immediately. If no *external don't care* [37] is specified for that primary output, then the CSPF is the same as the output function; otherwise, the CSPF is the union of the output function and the external don't care.

The CSPF of an input connection to a gate can be computed if the output functions of all of the input connections to that gate are available as well as the CSPF of that gate. Then the CSPF of the connection is computed as a function of the type of gate, the CSPF of the gate, the output functions of the sibling connections and the output function of the connection itself [17]. The CSPF of a gate can be computed if the CSPFs of all of the output connections are available. The CSPF of a gate is the set intersection of the CSPFs of its output connections [31].

The partitions which have any primary outputs start evaluation of the CSPF at the primary outputs. After evaluation of an input connection, it is checked if it is connected to the output of a gate which is in a different partition and the partition is in the same

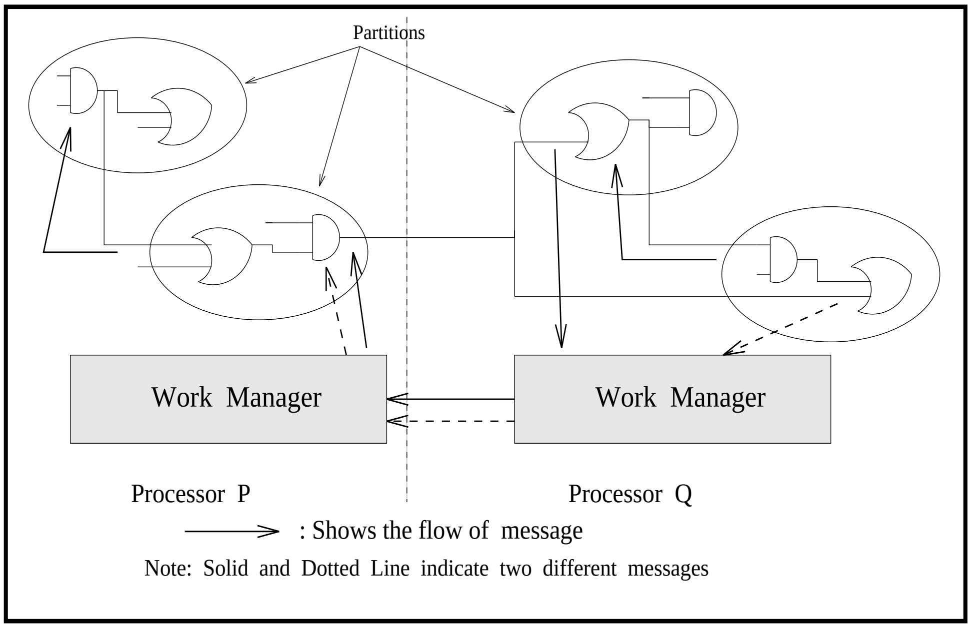

Figure 2.10 Evaluation of CSPF and the flow of messages

processor; then that *partition_object* is informed about the availability of the CSPF of the connection. If the partition is in a different processor, the CSPF is transmitted to the other processor through the *work managers* of the two processors, similar to the case of the evaluation of output functions. The procedure is explained in Figure 2.10.

Again, it is clear from the description of the parallel algorithm that the parallel evaluation of the CSPF proceeds in an asynchronous message-driven data-flow computational model with no barriers.

2.3.4 Parallel gate substitution

Gate substitution is one of the transformations provided in the Transduction method. The main idea of the gate substitution is to search a given network for pairs of gates such that in each pair, one of the gates can substitute for the other. However, as the network has to be loop-free, the substitute gate must not be a successor of the gate to be substituted. Hence, to minimize the chance of this happening, the substitute gate and the substituted one should be as near as possible to the primary inputs and outputs, respectively.

Let us denote p to be the number of *partition_objects* created. Then the *partition_object* x will create another p objects, called *gate_sub_objects*. They are denoted as the pair (x, y), x = 1, ..., p, y = 1, ..., p. The *gate_sub_object* denoted by (i, j) is responsible for substitution of any gate in the *partition_object* i by any gate in the *partition_object* j. Note that another *gate_sub_object* (j, i) will be created whose job is just the opposite of the job of (i, j). Hence, the number of *gate_sub_objects* created will be p^2. We have already mentioned that it is preferable to have the partition i close to the primary outputs and the partition j close to primary inputs to maximize the chance of finding a gate substitution pair. Thus, we adjust the priorities of the different gate_sub_object to achieve that (the priorities are used to direct the CHARM run-time system to process certain messages with higher priority). As shown in the partitioning algorithm given in Figure 2.7, the partitions closer to the primary outputs are formed earlier than the partitions closer to the primary inputs. Hence, the number designating a partition closer to the primary outputs will be smaller than the number designating a partition closer to the primary inputs. Thus, a simple method of assigning priority to gate_sub_object (i, j) will be to assign higher priorities to the objects with smaller i and larger j. The equation we used to assign the priority to a gate_sub_object (i, j) is given by

$$priority(gate\_sub\_object(i, j)) \ = \ 100000 + 1000 * i - j$$

The procedure is depicted in Figure 2.11.

Now, consider the *gate_sub_object* (i, j). A gate g_1 in the partition i can be substituted by a gate g_2 in the partition j if the output function of the gate g_2 is a subset of the CSPF of the gate g_1. Thus, the *gate_sub_object* (i, j) will need "valid" CSPFs of all the gates in the partition i and output functions of the gates in the partition j.

When the *gate_sub_object* denoted by (i, j) is awakened by a message, it starts by collecting the output functions and the CSPFs that it needs. Some output functions and CSPFs may not be available that time. Hence, it asks the *work manager* in this processor to obtain the BDD and inform it when that is available. If this processor does not "own" that portion of the network which contains the gate for which BDD has been

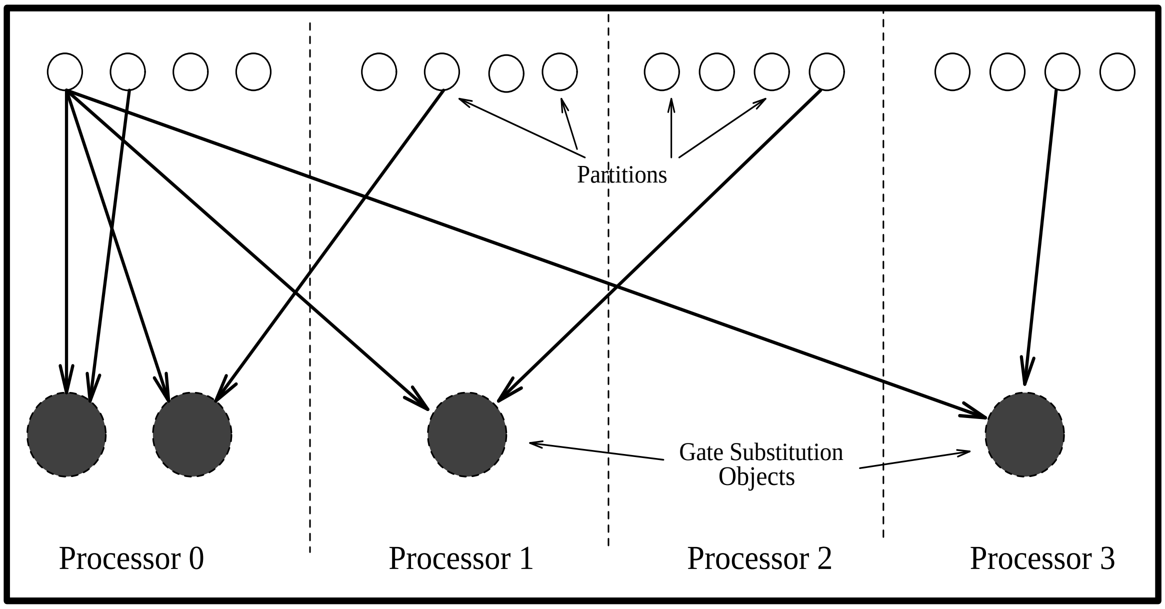

Figure 2.11 Creation of objects for gate substitution

requested, the *work manager* on the requesting processor will ask the *work manager* in the requested processor to send the BDD. After this *gate_sub_object* has collected all of the available "valid" BDDs, it does pairwise comparisons to discover any possible gate substitution. It performs the comparison only if that gate substitution will not create a cycle in the circuit. That check is performed by marking the fan-out cone of the gate to be substituted and checking if the substitute gate is not marked. If any gate substitution is possible, then it sends a message to a particular processor, designated to be the *master* processor for deciding about any change in the network, asking permission for this gate substitution. The message also contains the version numbers of the output function and the CSPF used for checking the gate substitution. When it is finished with all comparisons, it asks *partition_object* i to inform it if CSPF of any gate in the partition i changes, and it asks *partition_object* j to inform it if the output function of any gate in the partition j changes.

Whenever an output function or CSPF arrives from other processor, or the current version is computed in this processor, the *work manager* checks the list to find out if any object has asked for this BDD. If so, it sends a message to the corresponding objects.

```
Algorithm: Parallel Gate Substitution (i, j)
/* This is for the gate_sub_object (i, j) */

(1)  /* When object is awakened */
     1.1 Collect CSPFs of all of the gates in partition i
     1.2 Collect Output Functions of all of the gates in partition j
     1.3 If any CSPF or Output Function is not available, request
         work manager to get it
     1.4 Perform pairwise comparison of available CSPFs and output functions
     1.5 If possible gate substitution found, ask master processor for permission
     1.6 Ask partition i to inform if any CSPF changes in partition i
     1.7 Ask partition j to inform if any output function changes in partition j

(2)  /* When this object receives a new CSPF */
     2.1 Compare this CSPF with all available output functions of partition j
     2.2 If possible gate substitution found, ask master processor for permission

(3)  /* When this object receives a new output function */
     3.1 Compare this output function with all available CSPFs of partition i
     3.2 If possible gate substitution found, ask master processor for permission
```

Figure 2.12 Algorithm for Parallel Gate Substitution

Whenever a *gate_sub_object* (i, j) receives a message about the availability of an output function of a gate g_2 in the partition j, it then compares that output function with the "valid" CSPFs available for the gates in the partition i. If any substitution is possible, it then sends a message asking for permission for that gate substitution, as explained earlier. Similar operations are performed if this object receives a message about the availability of the CSPF of a gate in partition i. The algorithm is given in Figure 2.12.

We want to emphasize the fact that in our parallel algorithm, we check for all possible gate substitutions, as done in the sequential algorithm. We explain that fact with some examples. Consider the circuit shown in Figure 2.13(a) with 4 gates (A, B, C and D). If we perform all possible comparisons as shown using the directed edges, there are 12 comparisons in total. Let us assume that we partition the circuit into two partitions, where partition 1 contains A and B and partition 2 contains C and D. Then, our

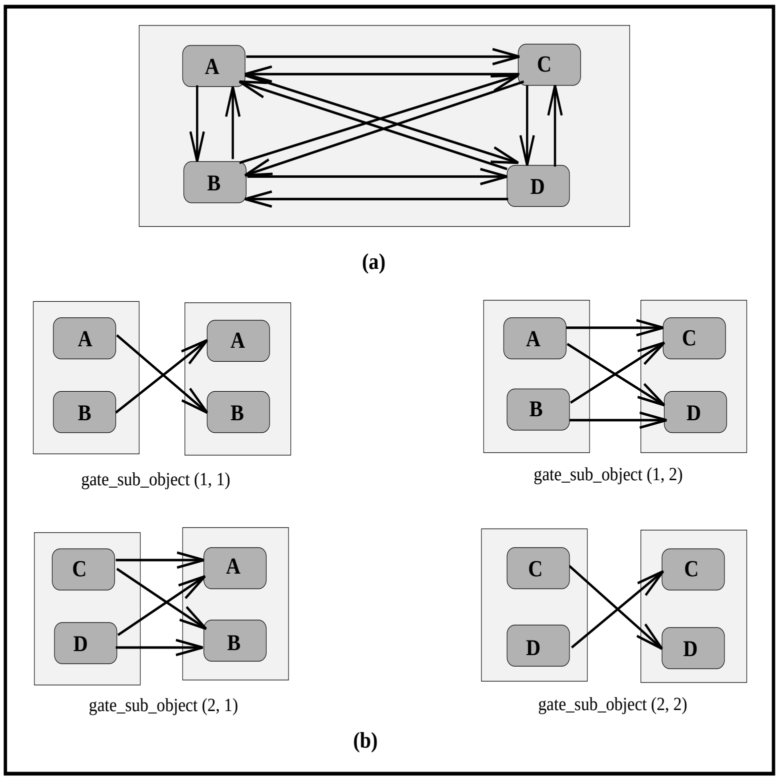

Figure 2.13 Comparisons performed by (a) the sequential algorithm and (b) our parallel algorithm for gate substitution. It shows that the number of comparisons is the same for both algorithms.

parallel algorithm will generate 4 *gate_sub_objects*. In Figure 2.13(b), we have shown the comparisons performed by different *gate_sub_objects*. One can observe that the total number of comparisons performed by them is 12, which is the same as the number of comparisons performed by the sequential algorithm.

When the *master* processor receives the request for permission for a gate substitution, it first checks if both of the gates in the gate substitution pair exist. Then it checks if the current version of the output function and the CSPF of the substitute gate and the substituted gate, respectively, are the same as that mentioned in the message. The version check is performed to make sure that the requested gate substitution is still "legal," i.e., no other transformation performed on the network has affected this gate substitution. The *master* processor does not check for the acyclic property of the circuit. That check was already performed in the gate substitution object. If any change was made to the network after that check, that would be reflected in the change in version number of the BDDs. That would make the requested transformation "illegal." The *master* processor then broadcasts a message to all other processors to perform this gate substitution in their copy of the network and then itself performs the substitution. It is through this centralized update that coherence is maintained in data updates and it is guaranteed that no errors are made in the optimization process.

After the substitution is performed, the output functions of all of the *successors* of the substituted gate will become invalid as well as the CSPFs of the *predecessors* of the substitute gate. This is shown in Figure 2.14(a). But if the output functions are reevaluated in the region where they are marked, then the CSPFs of all of the gates in the transitive fan-ins of that region will be invalidated. Since, in our approach, we do not have any synchronizing barrier to decide when to start reevaluation of the output functions and the CSPFs, the reevaluation can start any time in any order. Thus, we mark the invalidation of output function and the CSPFs as shown in the Figure 2.14(b). Invalidation of the output function or the CSPF is done by incrementing the current version number attached to the corresponding gates and connections such that any BDD in the system with lower version number will be considered "invalid."

After the invalidation of the output functions and the CSPFs, messages are sent to the corresponding *partition_objects* to reevaluate the BDDs. After they have reevaluated the BDDs, they will inform the corresponding *gate_sub_objects* to check for more gate substitutions as the BDDs have changed.

2.3.5 Parallel gate merging

Similar to gate substitution, gate merging is an iterative improvement procedure. In this transformation, we search for a pair of gates, (g_1, g_2) and try to synthesize a third gate g_3 which can replace both gates. Two gates, g_1 and g_2, are possible to merge if the intersection of the CSPFs of the two gates is nonempty. Then we try to form a third gate with four simple gate types: AND, OR, NAND and NOR. For each gate type, we try to connect the inputs of the gate from the minimal set of outputs of the existing gates in

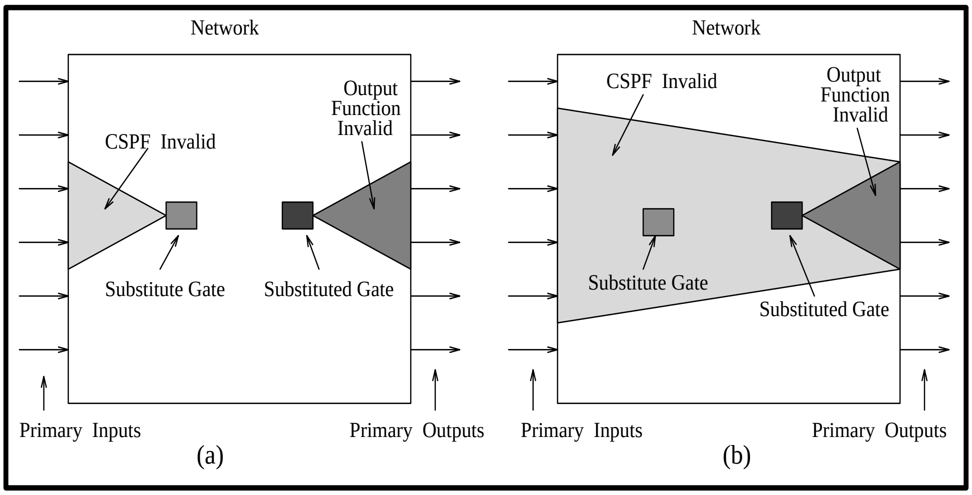

Figure 2.14 Invalidation of output function and CSPF due to gate substitution

the network. If we succeed in forming such a gate, we can replace both of the gates, g_1 and g_2, by the new gate. However, in order to maintain the loop-free condition of the network, the inputs of the new gate, g_3, must not come from the successors of g_1 and g_2. Hence, to maximize the chance of gate merging, we prefer the gates g_1 and g_2 to be close to the primary outputs.

Let us denote p as the number of partitions. For each partition pair, (i, j), an object is created of type *gate_merge_object*. The *gate_merge_object* denoted by (i, j) is responsible for testing for any gate merging between any gate in the partition i and any gate in the partition j. In addition to that, a number (equal to p) of *gate_merge_objects* will be created, one for each partition, to check for any possible gate merging in the same partition. Thus, in total $\binom{p}{2} + p$ objects are created. The procedure is depicted in Figure 2.15.

Now consider a *gate_merge_object* denoted by (i, j). The gate g_1 in the partition i can be merged with the gate g_2 in the partition j if the intersection of the CSPFs of these two gates is nonempty. Hence, this gate_merge_object will need "valid" CSPFs of all the gates in the partition i and the partition j.

A *gate_merge_object* behaves almost in the same manner as a *gate_sub_object* if it does not find any CSPF. It requests the *work manager* in the current processor, which sends a

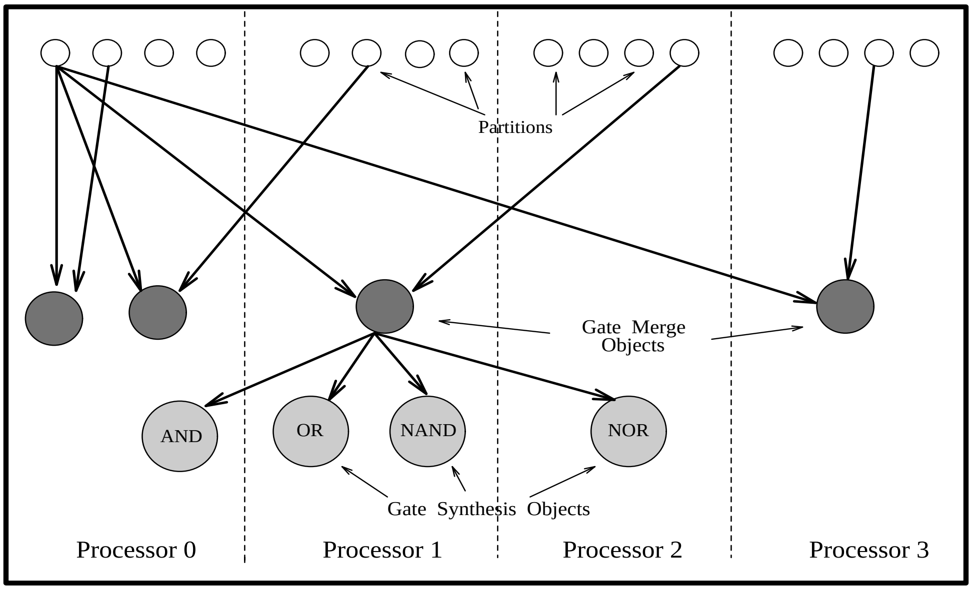

Figure 2.15 Creation of gate merge objects and creation of objects for gate synthesis

message to the *work manager* of the corresponding processor, if needed. Whenever that CSPF is available, the *work manager* will inform this object. If this *gate_merge_object* is finished with searching for gate merging pairs with available CSPFs, it requests the corresponding *partition_objects* to inform this object whenever the CSPF in those partitions change. We are omitting the details to avoid the repetition of the description of the almost similar procedure. The algorithm is described in Figure 2.16.

If any *gate_merge_object* finds a pair of gates, (g_1, g_2), which have nonempty intersection of their CSPFs, then it creates 4 new objects, called *gate_synthesis_objects*, for each of four types of simple gate, AND, OR, NAND and NOR. A *gate_synthesis_object* has the responsibility of synthesizing a gate of the type supplied to it whose output function is a member of the intersection of the CSPFs of the pair of gates which we want to merge. The input of the synthesized gate will be connected to a minimal set of outputs of the existing gates in the network; the successors of the two gates to be merged are forbidden to connect.

Whenever a *gate_synthesis_object* is awakened, it first checks if both of the gates exist and if they have the same CSPF version number as they had when they were checked for merging. Then it makes a list of all of the gates which are not successors of both of

Algorithm: Parallel Gate Merging (i, j)
/* This is for the gate_merge_object (i, j) */

(1) /* When object is awakened */
```
1.1 Collect CSPFs of all of the gates in partition i
1.2 Collect CSPFs of all of the gates in partition j
1.3 If any CSPF is not available, request work manager to obtain it
1.4 Perform pairwise comparison of available CSPFs to find
    any possible gate merging.
1.5 If possible gate merging found between gate g1 and g2, create 4 new gate
    synthesis objects, one for each of the gate types; AND, OR, NAND and NOR
1.6 Ask partition i to inform if any CSPF changes in partition i
1.7 Ask partition j to inform if any CSPF changes in partition j
```

(2) /* When this object receives a new CSPF of partition i */
```
2.1 Compare this CSPF with all available CSPFs of partition j
2.2 If possible gate merging found between gate g1 and g2, create 4 new gate
    synthesis objects, one for each of the gate types; AND, OR, NAND and NOR
```

(3) /* When this object receives a new CSPF of partition j */
```
3.1 Compare this CSPF with all available CSPFs of partition i
3.2 If possible gate merging found between gate g1 and g2, create 4 new gate
    synthesis objects, one for each of the gate types; AND, OR, NAND and NOR
```

Figure 2.16 Algorithm for Parallel Gate Merging

the gates which are to be merged. Let us denote this list as the *possible_input_list*. For every gate ζ in the *possible_input_list*, the *gate_synthesis_object* tries to connect that gate to the input of the synthesized gate if a "valid" output function ζ is available. If it is possible to connect, it first puts ζ in a *partial_input_list* and then checks if the new gate synthesis is complete, i.e., the output function of the new gate is already a member of the intersection of the CSPFs of the gate pairs to be merged. If it has already been realized, it first minimizes the number of input connections to the new gate and then asks the *master* processor for permission to perform this gate merging. If the synthesis process is not complete, it remembers the partial output function that has been produced thus far. If a "valid" output function of any gate in *possible_input_list* is not available, it requests the *work manager* in the current processor to inform itself whenever it is available. After going through *possible_input_list*, if it can not synthesize the gate and it has no new output function to receive from the *work manager*, it terminates itself.

When a *gate_synthesis_object* receives an output function, it again checks if both of the gates which are to be merged exist and that their CSPFs have not changed since they were checked for possible merging. It then checks if the output function received can be connected to the synthesized gate. If it is possible to connect to the synthesized gate, it checks for all of the gates in the *partial_input_list* to ensure that the version number of their output functions has not changed. If there is no change, it simply adds this new connection and updates the partial output function and *partial_input_list.* If any change has happened, it has to discard the partial output function and perform the computations again. The algorithm is described in Figure 2.17.

When the *master* processor receives a message, asking permission for a gate merging, it checks for the existence of the gates to be merged and the correct versions of CSPFs. It then checks for the existence of the immediate predecessors of the synthesized gate the correct version of their output functions. If everything is satisfied, it broadcasts a message to all other processors to perform this gate merging and performs the gate merging itself. Due to a gate merge, the invalidation of output functions and CSPF will occur similarly to the case of gate substitution described earlier.

2.3.6 Parallel pruning

The ProperSYN system *prunes* a connection to a gate if the connection is found *redundant*, i.e., removal of that connection will not change the output function of any primary output. When the CSPF is computed for a connection, the *redundancy* of that connection is checked. If it is found *redundant*, permission is asked from the *master* processor. If the *master* processor finds it "valid," it broadcasts a message to other processors, and then performs the pruning.

Algorithm: Parallel Gate Synthesis (g_1, g_2, type)
/* This object tries to synthesize a gate of type "type" whose output function */
/* is a member of the intersection of CSPFs of g_1 and g_2 */

(1) /* When this object is awakened */
```
1.1 Checks if g1 and g2 exist in the network and have
     proper version of CSPFs
1.2 Put all gates in the network, except the successor of g1
     and g2 in possible_input_list
1.3 new_cspf = CSPF(g1) ∩ CSPF(g2)
1.4 partial_output_function = ∅
1.5 for ζ in possible_input_list
    1.5.1 If ''valid'' output function ζ is not available,
           ask work manager to get it.
    1.5.2 Otherwise, check if ζ can be connected to the synthesized gate
    1.5.3 If possible to connect, include ζ in partial_input_list
           and update partial_output_function.
    1.5.4 If partial_output_function is a member of new_cspf, minimize the
           number of input connections to the synthesized gate and ask
           permission from master processor for this gate merging and then exit
1.6 If no new output function to receive from work manager, exit
```

(2) /* This object receives a new output function */
```
2.1 Checks if g1 and g2 exist in the network and they have
     proper version of CSPFs
2.2 Checks if the output function is connectable to the synthesized gate
2.3 If connectable, checks if there is no change in the version of the
    output function of any gate in partial_input_list
2.4 If any change, recompute partial_output_function with available
    ''valid'' output function of the gates in partial_input_list
2.5 Update partial_output_function with the new output function received
2.6 If partial_output_function is a member of new_cspf,
    minimize the number of input connections to the synthesized gate and
    ask permission from master processor for this gate merging and then exit
2.7 If no new output function to receive from work manager, exit
```

Figure 2.17 Algorithm for Parallel Gate Synthesis of a given type

2.3.7 Parallel generalized gate substitution

As the name suggests, generalized gate substitution is a more general form of gate substitution. In this transformation, every output connection of a gate is substituted by an existing gate in the network. Let C be the set of output connections of a gate Γ. The gate Γ can undergo "generalized gate substitution" if

$$\forall c \in C,\ \exists\, \zeta \;\; such that\ output\_func(\zeta) \subseteq CSPF(c)$$

To keep the network loop-free, the gates which substitute the gate Γ must not be a transitive fan-out of Γ.

For each gate γ in the network, we create a *gen_gate_sub_object*. The job of the *gen_gate_sub_object* corresponding to the gate γ is to search if the gate γ can be generalized gate substituted. For that purpose, it needs the CSPFs of its output connections and the output functions of all of the gates except those which are transitive fan-outs of γ.

When a *gen_gate_sub_object* is awakened by a message, it starts by collecting the CSPFs and the output functions it needs. If it does not find some, it asks the *work manager* to supply them later. Then it compares the available CSPFs and output functions. If the output function of the gate ζ is a subset of the CSPF of an output connection c, then the output connection c can be substituted by the gate ζ. If all of the output connections of the gate γ can be substituted by some existing gates in the network, permission is asked from the *master* processor.

If the *gen_gate_sub_object* receives a CSPF of an output connection of γ, it first checks if this output connection can be substituted by an existing gate in the network. If all of the output connections can be substituted, it asks for permission from the *master* processor to generalized gate substitute γ. Similarly, when it receives an output function it asked for earlier, the output function is compared to check if it can substitute any unsubstituted output connection of γ. If all of the output connections can be substituted, permission is asked from the *master* processor.

When the *master* processor receives a request for permission for generalized gate substitution, it checks if the gates and connections exist in the network. Then it checks for the version number of the BDDs. If they have not changed, the substitution is permitted. If any version number has changed, the *gen_gate_sub_object* is asked to check for substitution again. The algorithm is given in Figure 2.18.

2.3.8 Parallel gate input reduction

In this transformation, one tries to reduce the input connection to an existing gate by synthesizing a new gate with a smaller number of input connections. In this process, the number of connections in the network will be reduced.

To synthesize a new gate which can replace an existing gate γ in the network, the output function of the new gate must be a subset of the CSPF of γ. The input connections to the new gate are from the existing gates in the network. To check if a gate can be connected to the new gate which has a given CSPF, the *effectively connectable* condition is used as given in [17].

For each gate in the network, four *gate_inp_red_objects* are created, one for each gate type: NAND, NOR, AND and OR. Each *gate_inp_red_object* tries to synthesize a new gate of a type supplied to it during its creation. The number of input connections to the new gate must be less than that to the existing gate. Other than that, the parallel algorithm is almost similar to that of the gate synthesis algorithm described in Section 2.3.5. Hence, the details of this algorithm are omitted for the sake of conciseness.

2.3.9 Overview of the ProperSYN algorithm

In the previous subsections, we have discussed the parallel algorithms for each of the transformations performed in the ProperSYN system. In this subsection, we will give the overview of the entire system.

We should mention that the ProperSYN system is not the same algorithm as the Transduction system. In the transduction system, the user specifies the *order* in which different transformations are to be applied at the beginning, and the transformations are applied strictly in that order, one after the other. A typical run of the Transduction

Algorithm: Parallel Generalized Gate Substitution (γ)
/* This is for gen_gate_sub_object γ */

(1) /* When the object is awakened */
```
1.1 Collect CSPFs of the output connections of γ.
1.2 Collect output functions of the gates which are not
     transitive fan-out.
1.3 Compare CSPFs and output functions for possible substitution.
1.4 Ask permission from master processor if substitution possible.
1.5 If some output functions and CSPFs not available, request to
     send them later.
1.6 If all possible pairs of CSPFs and output functions are
     considered, quit.
```

(2) /* When the object receives a new CSPF */
```
2.1 Compare this CSPF with available output functions for
     possible substitution.
2.2 If all output connections are covered, ask for permission
     for substitution.
2.3 If all possible pairs of CSPFs and output functions are
     considered, quit.
```

(3) /* When the object receives a new output function */
```
3.1 Compare this output function with the CSPFs of the output
     connections of γ.
3.2 If all output connections are covered, ask for permission
     for substitution.
2.3 If all possible pairs of CSPFs and output functions are
     considered, quit.
```

Figure 2.18 Algorithm for Parallel Generalized Gate Substitution method is shown in Figure 2.19(a). The order in which different transformations are applied is determined by the cost-performance ratio of each transformation. For example, the *gate substitution* has the lowest cost-performance ratio, i.e., checking for gate substitution is the cheapest operation. Hence, the *gate substitution* transformation is applied first on the network to reduce the size of the network. Then one applies the more expensive transformations such as *gate merging* to further reduce the size of the network. On the other hand, if one chooses to apply the *gate merging* transformation first on the network, it will take significantly more time to reduce the network.

In the case of the ProperSYN, although we use the same transformations as those for the Transduction method, applications of the different transformations are intermixed. All of the transformations are simultaneously started to avoid any synchronizing barrier. However, the advantages of a particular ordering are preserved by the choice of priorities. We experimented extensively with the actual priorities until we found good results. We assign different priorities to different transformations according to their cost to performance ratio to guide the synthesis process. The list of transformations in decreasing order of priorities is : *gate substitution, generalized gate substitution, gate input reduction* and *gate merging*. It is the same order in which the transformations are applied in the Transduction system. There is no *synchronizing barrier* at the end of each transformation. Each transformation is applied repeatedly until there in no more change in the network. A typical run of ProperSYN on 4 processors is shown in Figure 2.19(b). This figure emphasizes the fact that different processors may be trying different transformations at any particular time.

For any possible transformation, permission is asked from the *master* processor to maintain coherence in the network. We could not use the multiple master model (namely, one master per partition) due to the fact that for any transformation, the nodes affected are not localized in any region. As explained earlier, all of the nodes in the fan-in and fan-out cones are affected. Hence, we were forced to use a single master. But the number of communications to/from the *master* for the purpose of permission for transformations is much smaller than the total number of communications. The number of "successes" in

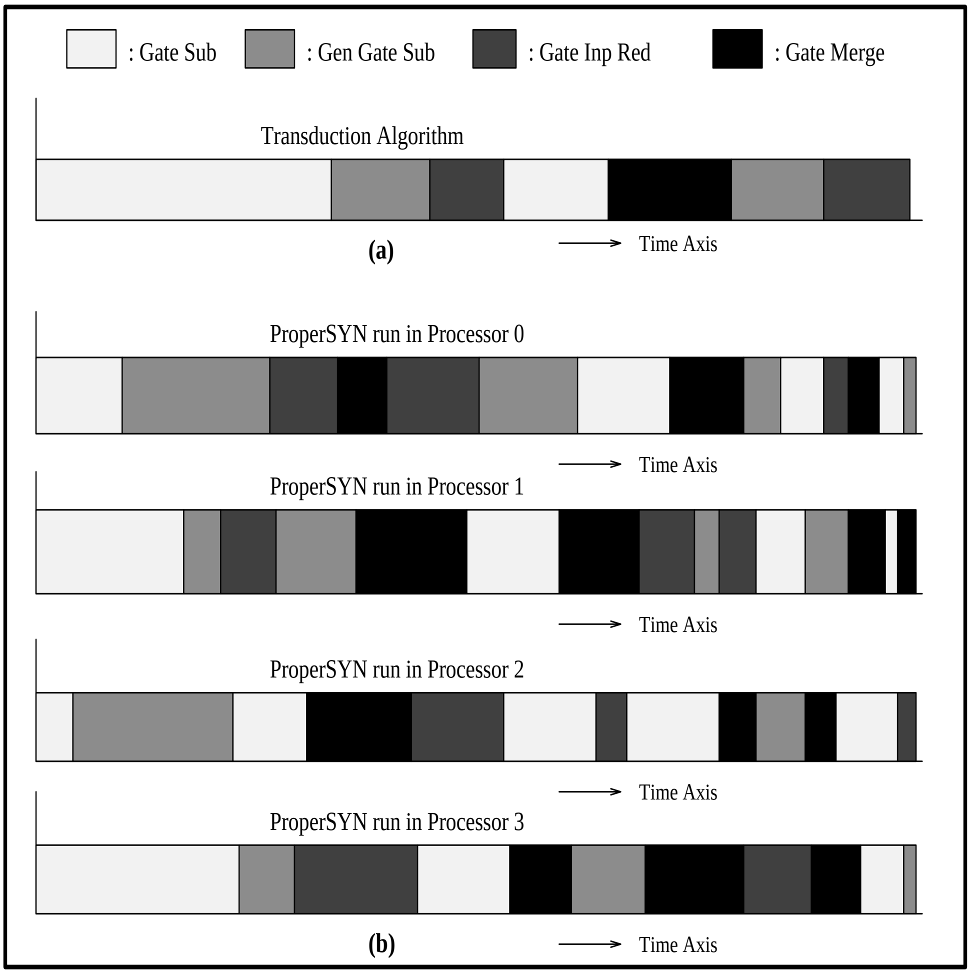

Figure 2.19 (a) A typical run of the Transduction method, (b) A typical run of ProperSYN with 4 processors

the search for possible transformations is very small compared to the number of searches. We found from extensive experimentation that the number of communications to/from *master* is less than 1% of the total communication.

2.4 Results

The parallel algorithms developed in the last section have been implemented as a part of the ProperCAD environment based on the CHARM run-time system and called ProperSYN. The system contains approximately 12000 lines of C code.

2.4.1 Run times and speedups on different parallel machines

In this subsection, we are going to present results of run-time and speedup information for some of the parallel machines on which ProperSYN currently runs. We would like to reiterate that to run ProperSYN in different machines, *no* change in the parallel program is necessary. The presence of "-" in any of the following tables indicate that data are not available due to limited time and limited availability of the machines.

Table 2.1 presents the result obtained in an Encore Multimax machine. It is a *shared memory MIMD* machine with 8 processors. Table 2.2 presents the results obtained in a Sequent Symmetry machine. It is another *shared memory MIMD* machine. Table 2.3 presents the results obtained in an Intel/860 machine. It is a *distributed memory MIMD* machine with 8 processors connected by a hypercube configuration. Table 2.4 presents the results in a network of workstations environment. This behaves as a *distributed processing* system. There is a *host* workstation, which distributes work to other workstations. The workstations used in this experiment are SUN4 workstations with a SPARC 1 processor in them. In all of the tables, we present run times for various *MCNC* benchmark circuits on different numbers of processors and speedups with respect to 1 processor runtime. One can observe that sometimes our algorithm produces "superlinear" speedups. This is due to the fact that synthesis is a search problem, and in parallel implementation of a search problem, *speedup anomaly* is possible. If one transformation precedes another

Table 2.1 Run times and speedups in Encore Multimax (shared memory MIMD)

CKT	1 Processor		2 Processors		4 Processors		8 Processors	
	Run Time	Spdup	Run Time	Spdup	Run Time	Spdup	Run Time	Spdup
5xp1	177.62	1.0	122.52	1.45	57.20	3.10	35.30	5.02
b9	231.24	1.0	123.35	1.87	57.82	3.99	40.21	5.75
bw	755.39	1.0	409.75	1.84	198.64	3.80	108.21	6.98
f51m	296.07	1.0	184.32	1.60	101.76	2.91	69.84	4.24
misex1	70.56	1.0	40.27	1.75	22.98	3.07	11.18	6.31
misex2	212.96	1.0	108.11	1.97	60.70	3.51	36.15	5.90
rd73	769.28	1.0	410.88	1.87	207.46	3.54	110.92	6.93
rd84	9159.49	1.0	3998.18	2.29	2390.73	3.83	1378.20	6.64
sao2	1174.38	1.0	659.78	1.77	315.15	3.73	152.03	7.72
vg2	1373.67	1.0	822.33	1.67	392.75	3.50	188.59	7.28
apex7	4868.01	1.0	2383.75	2.04	1341.27	3.63	727.92	6.69
apex6	54062.65	1.0	26976.26	2.00	14271.42	3.79	7241.64	7.47
duke2	40138.20	1.0	25390.33	1.58	12142.03	3.30	-	-
misex3c	50236.75	1.0	26412.00	1.90	13721.43	3.66	7011.62	7.16

transformation, the search space may be reduced considerably, which may result in a "superlinear" speedup.

2.4.2 Comparison of quality

In this section, we will compare the quality obtained by ProperSYN with that obtained by the Transduction method [17] version 1.2 and MIS 2.2 [16]. Since ProperSYN requires the network to consist of simple gates such as AND, OR, NAND, NOR and INVERTERS, we first perform a *technology mapping* of the given network using a simple gate library which contains only simple gates with at most 5 fan-ins. The purpose of this technology mapping is *only* to create the initial network, not for any other purpose. The transduction method generates the initial network using SYLON_XTRANS_INI [17], but we have not implemented that. Then ProperSYN is applied to that network, and at the end it is again technology mapped. In the case of MIS 2.2, we used the script provided in [38] to perform synthesis on the network. Then we performed *technology mapping* by using

Table 2.2 Run times and speedups in Sequent Symmetry (shared memory MIMD)

CKT	1 Processor		2 Processors		4 Processors		8 Processors	
	Run Time	Spdup	Run Time	Spdup	Run Time	Spdup	Run Time	Spdup
5xp1	443.73	1.0	183.01	2.42	94.06	4.72	62.11	7.14
b9	530.99	1.0	268.83	1.97	167.42	3.17	113.53	4.68
bw	1875.99	1.0	922.19	2.03	493.78	3.80	271.47	6.91
f51m	679.68	1.0	350.25	1.94	189.54	3.59	88.92	7.65
misex1	162.04	1.0	82.44	1.97	50.29	3.22	27.72	5.91
misex2	489.02	1.0	235.73	2.07	126.13	3.88	721.7	6.78
rd73	1766.60	1.0	1270.77	1.39	475.72	3.71	261.30	6.76
rd84	11032.91	1.0	5522.97	2.00	2791.69	3.95	1426.53	7.73
sao2	2696.72	1.0	1316.76	2.05	712.86	3.78	354.00	7.61
vg2	2299.64	1.0	1147.97	2.00	598.32	3.84	361.68	6.35
apex7	11178.38	1.0	4954.55	2.26	2871.58	3.89	1653.53	6.76
apex6	124142.60	1.0	61787.05	2.01	-	-	15871.98	7.82
duke2	92169.20	1.0	56531.91	1.63	24202.22	3.80	13075.20	7.05
misex3c	115358.18	1.0	60421.63	1.91	31592.42	3.65	15931.82	7.24

Table 2.3 Run times and speedups in Intel/860 hypercube (distributed memory MIMD)

CKT	1 Processor		2 Processors		4 Processors		8 Processors	
	Run Time	Spdup	Run Time	Spdup	Run Time	Spdup	Run Time	Spdup
5xp1	60.24	1.0	32.67	1.84	18.61	3.24	12.80	4.70
b9	70.23	1.0	33.12	2.12	21.90	3.20	15.42	4.55
bw	217.54	1.0	122.54	1.77	77.62	2.80	42.10	5.17
f51m	89.92	1.0	57.55	1.56	37.22	2.42	21.11	4.26
misex1	21.43	1.0	18.41	1.17	13.45	1.60	11.73	1.80
misex2	64.68	1.0	37.86	1.71	22.83	2.83	11.55	5.60
rd73	233.63	1.0	107.39	2.17	66.15	3.53	29.13	8.02
rd84	2381.77	1.0	1190.08	2.00	563.25	4.22	308.66	7.72
sao2	356.66	1.0	197.76	1.80	108.92	3.27	50.04	7.13
vg2	390.08	1.0	224.59	1.74	112.09	3.48	53.00	7.36
apex7	1478.43	1.0	884.24	1.67	461.96	3.20	222.94	6.63
apex6	15418.68	1.0	7700.81	2.00	3555.27	4.34	1842.22	8.37
duke2	12190.12	1.0	6371.77	1.91	3248.30	3.75	1686.32	7.23
misex3c	15257.09	1.0	7842.69	1.95	4074.12	3.75	1907.11	7.98

Table 2.4 Run times and speedups in network of workstations (distributed processing)

CKT	1 Processor		2 Processors		4 Processors	
	Run Time	Speedup	Run Time	Speedup	Run Time	Speedup
5xp1	102.99	1.0	55.49	1.86	31.97	3.22
b9	121.25	1.0	69.19	1.75	38.51	3.15
bw	468.19	1.0	264.80	1.76	172.58	2.71
f51m	180.97	1.0	118.07	1.53	72.24	2.51
misex1	41.68	1.0	22.66	1.84	13.92	2.99
misex2	109.00	1.0	60.83	1.79	35.85	3.04
rd73	387.74	1.0	251.32	1.54	129.42	3.00
rd84	4792.74	1.0	2476.32	1.94	1279.05	3.75
sao2	525.47	1.0	284.74	1.84	172.97	3.04
vg2	735.72	1.0	400.70	1.84	225.16	3.27
apex7	2272.41	1.0	1311.86	1.73	595.02	3.82

the same simple gate library. The Transduction method, XTRANS 1.2, was run with the script provided and then technology mapped. We present the results of different benchmark circuits in terms of the number of gates and the number of literals, gate/lit, in the final network in Table 2.5. As it can be observed from the results, ProperSYN produced better results than MIS 2.2 for almost all of the circuits. The results produced by ProperSYN and XTRANS 1.2 were comparable.

Table 2.6 shows the variation of the quality of the circuit produced with different numbers of processors. The result is shown for only one type of machine, and we have chosen the Intel/860 machine for that purpose. The result indicates that there is not much variation in the quality of the circuits and sometimes better circuits are produced with a larger number of processors. That can be due to the different orders in which different transformations are tried. The variations were similar for other parallel machines. The variation of quality with different numbers of processors in different parallel machines is shown in Table 2.7. The variation in the result from machine to machine is due to the nondeterministic nature of our algorithm and has to again do with the fact that different processor speeds on different machines can produce a different ordering of successful transformations.

Table 2.5 Comparison of quality of network produced by ProperSYN, the Transduction method and MIS 2.2

CKT	Initial (gate/lit.)	MIS 2.2 (gate/lit.)	XTRANS 1.2 (gate/lit.)	ProperSYN (gate/lit.)
5xp1	133/276	106/216	76/145	64/124
b9	106/227	116/227	*	87/185
bw	207/480	141/276	126/255	104/267
f51m	134/269	108/214	71/132	81/160
misex1	69/154	49/91	37/76	39/83
misex2	76/221	83/166	83/168	68/175
rd73	136/322	80/165	111/206	83/185
rd84	299/671	139/275	163/310	131/305
sao2	129/307	104/221	100/191	94/224
vg2	165/390	64/139	77/152	70/161
duke2	468/1206	353/711	320/633	292/745
misex3c	473/1202	393/792	328/676	314/821
apex7	241/474	210/420	*	176/372
apex6	765/1432	747/1406	*	615/1224

Note: ''*'' means XTRANS 1.2 can not handle this circuit
Note: XTRANS 1.2 does not handle any circuit with # PI > 32

Due to the nondeterministic nature of our algorithm, the quality and run time varied slightly from run to run for more than one processor. We performed an experiment to determine the distribution of the variations of the quality and the run time. For the benchmark circuit *bw*, we ran our algorithm in Encore Multimax for 20 times each for varying numbers of processors (2, 4 and 8). The distribution of the number of gates in the final circuit and the run time are shown in Figure 2.20.

2.5 Conclusions

In this chapter, we have presented a portable parallel algorithm of logic synthesis, called ProperSYN. ProperSYN has been implemented as a part of the ProperCAD environment [25] based on the CHARM system [26, 27], and it runs on a variety of parallel machines such as Sequent Symmetry, Encore Multimax, Intel/860 Hypercube and networks

Table 2.6 Comparison of Quality of Network with different number of processors in Intel/860

CKT	1 Processor (gate/lit.)	2 Processors (gate/lit.)	4 Processors (gate/lit.)	8 Processrs (gate/lit.)
5xp1	64/124	67/132	67/132	66/130
b9	87/185	87/185	89/185	87/189
bw	104/267	110/253	110/262	109/262
f51m	81/160	71/144	73/146	77/154
misex1	39/83	40/87	41/89	40/88
misex2	68/175	67/177	68/178	69/179
rd73	83/185	93/207	89/198	92/208
rd84	131/305	137/327	145/353	-
sao2	94/224	93/225	94/219	95/227
vg2	70/161	69/158	73/166	73/168
apex7	176/372	180/378	177/372	177/375
apex6	615/1224	615/1224	613/1211	620/1231
duke2	292/745	299/760	302/764	305/775
misex3c	314/821	316/830	320/835	320/835

Table 2.7 Comparison of Variation of Quality of Network $5xp1$ for Intel/860, Encore Multimax and network of Workstations

Machine	1 Processor (gate/lit.)	2 Processors (gate/lit.)	4 Processors (gate/lit.)	8 Processors (gate/lit.)
Intel/860	64/124	67/132	67/132	66/130
Encore Multimax	64/124	66/129	65/130/	67/135
Network of Workstations	64/124	65/130	68/132	-

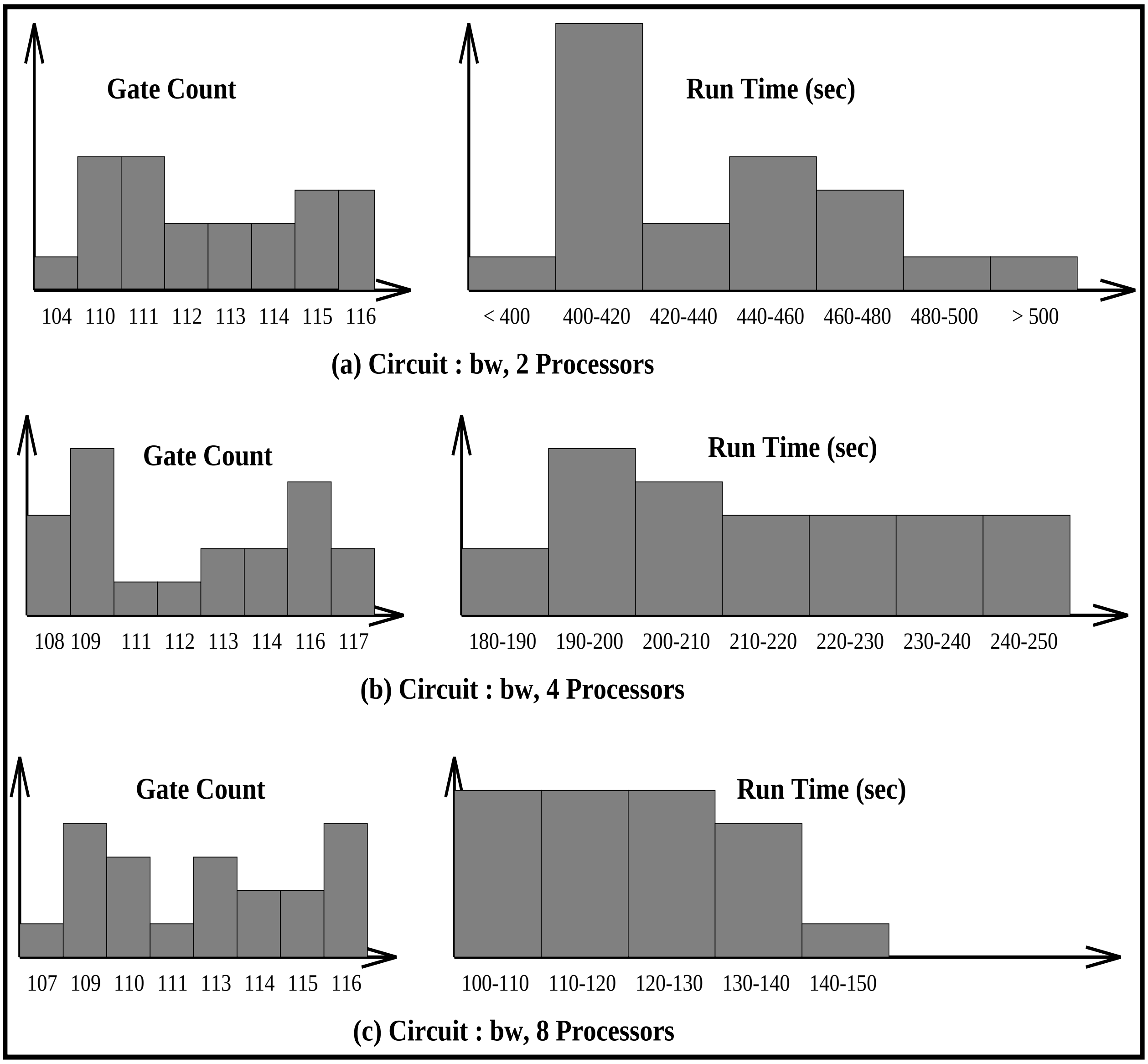

Figure 2.20 Distribution of the number of gates and the run time for circuit *bw* in Encore Multimax for 20 runs with (a) 2 processors, (b) 4 processors and (c) 8 processors

of workstations. The algorithm developed for ProperSYN uses an asynchronous message driven data-flow model of computation. No explicit synchronizing barrier is allowed in the algorithm, hence our algorithm is developed to scale well with larger numbers of processors. We have implemented the algorithm and have evaluated it on various *MCNC* synthesis benchmark examples. Some results are reported on different parallel machines. Results show excellent speedup in all of the machines with almost no degradation in the quality of the synthesized network over the uniprocessor algorithm.

For large combinational circuits, the output functions and the CSPFs will be very large and will require huge memory space. In our present algorithm, many copies of the same BDD may be residing in different processors. In the future, we propose to use caching techniques, in which the BDDs of a particular part of the network will be permanently stored in that processor which "owns" that part of the network. If any other processor needs those BDDs, it will first look for them in its local "cache" and if not found, ask the "owner" processor to send it. If the local "cache" size is small, the number of extra copies of a BDD will be small. In that case, we will be able to handle really large circuits.

For really large circuits, when the BDD size becomes too large to fit in the memory, the partitioning techniques based on [39], similar to the work reported by Lim et al. [23, 24], can be used. Lim has proposed to use both *interpartition* and *intrapartition* parallelism for really large circuits. The use of only interpartition parallelism for very large circuits is discussed in Chapter 4 of this thesis.

CHAPTER 3

PARALLEL ALGORITHM USING THE MIS APPROACH

3.1 Introduction

In the last chapter, we presented a portable parallel algorithm for logic synthesis using the Transduction method. Another important combinational synthesis approach is the MIS approach of synthesis [16]. The key features of that approach are 1) algebraic factoring and 2) node simplification using two-level logic minimization techniques. A previous attempt to develop a parallel algorithm for MIS did not perform well due to the presence of synchronizing barriers [20]. In this chapter, we present a new parallel algorithm called ProperMIS for logic synthesis using the MIS approach of algebraic factoring and node simplification in the ProperCAD environment based on the CHARM system [26, 27].

The parallel algorithm we present in this chapter has three major contributions. First, our algorithm uses an *asynchronous* message-driven computing model with no synchronizing barriers separating phases of parallel computation. Second, our algorithm is *portable* across a wide variety of parallel architectures, both *shared* and *distributed* memory machines. Finally, our algorithm is built around a well-defined *sequential algorithm interface*, so that we can benefit from future expansion of the sequential algorithm. One of the major contributions of this work is the fact that we have *reused nearly 80%* of the sequential code from the MIS synthesis program in our parallel implementation. This work has been submitted for publication [40, 41].

The rest of this chapter is organized as follows. Section 3.2 gives an overview of the MIS approach of logic synthesis using algebraic factoring and node simplification. Section 3.3 describes the general methodology of the parallel algorithm. Section 3.4 describes the parallel algorithms for the algebraic factoring methods, namely, *kernel extraction*, *cube extraction* and *resubstitution*. Section 3.5 describes the parallel algorithm for the node simplification method. Section 3.6 provides an overview of the ProperMIS system. Section 3.7 describes the experimental results obtained.

3.2 Overview of MIS Synthesis Method

3.2.1 Definitions

3.2.1.1 General definitions

In this section, we will provide some basic definitions as given in [16] to be used later in this chapter.

A *variable* is a symbol representing a single coordinate of the Boolean space (e.g., $\overline{a}$). A *literal* is a variable or its negation (e.g., a or $\overline{a}$).

A *cube* is a set C of literals such that $x \in C$ implies $\overline{x} \notin C$ (e.g., $\{a, b, \overline{c}\}$ is a cube and $\{a, \overline{a}\}$ is not a cube). A cube represents the conjunction of its literals.

An *expression* is a set f of cubes. For example, $\{\{a\}, \{b, \overline{c}\}\}$ is an expression consisting of two cubes $\{a\}$ and $\{b, \overline{c}\}$. An expression represents the disjunction of its cubes. We use the conventional algebraic notation for cubes and expressions. For example, the cube $\{a, b, \overline{c}\}$ can be written as $ab\overline{c}$ and the expression $\{\{a\}, \{b, \overline{c}\}\}$ can be written as $a + b\overline{c}$.

An expression is *nonredundant* if no cube in the expression properly contains another cube. For example, $a + ab$ is redundant because $\{a\} \subset \{a, b\}$. A *Boolean expression* is a nonredundant expression.

The *support* of an expression f is

$$sup(f) = \{a | a \, \exists \, cube \, C \, \in \, f \, such \, that \, x \in C \, or \, \overline{x} \in C\}$$

For example, $sup(ab + a\overline{c}) = \{a, b, c\}$. Two expressions f and g have *disjoint support* if $sup(f) \cap sup(g) = \emptyset$.

The *primary divisors* of an expression f are the set of expressions

$$D(f) \;=\; \{f/C | C \, is \, a \, cube\}$$

The *kernels* of an expression f are the set of expressions

$$K(f) \;=\; \{g | g \in D(f) \, and \, g \, is \, cubefree\}$$

In other word, 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.

3.2.1.2 Boolean network definitions

A Boolean network is a technology-independent structure for representing a multilevel logic function. It 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 nodes pointing to node i. The *fan-out* of a node i is the set of all nodes which node i points to. The *transitive fan-in* of a node i is the set of all the nodes which has a directed path to node i in the graph. The *transitive fan-out* of a node i is the set of nodes to which there is a directed path from node i.

3.2.1.3 Don't care definitions

In this subsection, we will provide some definitions, most of them are provided in [37]. The *external don't care* set DX_i of a primary output i of a Boolean network is a set of primary input combinations. A primary input combination, π, is in that set if π *will never occur* or for π the primary output i *will not be used.*

The *cofactor* of a sum-of-products representation, $F = \{c_i\}$, of a Boolean function with respect to a literal v_j is defined to be

$$(F)_{v_j} \;=\; \bigcup_i (c_i)_{v_j}$$

where

$$(c_i)_{v_j} = \begin{cases} c_i - \{v_j\} & v_j \in c_i \\ \emptyset & \overline{v_j} \in c_i \\ c_i & \text{otherwise} \end{cases}$$

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. The satisfiability don't care set for a node arises from the fact that y_i can take only those values determined by f_i.

The *transitive fan-out don't care* set of a node i is defined to be

$$DTF_i = \bigcap_{j \in PO \cap TFO_i} DTF_{ji}$$

where

$$DTF_{ji} = \{x \in B^n | (v_j)_{v_i}(x) = (v_j)_{\overline{v_i}}(x)\}$$

where B^n denotes the Boolean space induced by the primary inputs (assuming the number of primary inputs to be n), PO denotes the set of primary outputs and TFO_i denotes the transitive fanout of node i.

The *fan-out don't care* set of a node i is defined to be

$$DT_i = \bigcap_{j \in FO_i} DTF_{ji}$$

where

$$DTF_{ji} = ((f_j)_{y_i} \equiv (f_j)_{\overline{y_i}}) = (f_j)_{y_i}.(f_j)_{\overline{y_i}} + \overline{(f_j)_{y_i}}.\overline{(f_j)_{\overline{y_i}}}$$

where FO_i denotes the fan-out of the node i. The fan-out don't care set of a node i is a subset of the transitive fan-out don't care set of that node.

3.2.2 Types of logic transformations

3.2.2.1 Kernel extraction

In this transformation, a multiple cube divisor common to two or more functions is identified which can be used to reduce the total number of literals in the network.

Because the size of the set of all algebraic and Boolean divisors is very large, and because an algorithm for efficiently generating only useful Boolean divisors does not exist, we restrict ourselves to only algebraic divisor.

First, the kernels for all the nodes are generated. There exists a multiple-cube common divisor δ for l nodes η_1, η_2, ..., η_l if

$$\exists\, k_{\eta_1} \in K(\eta_1),\ k_{\eta_2} \in K(\eta_2), \ldots k_{\eta_l} \in K(\eta_l),\ such\,that\ k_{\eta_1} \cap k_{\eta_2} \ldots \cap k_{\eta_l} \ = \ \delta$$

Hence, all possible kernel intersections are performed and the corresponding multiple-cube divisors, if they exist, are recorded. Then the best divisor (which saves the maximum number of literals if included in the network) is chosen and the network is updated. This process continues until no more multiple-cube common divisors can be found.

For example, using the equations in a Boolean network as given in [42]

$$\begin{aligned} F &= af + bf + ag + cg + ade + bde + cde \\ G &= af + bf + ace + bce \\ H &= ade + cde \end{aligned} \tag{3.1}$$

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

$$\begin{aligned} F &= deX + fX + ag + cg \\ G &= ceX + fX \\ H &= ade + cde \\ X &= a + b \end{aligned} \tag{3.2}$$

The original network had 33 literals, and the modified network after the kernel extraction has 22 literals.

3.2.2.2 Cube extraction

In this transformation, a single-cube divisor common to two or more functions is identified which can be used to reduce the number of literals in the network.

There exists a single-cube common divisor ζ for m nodes, η_1, η_2, ..., η_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. Hence, to find a single-cube common divisor, all possible cube intersections are performed and the common divisor which reduces the literal count of the network maximally is chosen. This process is repeated until no more single-cube common divisor can be found.

For example, given the equations in a Boolean network as given in [42]

$$\begin{aligned} F &= abc + abd + eg \\ G &= abfg \\ H &= bd + ef \end{aligned} \tag{3.3}$$

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

$$\begin{aligned} F &= Xc + Xd + eg \\ G &= Xfg \\ H &= bd + ef \\ X &= ab \end{aligned} \tag{3.4}$$

The original network had 16 literals. After the cube extraction, the number of literals in the modified network is 15.

3.2.2.3 Resubstitution

This transformation is used to check whether an existing function itself is a divisor of other function. For example, suppose the network is

$$\begin{aligned} x &= ac + ad + bc + bd + e \\ y &= a + b \end{aligned} \tag{3.5}$$

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

$$x = y(c+d)+e \tag{3.6}$$

3.2.2.4 Node simplification

Each node of a Boolean network is is a Boolean function, expressed in a sum-of-products format (two-level logic). Two-level minimization is a very developed science, and very efficient algorithms exist for finding minimal two-level representations of Boolean functions. Hence, two-level minimization algorithms such as *espresso* [14] can be used to minimize each node of the Boolean network.

Two-level minimization can be made more powerful in the context of multilevel circuits. Previously in this section, we discussed different kinds of don't cares that arise due to the multilevel structure. One can choose the don't care set depending on how thorough and fast the simplification process needs to be.

3.3 General Methodology for Parallel Algorithm

In our ProperMIS algorithms, the logic network is divided into a set of nonoverlapping partitions. However, it should be noted that the partitioning is for *parallelization purposes only*; we *do not perform synthesis on those partitions separately.* The logic minimization is performed on the entire network as a whole. This approach of partitioning for the purpose of parallelization only was also followed for the ProperSYN system described in Chapter 2. We use the partitioning algorithm described in Figure 2.7.

For each partition, one *partition_object* is created. The object corresponding to the partition p is denoted as *partition_object(p)*. 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 CHARM run-time system can automatically distribute these objects to different processors. But the CHARM system assumes all of the objects to be of equal size when it distributes the objects for load balancing. But in this case, all of the nodes in the circuit may not be of the same size. That may lead to load imbalance and bad speedup. The distribution strategy is discussed in a later section. An example of distribution of objects is depicted in Figure 2.8. Each processor has a copy of the network, but has the responsibility for the shaded regions of the network. Effectively, we can say that each processor "owns" some parts of the network.

A major problem we have to deal with is the fact that different processors may perform different optimizations simultaneously on the network. Hence, to provide some coherence mechanism among various applications of these optimizations in parallel, we used the concept of *version number* of the nodes. Every node in the Boolean network has a *version number* attached to it. Whenever the functionality of that node changes due to some transformation, the version number of that node increases by 1. Also, we have designated a processor, called the *master* processor. The *master* processor has to give permission before any transformation can take place. Whenever any processor finds a possible transformation on a node η, it asks for permission for that transformation and it provides the *version number* of η in the request. When the *master* processor picks up the request, it checks if the *version number* provided in the request is the same as the current *version number* of η. If it is so, then it gives the permission because the functionality of η has not changed since it was checked for the possible transformation. Otherwise, the permission is denied. A similar coherence mechanism is described in Chapter 2 of this thesis.

3.4 Parallel Algorithm for Factoring

3.4.1 Parallel algorithm for kernel extraction

In Section 3.2, we have defined the kernel extraction process. From the definition, it is apparent that we have to generate the kernels for all the nodes. Then the intersection

```
KERNELS(j, g)
  R = ∅
  for(i = j; i ≤ n; i++) {
    if(l_i appears in more than one cube) {
      c = largest cube dividing g/{l_i} evenly
      if(l_k ∉ c for all k < i) {
        R = R ⋃ KERNELS(i+1, g/{l_i} ⋃ c)
      }
    }
  }
  R = R ⋃ {g}
  return R
```

Figure 3.1 Kernel Generation Algorithm

of different subsets of the kernels generates possible factors, out of which the best factor is chosen to create a new node in the circuit.

In our algorithm, we generate all possible kernels for each node. We have used the algorithm described in [16] for kernel generation. For a node f whose kernels have to be generated, the literals in the support of f are numbered from 1 to n. The algorithm for kernel generation is given in Figure 3.1. The procedure KERNELS is a recursive procedure and g is a cube-free expression. To use the algorithm for generating all kernels of f, KERNEL(1, g) is called, where g is the resultant expression after the cube χ is factored out of f and χ is the largest cube dividing f evenly.

The algorithm works as follows. The argument j in KERNELS is designed to find all of the kernels associated with any co-kernel not containing any literals l_i for i < j. The recursive call to KERNELS restricts the terms to those containing literal i. These are then kernels which have as co-kernel a cube whose literals include l_i and the literals of c, the largest cube factor of g/l_i. The recursion is done only if the cube c has no literals $k \leq i$, since all co-kernels associated with this recursion will involve the literals of c, and if one of them has been factored already, we would reproduce a kernel and co-kernel already found.

The kernels for the nodes can be generated in parallel. A message is sent to all of the *partition_objects* to generate the kernels of the nodes in that partition. The *partition_object(i)* will generate the kernels for all of the nodes in partition i. Then it will broadcast that information to all of the processors.

Finding useful intersections of kernels is facilitated with a concept called *co-kernel cube* matrix, as described in [42]. 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 example, consider the equations given in Equation (3.1). 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 (co-kernels) of G are $ce + f$ (a, b), $a + b$ (f, ce), and the only kernel of H is $a + c$ (de). The expressions F and G are also kernels (because they are cube-free), but they are not listed. The expression H is not a kernel as it is not cube-free.

The co-kernel cube matrix can be constructed from the data described as done in [42]. For each kernel (co-kernel), there is a row in the matrix (total 11). The *unique* cubes from all of the kernels are a, b, c, ce, de, f and g; these cubes are used to label the columns of the matrix. The product of a co-kernel for a row and a kernel-cube for a column yields a cube of some expression. For reference, the cubes of the original expressions are numbered from 1 to 13. The cube number of the cube resulting from the product of the co-kernel for row i and the kernel-cube for column j is placed at position (i, j) in the co-kernel cube matrix. For example, the co-kernel a when multiplied by the kernel $de + f + g$ yields the cubes numbered 5, 1, 3 (which are ade, af and ag). Note that there is often more than one way to form each cube in an expression. For example, cube 1 (af) is created by the co-kernel a multiplying the kernel $de + f + g$, and by the co-kernel f multiplying the kernel $a + b$. The co-kernel cube matrix for the example is given in Table 3.1.

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

Table 3.1 An example of co-kernel cube matrix for the set of expressions in Equation (3.1)

			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	.	.	.	.

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 the kernel generations are done in parallel and then broadcast to all of the processors, the order in which the kernels are received may be different 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 Table 3.1. But 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 Table 3.2, which has different labeling for the rows and the columns. To perform rectangular covering in parallel, we must make the co-kernel cube matrices in all of the processors the same, irrespective of the order in which they receive the kernel information.

To make the row and column labeling consistent in all of the processors, we have used the following strategy. For each node, we have assigned an interval of labels. For

Table 3.2 Another example of co-kernel cube matrix for the set of expressions in Equation (3.1)

			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	.	.

example, we can assign the intervals [1–1000], [1001–2000] and [2001–3000] to the nodes F, G and H, respectively. Thus, any kernel (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.

For column labeling, we have used a similar strategy. The unique kernel cubes generated for each node will be numbered initially using a number from the interval assigned to that node. For example, the kernel cube a will be labeled as 1 while generating kernels for F, 1001 while generating kernels for G amd 2001 while generating kernels for the node H. When the same kernel cube arrives in a processor with a different label, it will choose the minimum of the labels to label the column corresponding to that kernel cube. Hence, after kernels of all of the nodes are received, the label of the column corresponding to the kernel cube a will be 1 irrespective of the order in which kernels are received. The co-kernel cube matrix will finally look like the one given in Table 3.3. But in any intermediate time, the labeling can be different in different processors. For example, if the

Table 3.3 Consistent co-kernel cube matrix for the set of expressions in Equation (3.1)

			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	.	.	.	.

kernels of the nodes are received in the order H, G and F, then the column labeling for the kernel cube a will change from 2001 to 1001 to 1.

A rectangle of the co-kernel cube matrix identifies an intersection of the kernels; this kernel-intersection 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. The entries covered by the matrix correspond to the cubes from the original network. For example, the prime rectangle ({3, 4, 1003, 1004}, {1, 2}) in Table 3.3 (shown in italics) identifies the subexpression $a + b$ which divides the functions F and G. The cubes numbered 1, 2, 5, 6, 8, 9, 10, and 11 from the original set of functions are covered by this rectangle. The corresponding factorization is given in Equation (3.2). Another rectangle is ({3, 2001}, {1, 3}), which identifies the subexpression $a + c$ and divides the functions F and H.

The *weight* of a rectangle of the co-kernel cube matrix is chosen to reflect the number of literals in the network if the corresponding subexpression is inserted in the network. A minimum-weighted rectangular cover of the co-kernel cube matrix then corresponds to a simultaneous selection of a set of subexpressions to add to the network in order to minimize the total number of literals in the network.

Let w_j^c be the number of literals in the kernel-cube for column j. If a rectangle (R, C) is used to identify a subexpression, then a new function is formed using the columns of C. This new function has $\sum_{j \in C} w_j^c$ literals. The rows of the rectangle indicate that the expressions that will be divided by the subexpression. After algebraic division by the subexpression, each of these expressions consists of a sum of the corresponding co-kernel cubes multiplying the literal for the new expression. Let w_i^r be the number of literals in the co-kernel for row i plus 1. Therefore, the number of literals in the affected functions after extraction of the rectangle is $\sum_{i \in R} w_i^r$. Therefore, the weight of a rectangle (R, C) of the co-kernel cube matrix is defined as

$$w(R, C) \;=\; \sum_{i \in R} w_i^r \;+\; \sum_{j \in C} w_j^c \tag{3.7}$$

The *value* of a rectangle measures the reduction of the number of literals in the network if the particular rectangle is selected. The number of literals after the rectangle is selected is the weight of the rectangle as defined above. Let V_{ij} be the number of literals in the cube which is covered by position (i, j) of the co-kernel cube matrix. Then the number of literals before the extraction of the rectangle is simply $\sum_{i \in R, j \in C} V_{ij}$. Hence, the value of a rectangle (R, C) of the co-kernel cube matrix is defined as

$$v(R, C) \;=\; \sum_{i \in R, j \in C} V_{ij} \;-\; w(R, C) \tag{3.8}$$

The rectangular covering is performed by generating all possible rectangles and selecting the maximum valued rectangle as described in [42].

The search for the maximum valued rectangle can be performed in parallel. The procedure is conceptually illustrated in Figure 3.2 and illustrated with the help of a *co-kernel cube matrix* in Figure 3.3. We designate a processor as the *master* processor. After the generation of all of the kernels, the *master* processor creates an object called *kernel_extract_object* to compute the maximum-valued rectangle by generating all prime rectangles.

When a new *kernel_extract_object* is created, the following information is passed: the submatrix of the co-kernel cube matrix (M), the rectangle generated thus far $(rect)$ and

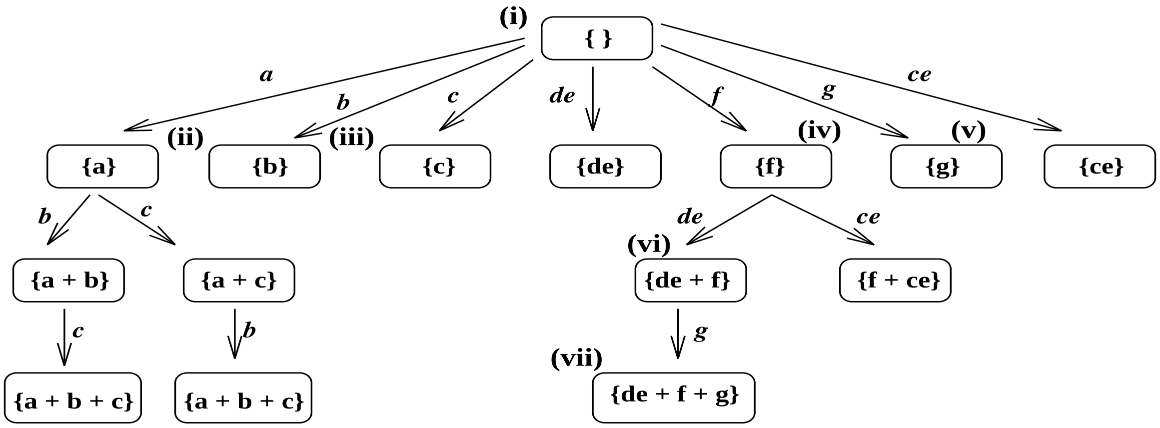

Figure 3.2 Conceptual search space for finding the maximal-valued rectangle

the index of the column from which it should search onwards for new prime rectangles (*index*). For the root *kernel_extract_object*, M is the co-kernel cube matrix itself, *rect* is a empty rectangle and the value of *index* is 0 as shown in Figure 3.3(i). Conceptually, it is the root of the search tree as shown in Figure 3.2(i), corresponding to the null subexpression. The responsibility of this *kernel_extract_object* is to generate all prime rectangles with fewer rows but more columns than *rect*. That is performed by adding columns with labels more than or equal to *index* to *rect*. Conceptually, it is equivalent to generating all 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*) to the *kernel_extract_object*. It also computes the value of *rect* and records the information if it is the best rectangle seen by this object.

If the number of columns in the submatrix M is less than a user-defined threshold, all possible prime rectangles are generated by applying the sequential algorithm shown in Figure 3.4 as given in [43].

Otherwise, new child objects are created and the search space is divided among those child objects. Each column c with a label greater than *index* is examined as a column to include in the rectangle. If the column has only a single element, then it cannot create a nontrivial rectangle, thus only columns with 2 or more elements are of interest. The submatrix $M1$ of the original matrix M is created by selecting only the rows in which

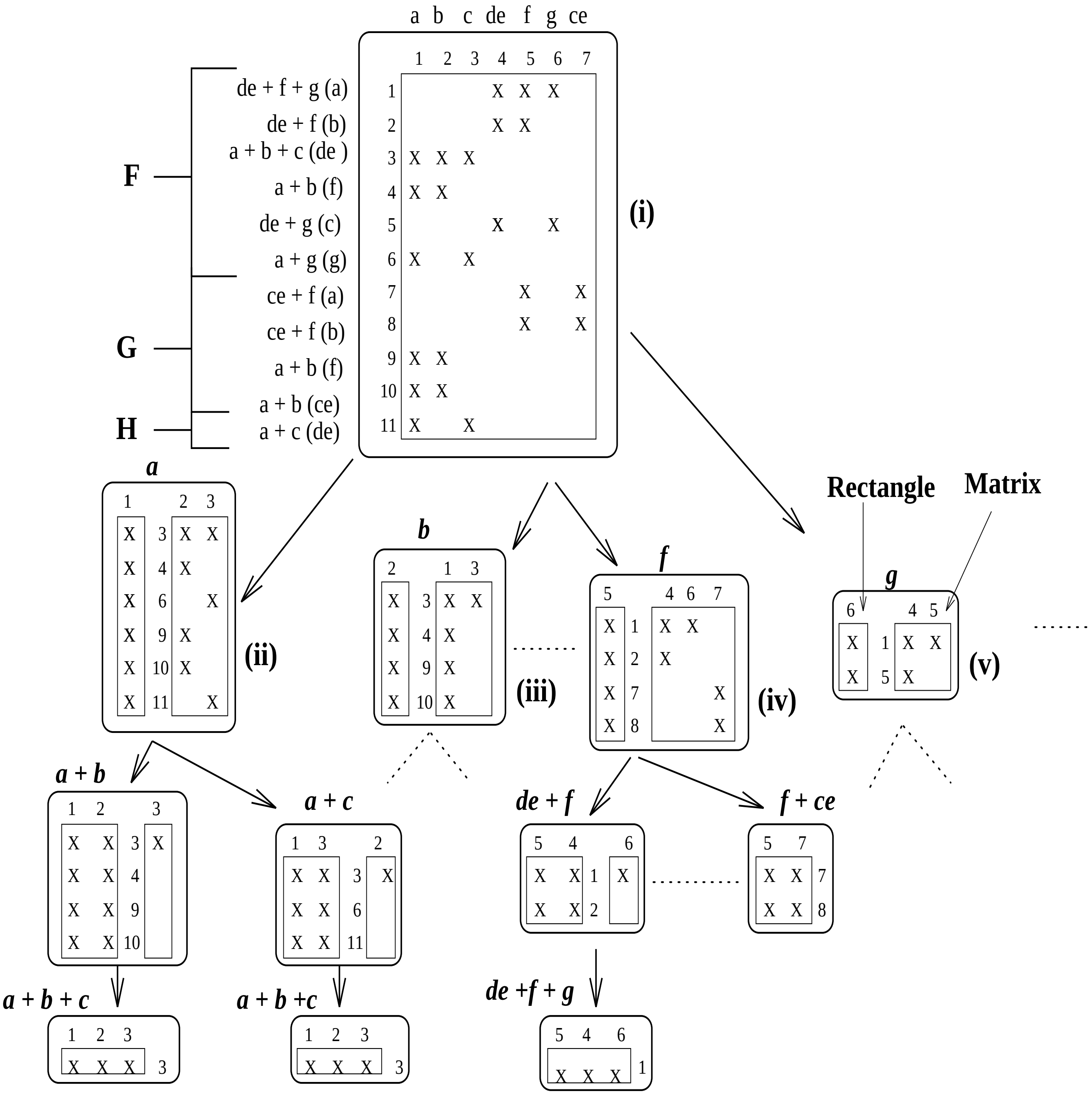

Figure 3.3 An example of performing kernel extraction by creating kernel_extract_objects

```
max_val_rect_recur (M, index, rect)
  foreach column c of M
    if(c->length >= 2 && c->column >= index)
      M1 = new matrix;
      foreach element p in column c
        copy row p->rownum of M to M1;
      rect1 = new rectangle;
      rect1->rows = duplicate column c of M;
      rect1->cols = duplicate rect->cols;
      prune = 0;
      foreach column c1 of M1
        if(c1->length == c->length)
          if(c1->colnum < c->colnum)
            prune = 1;
            break;
          else
            add c1->column to rect1->cols;
            delete column c1 from M1;
      if(not prune)
        compute value of the rectangle rect1;
        if it is maximum-valued rectangle so far, save it;
        max_val_rect_recur(M1, c->column, rect1);
```

Figure 3.4 Sequential algorithm for generating all of the rectangle and finding the maximum-valued rectangle

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. Any column of the submatrix $M1$ (including c) which is now all nonzero can also be added to the rectangle. At this point, *rect1* represents a new prime rectangle of the matrix, and $M1$ is a new submatrix to be searched for more prime rectangles. At this point, a new *kernel_extract_object* is created with the following arguments: the submatrix $M1$, the new rectangle *rect1* and the index c->column. 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 3.3(ii). This object receives the submatrix $M1$, the rectangle $rect1$ as shown in Figure 3.3(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 3.2(ii). The label of an edge in Figure 3.2 represents the kernel cube that will be added to the subexpression at the source node of that edge. For example, by adding the kernel cube de to the subexpression f in node (iv), a new subexpression $de + f$ is generated in node (vi) and by adding the kernel cube g to that, another new subexpression $de + f + g$ is generated in node (vii) in Figure 3.2. Similarly, the effects of adding columns 2, 5 and 6 to the empty rectangle at the root are shown in Figure 3.3(iii), 3.3(iv), 3.3(v) respectively. The corresponding nodes in the conceptual domain in Figure 3.2 are (iii), (iv) and (v) respectively.

If a *kernel_extract_object* generates prime rectangles by calling the sequential algorithm (*max_val_rect_recur*) given in Figure 3.4, it reports the best rectangle seen by that object to its parent after the computation is performed. If it creates child objects, it waits for them to report the best rectangles seen by them and then it computes the best rectangle by picking the best of them. It then reports that information to its parent. The root *kernel_extract_object* finally reports the best rectangle information to the *master* processor. The parallel algorithm for kernel extraction is given in Figure 3.5.

The child objects created dynamically are automatically distributed to different processors by the CHARM run-time system for good load balancing.

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. It also determines the set of expressions that can be divided by that subexpression. It broadcasts that information to all other processors. All of the processors perform the divisions. All of the kernels of the affected expressions are removed from the co-kernel cube matrix. The *version numbers* of the affected nodes are incremented by 1. The *partition_objects* corresponding to the partitions in which those affected nodes reside are sent messages to generate kernels again for those nodes. A new node is created for the subexpression and it is allocated to a suitable partition. Then a message is sent to that *partition_object* to create kernels for that node. Then the *master* processor creates another *kernel_extract_object* to continue searching for more kernel extractions. At this

```
Algorithm: Parallel Kernel Extraction (M, index, rect)
/* This is for kernel_extract_object(M, index, rect) */
(1)  /* When object is awakened */
     compute value of the rectangle rect and record;
     numchild = 0;
     if(number of columns of M <= threshold)
        call max_val_rect_recur (M, index, rect);
        report best rectangle to parent;
     else
        foreach column c of M
          if(c->length >= 2 && c->column >= index)
            M1 = new matrix;
            foreach element p in column c
              copy row p->rownum of M to M1;
            rect1 = new rectangle;
            rect1->rows = duplicate column c of M;
            rect1->cols = duplicate rect->cols;
            prune = 0;
            foreach column c1 of M1
              if(c1->length == c->length)
                if(c1->colnum < c->colnum)
                  prune = 1; break;
                else
                  add c1->column to rect1->cols;
                  delete column c1 from M1;
            if(not prune)
              Create kernel_extract_object(M1, c->column, rect1);
              numchild = numchild + 1;

(2)  /* When it receives the best rectangle info from a child object */
     if(child best rectangle better than best rectangle seen thus far)
        record the information of best rectangle;
     if(number of children reported == numchild)
        report best rectangle to parent;
```

Figure 3.5 Parallel Kernel Extraction Algorithm point, it does not wait for the regeneration of the kernels of the affected nodes. But the *kernel_extract_object* will be created with lower priority than that of the message for kernel regeneration. Therefore, it is very likely that kernels for the affected nodes will be re-generated before the *kernel_extract_object* is picked up by some processor, but it is not guaranteed. We have used this strategy to remove synchronizing barriers at every stage of kernel extraction.

For the sake of simplicity, we mentioned in the algorithm in Figure 3.5 that we pass the submatrix M and the rectangle *rect* to the new *kernel_extract_object*. But we have not proceeded exactly in that fashion. The size of the submatrix can be very large and passing that information may be very expensive. Instead, we pass only the information of the set of columns covered thus far. When a *kernel_extract_object* is awakened by a message, it computes the submatrix M and the rectangle *rect* by using that information and the co-kernel cube matrix in the local processor. For that reason, we have to maintain the co-kernel cube matrix in all of the processors.

3.4.2 Parallel cube extraction

In Section 3.2, we defined the cube extraction process. To perform the cube extraction, all possible cube intersections are generated and the common cube which reduces the literal count of the network maximally is extracted as the common subexpression.

To perform all possible cube intersections, a *cube-literal matrix* is formed at all of the processors. Unlike kernel generation, we do not distribute the job of creating the *cube-literal matrix* to different objects. The creation of the *cube-literal matrix* is a relatively inexpensive process. Each expression is already expressed in sum-of-products form. As a result, all of the cubes of the expressions are immediately available; they simply have to be filled up in the *cube-literal matrix*. Thus, each processor creates its *cube-literal matrix* independently.

Every expression (and the associated literal) in the network is assigned a unique number, called *node-id*, when it is created in the Boolean network. That number is the same in all of the processors. Each expression in the Boolean network is assigned

Table 3.4 Consistent cube-literal matrix for the expressions in Equation (3.3)

		a	b	c	d	e	f	g
		1	2	3	4	5	6	7
F_1	1	*1*	*1*	1	.	.	.	.
F_2	2	*1*	*1*	.	1	.	.	.
F_3	3	.	.	.	.	1	.	1
G_1	1001	1	1	.	.	.	1	1
H_1	2001	.	1	.	1	.	.	.
H_2	2002	.	.	.	.	1	1	.

an interval, and all of the cubes in that expression are labeled by numbers from that interval. The reason for using this strategy is to ensure that all of the processors will create the same *cube-literal* matrix irrespective of the order in which information for different expressions is filled up.

The *cube-literal matrix* is created by using the following method. Each row of the *cube-literal matrix* corresponds to a cube in an expression in the network, and the row is labeled by the cube number of the cube. Each column of the matrix corresponds to a *unique* literal, and it is labeled by the *node-id* of that literal. The position (i, j) in the *cube-literal matrix* is set to a 1 if cube i contains literal j. A rectangle in the cube literal matrix identifies a cube which can be extracted from the network [43]. The columns of the rectangle identify the literals in the common-cube, and the rows identify the cubes (and expressions) which will be affected if that common-cube is extracted.

For example, consider the set of expressions in Equation (3.3). Let us assume that we have assigned the intervals [1–1000], [1001–2000] and [2001–3000] to the nodes F, G and H, respectively. The *cube-literal matrix* for those expression is shown in Table 3.4.

After the *cube-literal matrix* is created, the *master* processor creates a *cube_extract_object* to generate all possible prime rectangles and find the maximum-valued rectangle. The algorithm is almost identical to that of kernel extraction. Hence, we are omitting the details for the sake of conciseness. For example, one prime rectangle is ({1, 2}, {1, 2}) (shown in italics) corresponding to the extracted cube ab.

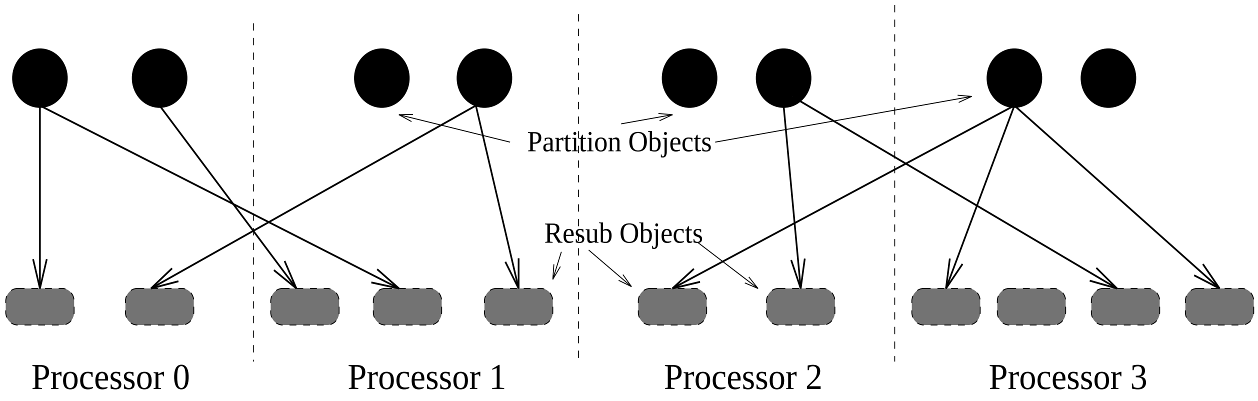

Figure 3.6 Creation of objects for resubstitution

After the root *cube_extract_object* finds the maximum-valued rectangle, it reports that to the *master* processor. The *master* processor extracts the common cube and determines the set of expressions that this common cube will divide. It then broadcasts that information to all of the processors. All of the affected cubes are removed from the *cube-literal matrix* and the new cubes are entered. The *version numbers* of the affected nodes are incremented by 1. The *master* processor then creates a *cube_extract_object* to search for more cube extraction. The master processor does not wait for all of the processors to finish the update for creating the object. But the priority of the *cube_extract_object* will be lower than that of the update message.

3.4.3 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. The *partition_objects* creates the *resub_objects* for each node in that partition. These objects are distributed across all of the processors by the CHARM run-time system as shown in Figure 3.6.

The responsibility of *resub_objects*(η) is to search for possible resubstitution by other nodes in the node η. We will restrict our search 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, the *resub_objects*(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, it requests for permission from the *master* processor. In the request, it supplies the *version numbers* of the divisors ν and η.

When the *master* processor receives that request, it checks if the *version numbers* mentioned in the request are the same as the current *version numbers*. If so, it performs the resubstitution and increments the *version number* of the resubstituted node η. It then broadcasts the information to all other processors. Since the functionality of the node η has changed, the *master* processor creates a new *resub_object* for the node η. The resubstitution procedure is very similar to the gate substitution procedure of the parallel Transduction algorithm described in Chapter 2.

3.5 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. The sum-of-products form of each expression in a Boolean network can be minimized. Two-level minimization can be made more powerful in the context of a multilevel network by providing the minimizer with various don't-care sets derived from the structure of the network. Those don't-care sets are defined in Section 3.2.1. The size of the don't-care sets determines how thorough and how fast the minimization process will be.

In a sequential algorithm for simplification, only one node is simplified at one time. As a result, the *satisfiability* don't-care set of any other node can be used in the simplification process, and the result will still be correct. Similarly, the *fan-out* don't-care sets can also be used without any constraint (except that the simplification process may become very time-consuming) because only one node in the network can change its function at one time.

We want to perform the node simplifications in parallel. In that case, many nodes may be simplified concurrently, and there will be some constraints on which don't cares can be used to ensure the result to be correct. For example, consider the set of expressions

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

If both of the nodes are simplified concurrently and they use each other's satisfiability don't-care sets, then the result will 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 if two nodes are being simplified concurrently, they should not use each other's satisfiability don't-care sets.

To make concurrent node simplification possible, we 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.

3.5.1 Use of satisfiability don't care

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. But if a large don't-care set is supplied to the two-level minimizer, the simplification process will be very expensive. Hence, some don't care filters have been proposed in [44]. One popular filter is the *subset support filter*, where the satisfiability don't care of only those nodes is considered whose *support* is a subset of the *support* η, where η is the node to be minimized. Another popular heuristic is to collect transitive fan-in nodes of η up to i_i level and their transitive fan-outs up to o_i level, except those nodes which are in the transitive fan-out of η, and use their satisfiability don't-care sets in the minimization of η, where i_i and o_1 are user-specified parameters.

Given the heuristic/filter to be used for the use of satisfiability don't-care set, one can compute a set of nodes for every node η in the network whose satisfiability don't-care sets will be used for the simplification of η. From that information, we can generate a directed graph, called the Satisfiability Don't-care Graph (SDG), denoted by (V, A). The vertex set of that graph consists of the nodes in the network. There exists a directed arc $(v_i, v_j) \in A$, v_i and $v_j \in A$, if the satisfiability don't-care set of the node v_j will be used in the simplification of the node v_i. An example of SDG is shown in Figure 3.7(a).

Theorem 1 *A set of nodes Γ in a Boolean network can be simplified concurrently without any other constraint if they do not form any cycle in the corresponding SDG and if only a satisfiability don't-care set is used to simplify those nodes.*

Proof: Since the set of nodes Γ does not form any cycle in SDG, the subgraph induced by them will be a directed acyclic graph. Let the size of that set Γ be n. It will always be possible to label the nodes in Γ from 1 to n in such a fashion that label(v_k) > label(v_i) if there is a directed path from v_i to v_k in the induced subgraph of SDG by the nodes in Γ. This numbering can be done by breadth-first search. Now, the concurrent simplification of the nodes in Γ is equivalent to the simplification of those nodes sequentially, starting from the node with label 1 and ending at the node with label n, using satisfiability don't-care sets with the constraint that during simplification of a node with label i, the satisfiability don't-care set of a node in Γ is *not* used if it has a label less than i (SDG may change when a node is simplified because the structure of the circuit may change, hence the new SDG may have an edge from the node with label i to another node with a lower label). The sequential algorithm is correct, because it is changing one node a time. Hence, the concurrent simplification of the nodes in Γ is also correct. ■

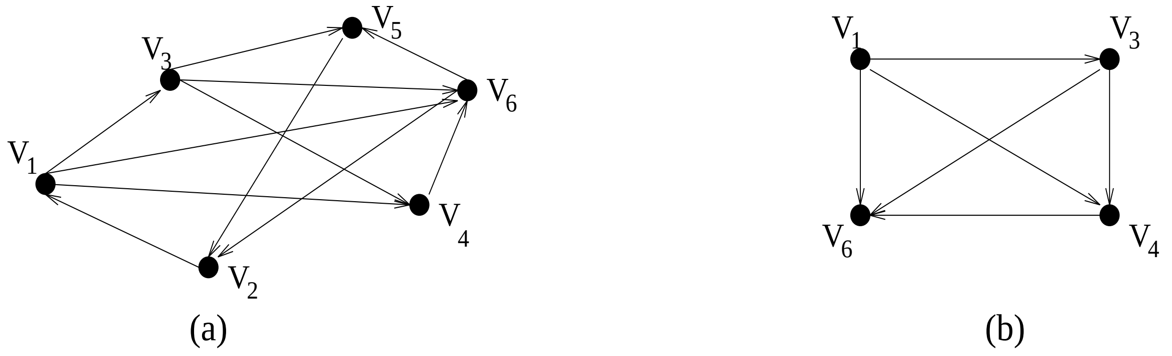

Figure 3.7 (a) An example of SDG, (b) largest acyclic vertex set for that SDG

We want to perform the simplification of the nodes in parallel. Hence, we would like to find the largest set of nodes such that simplification of those nodes can be performed concurrently. That problem can be reduced to finding the largest set of nodes in the corresponding SDG such that they do not form any cycle among themselves. We name this problem as the "largest acyclic vertex set" problem.

Theorem 2 *The "largest acyclic vertex set" problem is an NP-hard problem.*

Proof: A known NP-complete problem, *feedback vertex set* problem [45], can be reduced to the corresponding decision problem of the "largest acyclic vertex set" problem. The problem instance of *feedback vertex set* problem is a directed graph $G = (V, A)$ and a positive integer $K \leq \|V\|$. The problem statement is as follows: Is there a subset $V' \subseteq V$ with $\|V'\| \leq K$ such that V' contains at least one vertex from every directed cycle in G? It is clear that if such a set is found, then the rest of the nodes will form an acyclic vertex set. Hence, the *feedback vertex set* problem can be reduced to the following problem: Is there a subset $V'' \subseteq V$ with $\|V''\| \geq K$ such that the nodes in V'' will not have any cycle among themselves? This new problem is the decision problem corresponding to the "largest acyclic vertex set" problem and it is in NP. Thus the "largest acyclic vertex set" problem is an NP-hard problem. ∎

This shows that it is computationally untractable to determine the largest vertex set that will form an acyclic set. Hence, we have to resort to a heuristic method to solve that problem. But the heuristic method does not guarantee that the vertex set will be the largest set.

In most practical circuits, the size of the *largest vertex set* will be small. As a result, the number of nodes that can be simplified concurrently will be small. But, if we are willing to use a smaller don't-care set for some nodes, we may be able to find a larger set which can be simplified concurrently. This is equivalent to removing some directed edges from the SDG. On the other hand, we do not want to remove too many edges from the SDG because that may degrade the quality of the synthesized circuit.

Let us consider the following problem. We have the SDG, $G = (V, A)$. We want to find the largest set of nodes in SDG such that they do not form any cycle among themselves after at most p edges are removed, where p is a user-specified parameter. We call this problem the "minimum cyclic vertex set" problem.

Theorem 3 *The "minimum cyclic vertex set" problem is an NP-hard problem.*

Proof: We can reduce the "largest cyclic vertex set" problem to this problem by setting p to 0. Since the "largest cyclic vertex set" problem is proved to be an NP-hard problem, the "minimum cyclic vertex set" is also an NP-hard problem. ∎

Since that problem is also computationally intractable, we plan to use a heuristic method, described below.

3.5.2 Procedure for parallel simplification

At the beginning of the program execution, the user chooses the following parameters for simplification: a) don't-care type (whether satisfiability don't care or fan-out don't care or both to be used), b) don't care size (the heuristic to choose how much of satisfiability don't care to be used) and c) the don't-care filter type.

Each *partition_object* creates a *simp_object* for each node in that partition. The *simp_object*(η) is responsible for simplification of the node η. Those simplification objects are distributed across different processors by the CHARM run-time system, as shown in Figure 3.8.

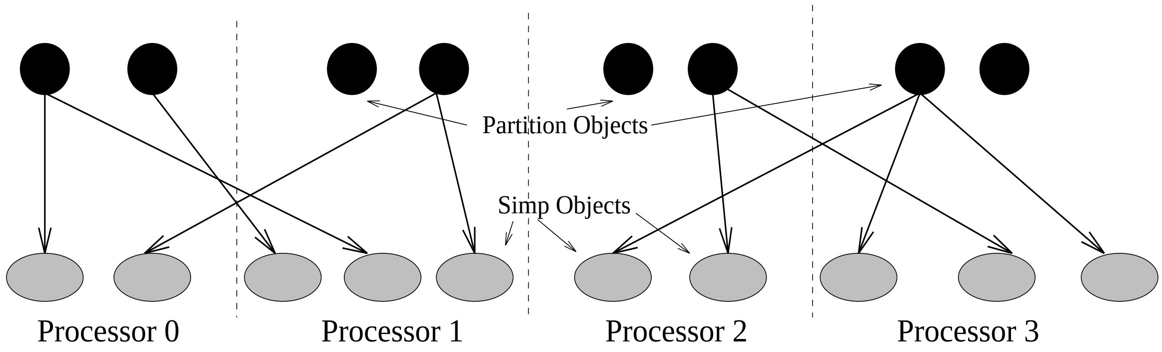

Figure 3.8 Creation of objects for simplification

Whenever a *simp_object*(η) is awakened by a message, it asks for permission from the *master* processor for the simplification of η. Initially all of the nodes are unlocked. The *master* processor picks up the requests for permission one by one. When it picks up the request for simplification for the node η, it checks if η is *Boolean locked*. If it is, the *master* processor denies permission and ask the *simp_object*(η) to try later. Otherwise, if the don't-care type chosen by the user includes the fan-out don't care, it checks the fan-outs of η to see if any one of them is *algebraic locked*. If so, the *master* processor denies permission and ask the *simp_object*(η) to try later (as shown for the node e in Figure 3.9). Otherwise, it *Boolean locks* all of the fan-outs of η (as shown for the node g in Figure 3.9). Then it checks if the satisfiability don't-care set is chosen in the don't-care type. If it is not chosen, the permission is sent to *simp_object*(η) and η is *algebraic locked*. If it is chosen, it computes the list, *satis_DC_list*, of nodes whose satisfiability don't-care sets should be used for the simplification of η using the heuristic technique chosen by the user. From the *satis_DC_list*, it removes all of the nodes which are *algebraically locked* that time. It then sends the permission along with the *satis_DC_list* to the *simp_object*(η) and it *algebraically lock* the node η.

When the *simp_object*(η) receives permission to execute, it starts the simplification of η. It computes the fan-out don't-care set if it is chosen in the don't-care type to be used. It then computes the satisfiability don't cares of the nodes in *satis_DC_list*, which is supplied by the *master* processor if the don't-care type chosen by the user includes the satisfiability don't care. It then performs two-level minimization using ESPRESSO [14]

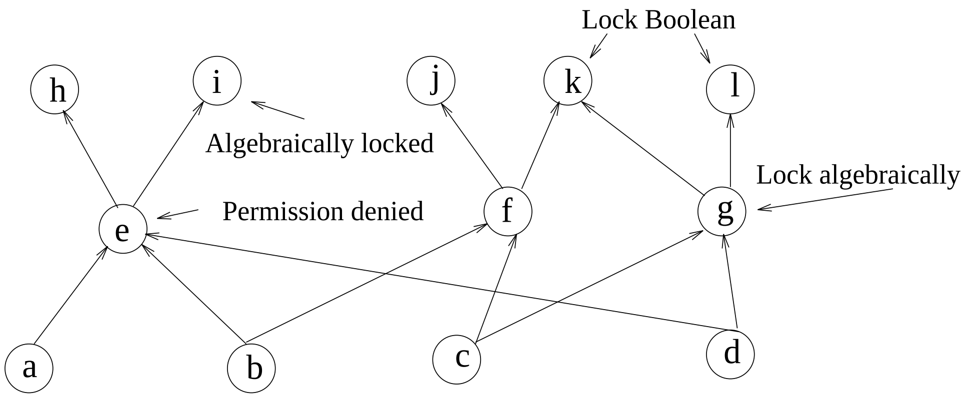

Figure 3.9 "Algebraic" and "Boolean" locking of nodes during simplification

and the reduce offset technique [46]. After the minimization is completed, it informs all of the processors about the new functionality of η.

All of the processors update the functionality of η when they receive the result from the *simp_object*(η). The *master* processor does some other tasks in addition. It unlocks the *algebraic lock* of η. Also, if the fan-in don't-care set was used, it unlocks the *Boolean locks* of the fan-outs of η. The parallel algorithm is shown in Figure 3.10.

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. The next node cannot use the satisfiability don't-care set of the first node if it asks permission before the simplification of the first node is completed. Hence, for better simplification, we have assigned different priorities to different *simp_objects*. The priorities were assigned proportional to the size of the nodes (in terms of the literal count). If two *simp_objects* ask permission from the *master* processor simultaneously, the one with higher priority will be picked up first.

At any time, a processor p will have a list of messages to be processed. Some of them will be the permission from the *master* processor for simplification for some node, some will be new *simp_object* creations (also other synthesis objects as well). These messages will be sorted according to decreasing value of the priorities, and the highest priority message will be picked up for processing. Hence, it will process simplification of a high-priority node (after it receives the permission) before creation (and asking permission)

```
Simp_Object(η)
(1)   /* Object is awakened */
      send request for permission to master proc.;
(2)   /* Object receives permission/denial back */
      if(permission denied)
           lowers priority and requests perm.  again;
      else
      D = ∅;
      if(fan-out don't care to be used)
           D = D ⋃ fan-out don't-care set;
      if(satisfiability don't care to be used)
           for node ν in satis_DC_list
             D = D ⋃ satisfiability don't care of ν
      η = espresso(η, D);
      if(any improvement)
           report new expression of η to all proc.
      else
         inform master proc.  to unlock nodes etc.;


Master Processor Operations
(1)   /* receive request for permission for η */
      if (η is Boolean locked)
           send denial to simp_object(η)
      if(fan-out DC to be used)
           if(any fan-out(η) algebraic locked)
             send denial to simp_object(η)
           else
             Boolean lock fan-outs of η
      if(satisfiability DC to be used)
           compute satis_DC_list using given heuristics;
           remove algebraic locked nodes from satis_DC_list;
      send permission and satis_DC_list to simp_object(η);
      algebraic lock η
(2)   /* receives info after simp. of η */
      if(fan-out DC to be used)
         Boolean unlock fan-outs of η
      algebraic unlock η;
      update expression of η
```

Figure 3.10 Parallel algorithm for node simplification of a low-priority node. As a result, the low priority will not prematurely ask permission for simplification. When the message for creation for a low-priority node is picked up, the synthesis of some high-priority nodes are already completed. Hence, the satisfiability don't-care sets of those nodes will be available for use. That can potentially improve the quality of node simplification.

3.6 Overview of the ProperMIS Algorithm

In the previous sections, we have described the parallel algorithm for each individual logic transformation. In this section, we will describe how the entire system works with all of the transformations.

In Section 3.3, we have described how *partition_objects* are created for each partition. Those objects could have been automatically distributed by the CHARM run-time system. But the CHARM system assumes all of the objects to be of equal size when it distributes the objects for load balancing. But in this case, all of the nodes in the circuit may not be of same size. That may lead to load imbalance and bad speedup.

Therefore, we have decided to distribute the objects to different processors in such a fashion that all of the processors are equally loaded. We assume that the *literal count* (number of literals in the sum-of-products form) of a node is a good measure of the complexity of that node. Hence, a good estimate of the complexity of a partition will be the literal count of that partition, which is the sum of the literal counts of all of the nodes in that partition. We first compute the literal count of all of the partitions. Then we sort the partitions according to decreasing values of the literal count.

We initialize the loads in all of the processors to be zero. We then put the partition objects one by one in different processors, starting at the beginning of the sorted array. Every time we want to create an object for a partition π, we search for the processor with the lightest load till then and put the object in that processor. Also, the literal count of that partition π is added to the load of that processor.

After the circuit is read, the partitions are formed and the *partition_objects* are created. The partition objects create the *simp_objects*. Then the simplification process goes on as described in Section 3.5.

When all of the nodes in a partition π are simplified, the *partition_object*(π) is sent a message to generate kernels for the nodes in π. All of the processors are sent messages to fill up the *cube-literal matrix* with the cubes of the nodes in π. Also, *partition_object*(π) is sent a message to create the *resub_objects* for the nodes in π. We assign priorities to different transformations. The initial priorities are ordered as (highest to lowest): simplification, kernel extraction, cube extraction and resubstitution. A processor p will pick up a message of priority pr and process that request if there is no message with priority higher than pr waiting to be processed. For example, a processor will not pick up any message regarding kernel generation or kernel extraction if some other message regarding simplification is waiting to be processed in that processor.

After the *master* processor receives kernels of at least Δ% of the partitions, it creates a *kernel_extract_object*, as described in Section 3.4; (Δ equal to 100 means the *master* processor creates the *kernel_extract_object* after kernels for *all* of the partitions are generated). Similarly, the *master* processor creates a *cube_extract_object* after it fills up cubes of the nodes of at least Δ% of the partitions.

Whenever the *master* processor performs any logic transformation, it invalidates the kernels and the cubes of the affected nodes in the *co-kernel cube matrix* and *cube-literal matrix*, respectively, if the data are present. All of the processors do the same when they receive the instruction from the *master* processor. Then messages are sent to the corresponding *partition_objects* to regenerate the kernels of the affected nodes. Similarly, messages are sent to reinsert the cubes of the affected nodes in the *cube-literal matrix* in all of the processors.

From the above description, it is clear that unlike the MIS algorithm, we do not have any separate phase for regeneration of kernels for the affected nodes. The regeneration of the kernels of the affected nodes is done *asynchronously*. The same thing is true for cube extraction.

After one kernel extraction, another *kernel_extract_object* is created immediately, hence we do not wait for the regeneration of kernels of the affected nodes. Therefore, we do not have any synchronizing barrier at every step of the kernel extraction process. The *kernel_extract_object* has lower priority than the kernel generation messages. It is possible that the regenerated kernels may arrive at the middle of the kernel extraction, making some generated rectangles in the kernel extraction algorithm invalid. Hence, the computed maximum-valued rectangle may not be the maximum-valued rectangle in the modified *co-kernel cube matrix*. But we go ahead with that rectangle because computing the maximum-valued rectangle at every step to find a rectangular cover is a heuristic method and compromising the maximum-value property at some stages of the covering process does not necessarily compromise quality. The cube extraction algorithm is also implemented similarly.

If we keep the priority of kernel extraction always higher than that of cube extraction, the system will tend to perform almost all of the kernel extractions before any cube extraction. That may lead to local minima. Thus, it is advised in [38] to use kernel extraction and cube extraction in an interleaved fashion. First choose a high *threshold* (e.g., 30), and perform first kernel extraction and then cube extraction in such a fashion that a factor is extracted if the gain in literal count is more than the *threshold*. Then, reduce the *threshold* (e.g., to 10) and repeat the process. Then, reduce the *threshold* to 0 and repeat the process again.

We emulated that method by using priority allocation as shown in Figure 3.11. We kept two thresholds, *kernel_threshold* and *cube_threshold*. Initially, both of them are set to high values (e.g., 30) and the priority of kernel extraction is higher than that of cube extraction (time $T1$ in Figure 3.11). Hence, initially, mostly kernel extractions will be performed. Whenever the gain from kernel extraction reaches below the *kernel_threshold*, we reduce the priority of the kernel extraction process below the cube extraction process and reduce the *kernel_threshold* to a lower value (e.g., 10). Now, mostly cube extractions will be performed as they are of higher priority (time $T2$ in Figure 3.11). Whenever the gain from cube extraction reaches below the *cube_threshold*, the priority of the cube

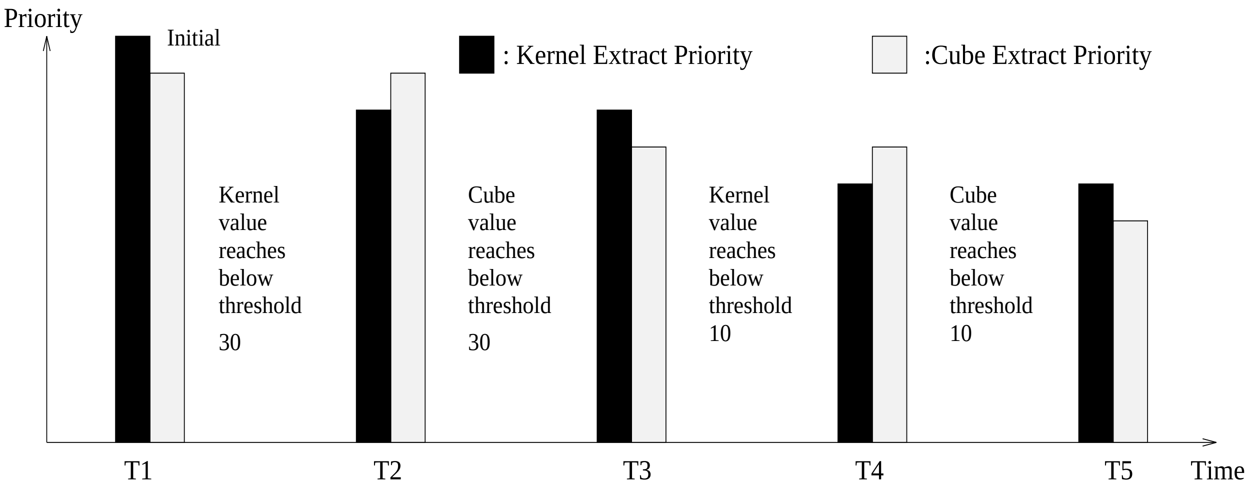

Figure 3.11 An example of priority adjustments of kernel and cube extraction

extraction process is reduced such that it goes below the priority of the kernel extraction process and reduces the *cube_threshold* to a lower value (e.g., 10). Now, again mainly kernel extractions will be performed as the kernel extraction is of higher priority (time $T3$ in Figure 3.11). This procedure continues until both of the thresholds reach 0. Note that this procedure accomplishes the goal of interleaving kernel and cube extraction *without* the use of synchronizing barriers in the parallel algorithm.

3.7 Experimental Results

The parallel algorithm described in the previous sections has been implemented as a part of ProperCAD project on top of the CHARM run-time system. In this section, we will describe the results obtained by applying ProperMIS on various *MCNC* and *ISCAS* benchmark circuits.

3.7.1 Results with algebraic factoring and simplification

In Table 3.5, we compare the quality of circuits obtained by MIS-II with that of the ProperMIS algorithm in a uniprocessor environment. To run MIS-II, we have used mainly the script given in [38]. Sometimes, for smaller circuits, we have used don't-care sets larger than those used in the given script [38] to obtain better quality. For example, the circuit *bw* is run with the satisfiability don't-care sets of all of the nodes, instead of using only the satisfiability don't-care sets of the nodes with subset support. But, the parameters 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. One can observe that the quality of the circuits produced by *ProperMIS* are almost the same (sometimes better) as those produced by *MIS-II*. The last two columns compare the run times for *MIS-II* and *ProperMIS* for a SUN4 workstation (run times are given in seconds). Again, one can observe that the run times are comparable. The run times and qualities are not identical due the nondeterminism in the order of applying various transformations.

We would like to reiterate the fact that to run ProperMIS in different machines, *no change* in the parallel program is necessary. One has to move the program to those machines, and then compile and link to the necessary libraries, and the parallel program will then be ready to run. Table 3.6 presents the results obtained on an Encore Multimax machine. It is a *shared memory MIMD* machine with 8 processors. For different numbers of processors, the quality of the synthesized circuits in terms of the literal count and the run times in seconds (speedup result in parenthesis) are shown for some benchmark circuits. Table 3.7 presents the similar results obtained on an Intel/860 machine. It is a *distributed memory MIMD* machine with 8 processors connected by a hypercube configuration. Table 3.8 presents the similar results on a network of workstations environment. This behaves as a *distributed processing* system. There is a *host* workstation, which distributes work to other workstations. The workstations used in this experiments are SUN4 workstations with a SPARC 1 processor in them. ProperMIS runs on other parallel machines such as Sequent Symmetry (shared memory MIMD), but the results are omitted due to lack of space.

From the tables it is clear that the ProperMIS algorithm produces excellent speedups and maintains quality comparable to that of the MIS approach. Some of the superlinear speedups are due to the search anomalies in parallel search techniques. One can observe

Table 3.5 Comparison of quality and run time between MIS-II and ProperMIS

CKT	Literal Count			Run Time (sec)	
	Initial	MIS-II	ProperMIS	MIS-II	ProperMIS
bw	452	174	172	99.47	105.53
duke2	1775	456	466	90.50	110.40
misex3	1675	520	525	169.30	196.26
misex3c	1708	531	519	126.50	178.49
x3	1816	832	850	76.90	79.95
dalu	3588	1306	1277	248.90	281.67
pair	2673	1878	1874	82.20	86.78
frg2	2855	807	797	163.80	179.64
i8	4626	1145	1150	174.60	184.43
i9	1453	623	614	50.50	52.71
C499	616	598	598	58.10	67.25
C5315	4386	2009	1975	1540.50	2069.74
des	7412	4024	4153	2784.00	3215.00

that the maximum degradation of quality with multiple processors is less that 3%. The quality is sometimes improved with a larger number of processors. Again, that happens due to the nondeterminism.

3.7.2 Effect of Δ on runtime and quality

In Section 3.6, we reported that the kernel extraction process starts after Δ% of the kernels are generated, where Δ is a user-specified parameter. Similarly, the cube extraction process starts after Δ% of the cubes are filled in the *cube-literal matrix.* In the last subsection, all of the results presented were with Δ equal to 100.

In Figure 3.12, we present the results obtained by applying ProperMIS on the benchmark circuit *duke2* with different values of Δ. Figure 3.13 presents the results obtained by performing the same experiment on the benchmark circuit *dalu.* As expected, the quality of the synthesized circuit is best with the value of Δ close to or equal to 100. If the kernel extraction process starts after all of the kernels are generated, it is expected to produce better quality circuits. It was expected to produce smaller run-time with smaller

Table 3.6 Run times (speedups) and circuit qualities in Encore Multimax (shared memory MIMD)

CKT	1 Processor		2 Processors		4 Processors		8 Processors	
	Lit	Time(spd)	Lit	Time(spd)	Lit	Time(spd)	Lit	Time(spd)
bw	172	166.29(1.0)	182	117.27(1.4)	181	35.88(4.6)	181	22.75(7.3)
duke2	459	262.15(1.0)	467	177.04(1.5)	488	94.34(2.8)	475	62.40(4.2)
misex3	517	554.08(1.0)	524	503.60(1.1)	524	181.17(3.1)	523	122.33(4.5)
misex3c	520	483.46(1.0)	537	424.66(1.2)	524	204.12(2.4)	525	150.71(3.2)
x3	849	250.18(1.0)	862	107.71(2.3)	860	73.52(3.4)	856	46.19(5.4)
dalu	1278	571.77(1.0)	1279	431.74(1.3)	1296	227.86(2.5)	1297	119.51(4.8)
pair	1874	208.41(1.0)	1880	123.39(1.7)	1879	88.81(2.3)	1880	60.76(3.4)
frg2	798	284.98(1.0)	810	221.15(1.3)	815	130.84(2.2)	823	60.19(4.7)
i8	1150	295.93(1.0)	1161	206.60(1.4)	1149	97.05(3.1)	1159	65.89(4.5)
i9	614	54.30(1.0)	618	32.25(1.7)	616	21.11(2.6)	1159	15.62(3.9)
C499	598	136.74(1.0)	598	81.52(1.7)	599	25.97(5.3)	599	16.77(8.2)
C5315	1975	4300.29(1.0)	2020	1359.83(3.2)	2027	795.94(5.4)	2025	557.21(7.7)
des	4153	7727.84(1.0)	4034	3676.37(2.1)	4055	2056.19(3.8)	-	-

Table 3.7 Run times (speedups) and circuit qualities in Intel/860 hypercube (distributed memory MIMD)

CKT	1 Processor		2 Processors		4 Processors		8 Processors	
	Lit	Time(spd)	Lit	Time(spd)	Lit	Time(spd)	Lit	Time(spd)
bw	172	34.62(1.0)	182	15.21(2.3)	182	6.82(5.1)	183	3.97(8.7)
duke2	459	66.32(1.0)	465	50.78(1.3)	463	25.30(2.6)	464	18.12(3.7)
misex3	511	102.81(1.0)	528	74.10(1.4)	521	40.09(2.6)	520	21.44(4.8)
misex3c	520	117.64(1.0)	535	97.54(1.2)	515	45.35(2.6)	514	38.77(3.1)
x3	850	58.64(1.0)	850	36.68(1.6)	855	25.96(2.3)	855	18.90(3.1)
dalu	1277	131.39(1.0)	1292	89.10(1.6)	1284	50.31(2.6)	1286	32.22(4.1)
pair	1874	63.13(1.0)	1876	45.92(1.4)	1877	30.37(2.1)	1886	21.69(3.0)
frg2	801	82.59(1.0)	816	50.74(1.6)	827	40.67(2.1)	825	27.21(3.1)
i8	1151	80.55(1.0)	1162	51.80(1.6)	1158	34.98(2.3)	1162	20.42(4.0)
term1	206	41.75(1.0)	211	34.56(1.2)	216	25.84(1.6)	216	10.32(4.0)
C499	598	32.76(1.0)	598	13.61(2.4)	599	6.52(5.0)	599	6.22(5.3)
C5315	1975	1433.42(1.0)	2023	1108.35(1.3)	2027	678.41(2.1)	2039	196.63(7.3)

Table 3.8 Run times (speedups) and circuit qualities in network of workstations (distributed processing)

CKT	1 Processor		2 Processors		4 Processors		6 Processors	
	Lit	Time(spd)	Lit	Time(spd)	Lit	Time(spd)	Lit	Time(spd)
bw	172	54.42(1.0)	182	29.05(1.8)	182	15.62(3.4)	183	11.69(4.6)
duke2	467	91.73(1.0)	478	63.05(1.4)	481	38.43(2.4)	472	30.21(3.1)
misex3	523	132.58(1.0)	523	76.12(1.7)	516	39.31(3.4)	526	36.50(3.6)
misex3c	524	167.24(1.0)	527	117.66(1.4)	517	61.74(2.7)	519	56.27(3.0)
x3	838	56.49(1.0)	838	35.91(1.5)	853	22.29(2.5)	851	21.78(2.6)
dalu	1275	185.72(1.0)	1287	146.22(1.3)	1289	77.19(2.4)	1290	61.13(3.1)
pair	1874	67.15(1.0)	1881	52,26(1.3)	1884	33.26(2.0)	1884	30.42(2.2)
frg2	801	103.91(1.0)	819	65.07(1.6)	820	51.40(2.0)	825	36.73(2.8)
i8	1151	111.33(1.0)	1165	71.78(1.5)	1140	50.25(2.2)	1141	40.44(2.8)
i9	614	22.77(1.0)	618	13.99(1.6)	618	10.53(2.2)	618	9.31(2.4)
C499	598	55.62(1.0)	598	35.12(1.6)	599	21.14(2.6)	599	13.79(4.0)

Δ because the kernel extraction process did not need to wait until all of the kernels were generated. Contrary to the expectation, the run time also was the lowest with the value of Δ in the vicinity of 100. This is due the fact that if the kernel extraction process starts before the completion of all kernel generation, it does not produce optimal factoring and the search process requires a larger number of iterations to converge to the final result.

The results obtained on other benchmark circuits by running this experiment show similar trends. Hence, it is advisable to set the value of Δ to a number very close to or equal to 100 as was done when obtaining results for Tables 3.6, 3.7 and 3.8.

3.7.3 Results with simplification alone

In this and the next subsection, we report on how effectively we have parallelized each of the phases in logic synthesis, namely, simplification and factoring. In this subsection, we will present the results obtained by performing node simplification only; here, algebraic factoring was not performed. In Table 3.9, we present the results obtained by running different benchmark circuits on Encore Multimax (shared memory MIMD), and

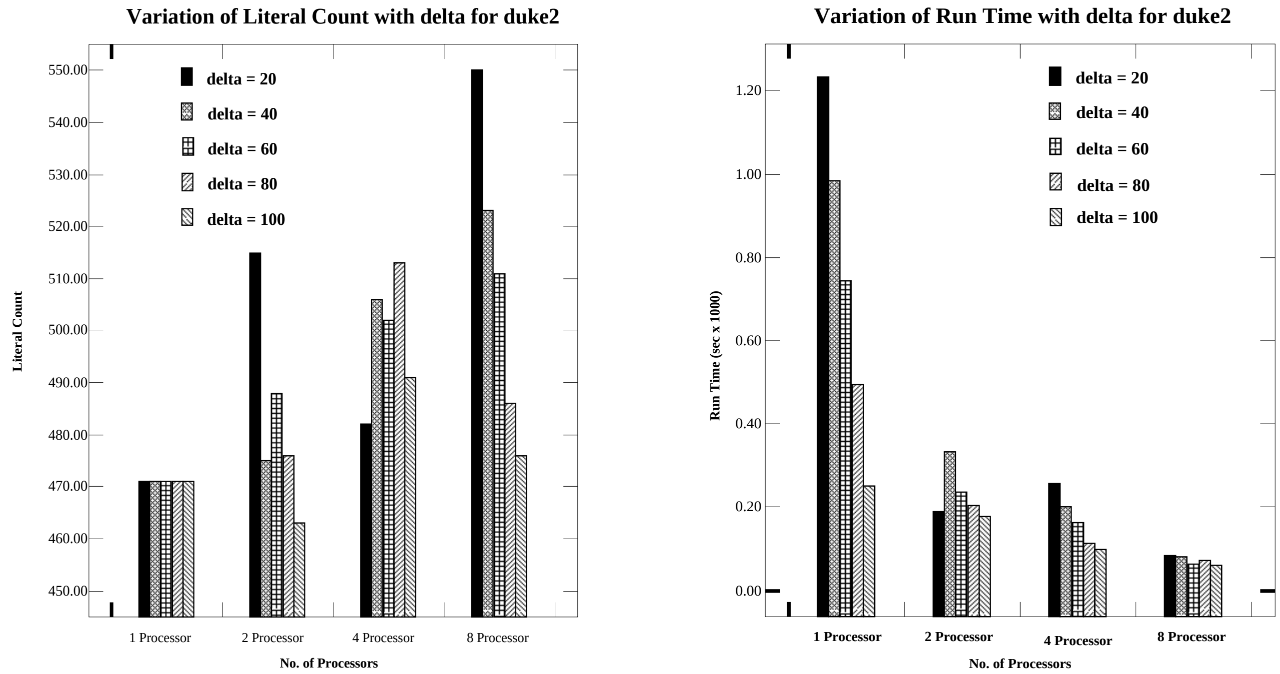

Figure 3.12 Variation of quality and run-time with gamma for *duke2*

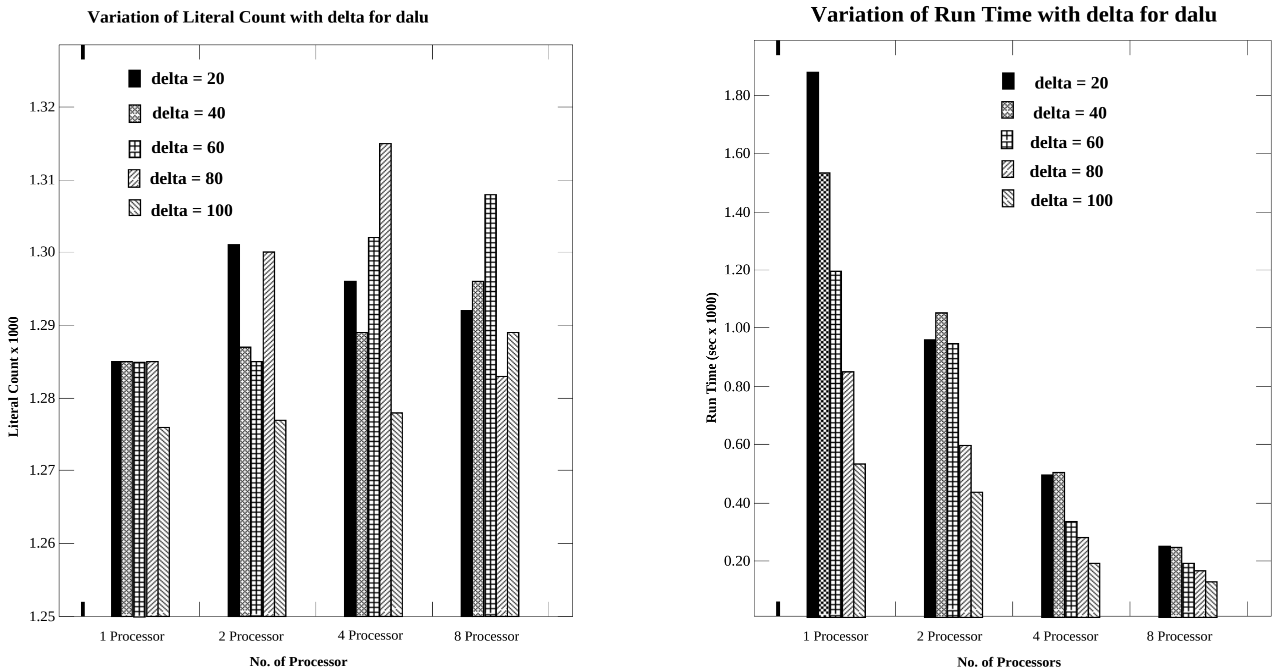

Figure 3.13 Variation of quality and run time with gamma for *dalu*

Table 3.9 Quality and run time for different circuits in Encore Multimax with only node simplification

CKT	1 Processor		2 Processors		4 Processors		8 Processors	
	Time	Lit	Time	Lit	Time	Lit	Time	Lit
bw	163.12	181	111.47	197	38.07	195	19.05	196
duke2	168.80	996	122.77	1043	55.86	1089	44.50	1059
misex3	502.18	702	467.87	723	173.35	706	102.28	745
misex3c	440.16	718	364.23	919	174.78	720	217.65	916
x3	163.65	1397	61.95	1410	42.61	1430	39.58	1409
C499	126.13	608	53.99	608	23.09	608	14.45	609
C5315	3542.19	2289	1210.09	2389	812.58	2323	365.11	2384
dalu	432.17	2738	353.27	2755	136.61	2757	62.83	2757
pair	56.44	2116	41.46	2120	31.06	2116	30.99	2120
frg2	216.70	1441	181.42	1527	98.62	1491	50.45	1463
i8	207.84	2368	147.07	2397	57.09	2386	55.04	2366
i9	34.74	794	20.37	806	14.33	797	12.95	797

in Table 3.10 we present the same results on Intel/860 hypercube (distributed memory MIMD). One can observe that the speedup obtained is usually better than that obtained in Section 3.7.1 because node simplification is inherently more parallel that the algebraic factoring method. But one can also observe that there is a larger loss in quality for some circuits as we employ a higher number of processors. Usually the algebraic factoring phase compensates for the losses which occurred in the simplification phase due to the use of multiple processors. But in this experiment, we do not have any algebraic factoring phase. Hence, in some circuits, the loss in quality is greater.

3.7.4 Results with algebraic factoring alone

In this subsection, we will present results obtained by performing algebraic factoring only; here, node simplification was not performed. Table 3.11 shows the results obtained in Encore Multimax and Table 3.12 shows the same results in Intel/860 hypercube. For different numbers of processors, we have presented the run time and the quality of the circuit obtained for some benchmark circuits.

Table 3.10 Quality and run time for different circuits in Intel/860 with only node simplification

CKT	1 Processor		2 Processors		4 Processors		8 Processors	
	Time	Lit	Time	Lit	Time	Lit	Time	Lit
bw	33.59	181	32.03	201	5.53	193	2.72	193
duke2	31.80	996	23.90	984	10.54	1042	5.55	1102
misex3	79.79	701	57.61	717	33.31	717	18.66	739
misex3c	98.85	718	79.76	920	36.95	720	36.11	719
x3	19.55	1408	8.48	1416	5.98	1416	5.81	1416
C499	29.90	608	12.05	608	4.40	608	2.38	608
C5315	619.93	2289	244.79	2378	109.13	2359	78.29	2386
dalu	95.30	2738	50.33	2753	26.35	2758	15.53	2737
pair	16.08	2116	10.72	2122	9.73	2116	8.12	2116
term1	39.90	527	21.02	543	15.66	543	9.57	543
frg2	54.44	1426	28.06	1491	21.38	1500	12.19	1450
i8	54.74	2367	30.55	2382	19.35	2366	17.54	2371
i9	10.78	794	6.91	797	5.51	797	5.46	797

One can observe that the speedups obtained for most of the circuits are not as great as those obtained with node simplification. This is due to the fact that the amount of parallelism in the algebraic factoring procedure is not as large as that in the node simplification procedure. Also, the communication-to-computation ratio in the algebraic factoring procedure is much higher than that in the node simplification procedure. That degrades the speedup in Intel/860 even more because the communication is much more expensive there. But when we perform both node simplification and algebraic factoring, we obtain better results as shown earlier because we can overlap communication and computation.

3.7.5 Effect of priority assignment on quality and speedup

In this chapter, we have discussed how we have assigned the priorities of different transformations. We have also described how we dynamically adjusted the priorities of

Table 3.11 Quality and run time for different circuits in Encore Multimax with only algebraic factoring

CKT	1 Processor		2 Processors		4 Processors		8 Processors	
	Time	Lit	Time	Lit	Time	Lit	Time	Lit
bw	13.86	215	10.09	215	7.93	225	6.53	220
duke2	187.89	420	113.05	424	70.52	420	53.30	419
misex3	289.62	650	157.85	643	96.04	649	67.61	647
x3	710.08	1079	438.29	1082	255.13	1104	202.41	1113
C499	7.47	598	4.57	599	3.01	599	2.44	599
C5315	471.70	2201	314.28	2222	252.03	2238	219.61	2235
dalu	892.58	1770	520.41	1808	429.01	1825	369.74	1823
pair	336.22	1998	175.89	2004	117.64	2018	102.16	2019
i8	234.32	1230	135.86	1221	94.06	1241	75.38	1238
i9	33.67	637	21.58	639	14.84	643	12.55	641

Table 3.12 Quality and run time for different circuits in Intel/860 with only algebraic factoring

CKT	1 Processor		2 Processors		4 Processors		8 Processors	
	Time	Lit	Time	Lit	Time	Lit	Time	Lit
bw	11.15	216	4.66	217	3.69	221	3.54	215
duke2	64.30	420	45.73	428	36.67	426	31.22	422
misex3	98.90	644	62.19	645	48.24	650	32.16	641
misex3c	1103.41	669	851.22	694	514.02	672	432.92	650
x3	158.46	1079	120.89	1073	106.11	1066	102.11	1057
C5315	110.01	2201	74.45	2204	62.18	2205	52.36	2206
dalu	179.45	1770	133.66	1741	123.84	1704	127.66	1697
pair	78.87	1998	53.28	2022	39.10	2021	32.65	2027
term1	56.62	352	53.85	340	47.08	333	45.13	324
frg2	589.83	1380	552.72	1271	575.81	1388	540.14	1428
i8	68.90	1230	53.72	1228	41.00	1241	39.35	1229
i9	11.29	637	8.81	637	6.43	637	5.39	639

Table 3.13 Comparison of quality and run time for different circuits on Encore Multimax, with and without any priority assignment

CKT	4 Processors				8 Processors			
	Priority		No Priority		Priority		No Priority	
	Lit	Time	Lit	Time	Lit	Time	Lit	Time
bw	181	35.88	194	33.12	181	22.75	192	15.78
duke2	488	94.34	525	90.87	476	62.40	531	73.73
misex3	524	181.17	551	181.38	523	122.33	557	102.78
misex3c	524	204.12	549	274.85	525	150.71	533	326.02
x3	860	73.52	992	72.37	856	46.19	1039	70.56
dalu	1296	227.86	1543	245.90	1297	119.51	1589	126.63
pair	1879	88.81	1908	86.37	1880	60.76	1894	65.50
frg2	815	130.84	871	137.72	823	60.19	856	67.50
i8	1149	97.05	1263	95.40	1159	65.89	1305	62.43
i9	616	21.11	626	21.88	619	15.62	630	24.70
C499	599	25.97	598	25.67	599	16.77	600	15.24
C5315	2027	795.94	2041	1190.41	2025	557.21	2030	574.41

the kernel extraction and the cube extraction procedures to guide the synthesis process such that it avoids the local minima.

In this subsection, we will evaluate the merit of the priority scheme used in ProperMIS by performing the following experiment. In this experiment, we have assigned identical priorities to *all* of the transformations. Table 3.13 compares the results (literal count and runtime) obtained from this experiment with the results obtained with our priority scheme, on 4 and 8 processors, for Encore Multimax. Similar results are presented for Intel/860 in Table 3.14. The quality of the synthesized circuit is always poor with no priority scheme compared to the quality of the synthesized circuits with our priority scheme. Also, the run time is poor most of the time for the no priority scheme.

Hence, these results demonstrate that the assignment of priorities to different transformations is essential to maintaining good quality of the synthesized circuits and to obtaining good speedups.

Table 3.14 Comparison of quality and run time for different circuits on Intel/860, with and without any priority assignment

CKT	4 Processors				8 Processors			
	Priority		No Priority		Priority		No Priority	
	Lit	Time	Lit	Time	Lit	Time	Lit	Time
bw	182	6.82	189	5.69	186	3.97	186	3.05
duke2	463	25.30	483	24.96	464	18.12	494	21.22
misex3	521	40.09	520	39.05	464	18.12	530	24.60
misex3c	515	45.35	574	83.83	514	38.77	704	71.32
x3	855	25.96	898	36.82	855	18.90	937	29.00
dalu	1284	50.31	1336	85.90	1286	32.22	1330	65.58
pair	1877	30.37	1894	35.59	1886	21.69	1890	32.08
frg2	827	40.67	841	39.92	825	27.95	885	31.64
i8	1158	34.98	1157	42.28	1162	20.42	1179	48.65
i9	618	7.95	623	8.28	618	7.87	623	8.21
C499	599	6.52	600	5.93	599	6.22	601	5.64
C5315	2027	678.41	2040	477.75	2039	196.63	2048	193.05

3.7.6 Effect of synchronizing barriers on quality and speedup

In this chapter, we have presented an asynchronous parallel algorithm, with *no* synchronizing barriers. In this subsection, we want to demonstrate the effect of synchronizing barriers on the speedup and the quality of the circuit produced.

In this experiment, the following transformations are performed with synchronizing barriers separating different phases of transformations: 1) *node simplification*, 2) *kernel extraction with threshold 30*, 3) *cube extraction with threshold 30*, 4) *kernel extraction with threshold 10*, 5) *cube extraction with threshold 10*, 6) *kernel extraction with threshold 0*, 7) *cube extraction with threshold 0* and 8) *resubstitution*.

In Table 3.15, we compare the results (literal count and run time) obtained with the synchronizing barriers with the results obtained with the asynchronous algorithm of ProperMIS, on 4 and 8 processors, for Encore Multimax. Similar results are presented for Intel/860 in Table 3.16. One can observe that the scheme with synchronizing barriers

Table 3.15 Comparison of quality and run time for different circuits in Encore Multimax, with and without Synchronizing Barriers

CKT	4 Processors				8 Processors			
	No Syncronization		Synchronization		No Syncronization		Synchronization	
	Lit	Time	Lit	Time	Lit	Time	Lit	Time
bw	181	35.88	182	37.14	181	22.75	179	24.41
duke2	488	94.34	475	104.68	476	62.40	458	75.88
misex3	524	181.17	522	193.19	523	122.33	521	117.92
misex3c	524	204.12	521	210.99	525	150.71	515	200.76
x3	860	73.52	858	75.25	856	46.19	854	81.07
dalu	1296	227.86	1279	206.43	1297	119.51	1272	151.98
pair	1879	88.81	1880	111.29	1880	60.76	1880	113.84
frg2	815	130.84	781	135.15	823	60.19	824	101.98
i8	1149	97.05	1158	98.09	1159	65.89	1154	90.60
i9	616	21.11	618	25.62	619	15.62	619	25.54
C499	599	25.97	598	30.52	599	16.77	598	21.38
C5315	2027	795.94	1996	936.19	2025	557.21	2022	588.58

produces synthesized circuits of marginally better quality compared to ProperMIS. But the run times are much worse for the scheme with the synchronizing barriers.

3.8 Conclusions

In this chapter, we have presented an asynchronous portable parallel algorithm of logic synthesis using the MIS [16] approach, called ProperMIS. The features of the MIS approach of logic synthesis are 1) algebraic factoring and 2) node simplification with implicit don't cares [37] using two-level minimization techniques. ProperMIS has been implemented as a part of the ProperCAD project [25] based on the CHARM system [26, 27], and it runs on a variety of parallel machines such as Sequent Symmetry, Encore Multimax, Intel/860 Hypercube and a network of workstations. The algorithm developed for ProperMIS uses an asynchronous message-driven data-flow model of computation. No explicit synchronizing barriers are allowed in the algorithm. Another key contribution is

Table 3.16 Comparison of quality and run time for different circuits in Intel/860, with and without Synchronizing Barriers

CKT	4 Processors				8 Processors			
	No Syncronization		Synchronization		No Syncronization		Synchronization	
	Lit	Time	Lit	Time	Lit	Time	Lit	Time
bw	182	6.82	180	7.75	186	3.97	182	5.96
duke2	463	25.30	459	29.17	464	18.12	459	24.27
misex3	521	40.09	524	42.60	520	21.44	513	26.95
misex3c	515	45.35	521	52.87	514	38.77	522	51.06
x3	855	25.96	849	24.18	855	18.90	851	21.42
dalu	1284	50.31	1280	62.24	1286	32.22	1278	46.58
pair	1877	30.37	1880	42.91	1886	21.69	1880	40.86
frg2	827	40.67	815	43.24	825	27.95	817	31.60
i8	1158	34.98	1152	39.07	1162	20.42	1150	39.50
i9	618	7.95	616	8.45	618	7.87	616	10.20
C499	599	6.52	599	9.43	599	6.22	599	7.58
C5315	2027	678.41	2002	793.18	2039	196.63	2028	228.18

that we have *reused* about 80% of the sequential code from MIS. We have implemented the algorithm and have evaluated it on various *MCNC* and *ISCAS* synthesis benchmark examples. Some results are reported on different parallel machines. Results show good speedups in all of the machines with almost no degradation in the quality of the synthesized network over the uniprocessor algorithm.

CHAPTER 4

PARALLEL SYNTHESIS USING PARTITIONING

4.1 Introduction

In the previous two chapters, we have presented portable parallel algorithms for logic synthesis for multilevel circuits using the Transduction [17] and MIS approaches [16]. But the present day computational resources are inadequate to handle very large circuits due to very high memory requirements and prohibitive run times. As the semiconductor industry moves towards more and more integration, circuit size is going to grow very rapidly. As a result, the rapid improvement in the computer performances and memory size will not be able to cope up with the growth of the circuit size.

The worst-case time and storage complexities of the above logic synthesis algorithm are exponential with the size of the circuit. The only solution to the problem of synthesizing a very large circuit is to partition the circuit into smaller parts, synthesize each part separately, and then merge the synthesized parts. Since each part is smaller than the original circuit, the run time and memory requirement will be much smaller than those of the original circuit due to the exponential nature of the time and storage complexity.

However, this partitioning approach to synthesis has disadvantages too. Since different parts are synthesized separately, the sharing of common logic factors is limited to a single part. As a result, the quality of the synthesized circuit will not be optimal.

Hence, to obtain a good quality circuit by using this partitioning approach, the logic partitioning algorithm has to partition a given circuit in such a way that the potential

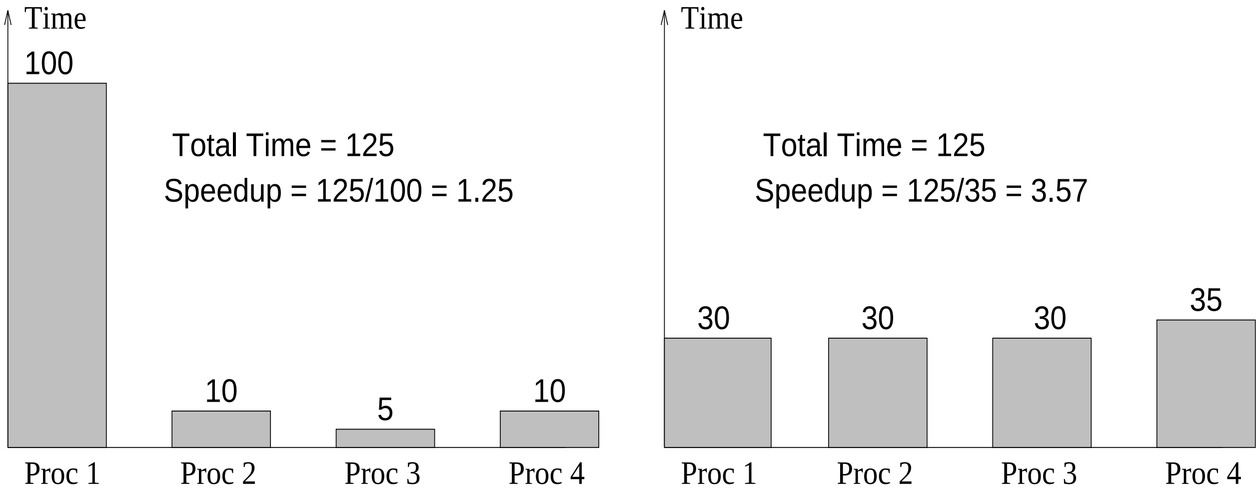

Figure 4.1 Effect of load balancing on speedup in parallel processing

of sharing common terms among the nodes in a partition is maximal and the potential of sharing common terms among the nodes in different partitions is minimal. Thus, the primary objective of this approach is to partition the circuit in such a fashion that the quality of the synthesized circuit is optimal.

We also have to consider one more property of the partitioned circuit during the partitioning process. We plan to synthesize the partitions in parallel. The completion time of the synthesis procedure in parallel is bounded by the largest completion time among all of the partitions. To reduce the parallel synthesis time, one has to minimize the largest synthesis time among all of the partitions. This is illustrated in Figure 4.1. Hence, the secondary objective of this approach is to partition in such a fashion that the largest synthesis time among all of the partitions is minimized.

In this chapter, we will describe a parallel logic synthesis system using the partitioning approach.

4.2 Previous Work in Partitioning

A multilevel logic network can be viewed as a directed acyclic graph (DAG) where each node corresponds to a variable y_i with its logic function in sum-of-products and/or factored form. An arc corresponds to the direct dependence of the variables. The primary inputs (PI) and primary outputs (PO) form the sources and sinks of the graph. Our problem is to partition a given DAG of the entire circuit into DAG's of smaller circuits.

The graph partitioning problem has been addressed by researchers in numerous disciplines. Given a graph of nodes and arcs, the objective is to partition the nodes into two or more subgraphs such that the total cost of the arcs between nodes in different partitions is minimized.

The optimum graph-partitioning problem is known to be NP-complete [45]. Efficient heuristics for partitioning based on the group migration method have been proposed by Kernighan and Lin to reduce the total cost of the cut between two partitions [47]. Such a heuristic can be used to reduce the number of connections across the cut if the cost of each connection is considered unity. The algorithm takes $O(n^3)$ time per pass, where n is the number of nodes in the graph. A more efficient algorithm has been proposed with a similar approach with linear time complexity [48]. Quadrisection is an extension of the above bipartitioning algorithm in which nodes are partitioned into four subsections [49]. Recently, the *ratio-cut* approach has been proposed which does not impose hard limits on the subset sizes [50]. It has been demonstrated that the ratio-cut algorithm can locate clustering structures in the circuit. In [51], top-down clustering is performed by using a ratio-cut algorithm, and then iterative moves are made to enhance the partitioning.

A partitioning approach based on the seed clustering method has been reported in [52]. This method generates k seeds for k partitions and then the remaining nodes are clustered around the seeds to generate k partitions. This method satisfies some constraints on the number of internal nodes and primary inputs/outputs of each block by moving nodes from one partition to another.

Recent work has been reported on a circuit partitioning method based on the analysis of reconvergent fan-out [53]. In this approach, a *corolla* is defined as a set of overlapping reconvergent fanout regions. The circuit is partitioned into a set of nonoverlapping corollas. Since the circuits can have a large number of reconvergent fan-out regions, a reconvergence zone may contain the entire circuit. Hence, a heuristic method is applied to choose the corollas. In [54], a circuit segmentation scheme has been presented for pseudoexhaustive testing of a circuit. In that scheme, a circuit is reduced by collapsing the reconvergent zones into single gates as far as possible such that a reconvergent zone is contained in one segment entirely.

None of the work described above addressed the testability of the block as an aspect to be included in the measure of the quality of a partition. When a circuit is partitioned into smaller blocks, each block is synthesized separately and then synthesized blocks are merged into the resultant circuit, the resultant circuit is not guaranteed to be 100% irredundant even if each of the synthesized blocks is 100% irredundant. This is due to the don't-care sets that exist because of the interconnections between different blocks. A recent work presented a probabilistic scheme to estimate the size of the don't-care sets across the partitions and that estimate was used to minimize the cost of partitioning [39]. The results presented in the paper showed marked improvement in the testability of the synthesized circuits over other known partitioning methods.

4.3 Partitioning Algorithm

In this section, we will present the logic partitioning algorithm we propose for our partitioning approach of logic synthesis. The primary objective of this partitioning algorithm is to retain the logic minimization potential as much as possible after the partitioning. To achieve that, the partitions have to capture the gross structural features of the given circuit. We use a variant of the clustering method proposed in [55, 52] in our partitioning algorithm to make effective use of the information regarding the structure of the given circuit.

However, the primary reason for resorting to the partitioning approach of logic synthesis is that the computational resources available today are inadequate to handle very large circuits. A large circuit is partitioned into smaller blocks such that each individual block is small enough to be handled by the synthesis tool. But if one of the partitions is much larger than the rest of the partitions (in order to preserve the gross structural features of the given circuit), that partition may be too large to be handled by the synthesis tool. Hence, care should taken to ensure that all of the partitions are of almost equal size.

We have a secondary objective during partitioning a logic circuit. We plan to synthesize the partitions in parallel. The completion time of the parallel synthesis procedure is bounded by the largest completion time among all of the partitions. Hence, to reduce the completion time of the parallel synthesis procedure, one needs to minimize the largest completion time for synthesis among all of the partitions.

4.3.1 Size of circuit and synthesis time

Since we want to minimize the maximum completion time for synthesis among all of the partitions, we must have some estimate of the completion time for synthesis during partitioning. But the synthesis time of a circuit depends on many factors, such as the size of the circuit, the synthesis algorithm used, the number of primary inputs and outputs, and the complexity of the logic expressions of nodes in the circuit. To generate a complete mathematical model for the completion time for synthesis for any give circuit is a very complex task and is beyond the scope of this thesis.

Hence, we simplified our model considerably. We plan to use MIS-II [16] to synthesize each partition, thus the synthesis algorithm is not a variable in the model. We assume that the synthesis time is a function of the size of the circuit alone. The size of the circuit is measured by the initial literal count of the circuit. We assume that the synthesis time (T) is proportional to some power of the size of the circuit (S) as given in Equation (4.1).

$$T \;=\; \beta \star S^{\alpha} \tag{4.1}$$

By taking the natural logarithm of both sides of Equation (4.1), we obtain

$$\log T \;=\; \alpha \star \log S \;+\; \log \beta \tag{4.2}$$

To determine the values of α and β empirically, we performed an experiment. We performed synthesis using MIS-II on 27 benchmark circuits with various sizes and collected the run times for synthesis and the original sizes of the circuit in terms of the literal

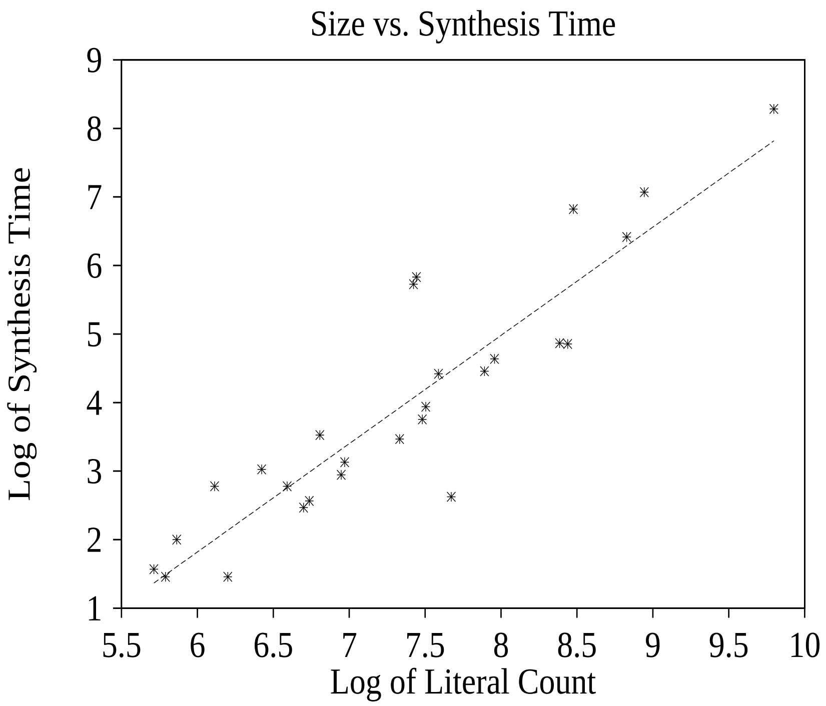

Figure 4.2 Variation of synthesis time with the size of the original circuit in terms of literal count

count. Those data, scatter plotted on a log scale, are presented in Figure 4.2. Using the statistical method of *least-square curve fitting* on the data, we computed the value of α to be 1.58 and the value of β to be 0.00047. Hence, the empirical equation relating the run time of the synthesis procedure (T) to the size of the circuit in terms of the literal count (S) is given in Equation (4.3).

$$T \;=\; 0.00047 \star S^{1.58} \tag{4.3}$$

4.3.2 Cost function

A cost function is used to guide the partitioning process and discern the best move among all possible moves. As we have mentioned earlier, our first objective during partitioning is to capture the gross structural features of the circuit as much as possible so that an optimal quality circuit is produced after the synthesis is performed. But we also want to minimize the maximum completion time for synthesis among all of the partitions. As suggested by Equation (4.3), if two partitions have the same size in terms of the literal count, the run times for synthesis for those two partitions will be approximately the same. Hence, our second objective during the partitioning is to partition into equal size blocks, in terms of literal count.

We have modified the cost function given in [52] to suit our purpose. We compute the average size of a partition, AS, as follows:

$$AS \;=\; \sum_{for\,all\,nodes\,in\,circuit} S(node) \,/\, N$$

where

$$S(node) \;=\; 0.00047 \star (literal\_count(node))^{1.58}$$

and N is the number of partitions. Let us denote AI to be the average number of inputs to each partition. Since AI can not be exactly determined a priori, it is approximated as

$$AI \;=\; number\,of\,primary\,inputs \,/\, 2$$

Let us consider a node η we want to put in a partition π. Let us denote DI_π to be the change in the number of inputs of the block π caused by moving η into π, and $PS(\pi)$ to be the size of the partition π prior to moving the node η to π. Then the cost of moving the node η to the partition π is expressed as follows:

$$cost(\eta, \pi) \;=\; C_1 \star (DI_\pi / AI) - (1 - C_1) \star (S(\eta)/AS) \star SIGN(AS - PS(\pi) - S(\eta)) \quad (4.4)$$

where

$$SIGN(val) = \begin{cases} -1.0 & \text{val} < 0 \\ 1.0 & \text{otherwise} \end{cases}$$

The cost function given in Equation (4.4) has two parts. The first part penalizes a move if it introduces many additional inputs to the block. Hence, that part encourages the acceptance of a node which forms a good cluster. The second part of the cost function encourages a move of a large-sized node into the block as long as the block size does not exceed AS after the node is moved. On the other hand, if the size is going to exceed

Table 4.1 Comparison on literal counts and the run times (1, 2 and 4 processors) between our partitioning algorithm and BEAT_NP for 4 partitions

CKT	Our Partitioner				BEAT_NP			
	Lit	Run Time (sec)			Lit	Run Time (sec)		
	Cnt	1 P	2 P	4 P	Cnt	1 P	2 P	4 P
seq	2369	687.55	417.52	236.30	1936	800.14	524.26	476.84
des	4526	1718.01	1157.29	717.70	4202	2074.81	1106.38	903.99
k2	1611	2397.37	2210.11	2123.63	-	-	-	-
C7552	2948	443.29	335.99	231.78	3469	936.62	785.40	738.00
C6288	3673	1091.57	602.62	365.58	3998	379.35	260.61	204.08
C5315	2284	234.48	168.51	132.56	2438	228.43	172.01	129.79
duke2_berger	826	162.60	130.39	99.44	-	-	-	-

AS, it penalizes that move. This part of the cost function encourages the formation of equal-sized partitions.

The BEAT_NP partitioning algorithm [52] implicitly assumes all of the nodes to be of equal size; hence, it gives the same weight to all of the individual nodes. In our partitioning algorithm, we used the literal count of a node as a weight of that node. We performed experiments to observe the effectiveness of that decision. In Table 4.1, we compare the final literal counts and the run times (on 1, 2 and 4 processors) obtained by applying the one-pass approach (described in a later section of this chapter) with two partitioning algorithms: 1) the partitioning algorithm described earlier in this section with the size of a node as its literal count, and 2) our implementation of the BEAT_NP algorithm assuming the nodes to be of equal size. This experiment was performed by partitioning the given circuits into 4 parts.

One can observe from the data presented in Table 4.1 that for most of the circuits, the run times were much higher for the BEAT_NP algorithm for 1 processor compared to those with our partitioner. For two circuits, *k2* and *duke2_berger*, synthesis could not be completed when BEAT_NP was used to partition them because one of the partitions became very large and complex. Another point to be observed is that the speedups obtained with BEAT_NP were poor compared to those obtained with our partitioner, as we employ multiple processors. This shows that the load balancing is not good with the BEAT_NP algorithm. The run times for synthesis for various partitions varied significantly and as a result, the speedups obtained were poor.

4.3.3 Methodology

The partitioning procedure starts by generating N seeds for N partitions. The seed generation method is the same as that given in [52]. All of the primary inputs and primary outputs of the circuit are grouped into a set called the *boundary vertices*, B. The rest of the nodes, C, are called the *candidate vertices* for the seeds. The shortest path distance $dist(\gamma, \beta)$ from a vertex $\gamma \in C$ to every $\beta \in B$ is computed. The $mindist$ is computed as

$$mindist(\gamma) \;=\; MIN_{\beta \in B}(dist(\gamma, \beta)) \;\forall \gamma \in C$$

Then $\psi \in C$ is chosen as a seed if

$$mindist(\psi) = MAX_{\gamma \in C}(mindist(\gamma))$$

Then ψ is put into the set of boundary nodes, B, and the same procedure is repeated until N seeds are generated. These N seeds are put in N partitions.

Then the others nodes are put one by one into different partitions. The algorithm starts by selecting the partition π which has the minimum size in terms of the literal count. It checks all the neighbors of the partition π, and the node ν, which has the minimum cost to move into π according to Equation (4.4), is chosen and put into π. If no such suitable neighbor is found, a new seed is generated by using the procedure described in the last paragraph and is put to π. This process is repeated until all of the nodes are put into one of the N partitions.

4.3.4 Impact of partitioning algorithm

In this section, we have presented a clustering-based partitioning algorithm, which is a modified version of the algorithm presented in [52]. Another important partitioning

Table 4.2 Comparison on Literal counts, partition times and the total run times (1 and 4 processors) between our partitioning algorithm and Kernighan-Lin's algorithm for 4 partitions

CKT	Our Partitioner				Kernighan Lin			
	Lit Cnt	Part Time	Run Time (sec) 1 P	Run Time (sec) 4 P	Lit Cnt	Part Time	Run Time (sec) 1 P	Run Time (sec) 4 P
seq	2369	2.30	687.55	236.30	2460	3.18	724.07	273.54
des	4526	110.97	1718.01	717.70	4059	82.51	2155.36	584.73
C7552	2948	124.90	443.29	231.78	3416	1239.43	1851.34	1406.20
C6288	3673	44.09	1091.57	365.58	4207	1504.20	2198.66	1672.67
C5315	2284	54.10	234.48	132.56	2499	309.70	566.85	378.08
duke2_berger	826	18.63	162.60	99.44	726	4.38	296.65	254.14

approach is the group-migration-based approach and Kernighan-Lin's algorithm [47] is an example of that approach to partitioning. In this subsection, we want to justify our decision of using a clustering-based partitioning algorithm as opposed to a group-migration-based partitioning algorithm. In Table 4.2, we compare the final literal counts, the run times for partitioning and the total run times (on 1, 2 and 4 processors) obtained by applying the one-pass approach (described in a later section of this chapter) by using our partitioning algorithm with those obtained by using Kernighan-Lin's partitioning algorithm [47].

One can observe from Table 4.2 that in most cases, the run times are much higher if Kernighan-Lin's algorithm of partitioning is used. This is due to the fact that Kernighan-Lin's algorithm is computationally more expensive. However, the quality of the final synthesized circuit produced by using Kernighan-Lin's algorithm is comparable (in many cases, worse) to the quality of the final synthesized circuit produced by our algorithm. These results show that it is better to use a clustering-based partitioning method compared to a group-migration-based method, since the extra partitioning cost for the group migration method is not paying off.

4.4 One-Pass Approach of Synthesis

4.4.1 Methodology

In this section, we will describe the overall synthesis methodology using the one-pass approach we have used in our partitioning approach of parallel logic synthesis. We have described the partitioning algorithm earlier in this chapter. But one can potentially use any partitioning algorithm to partition the circuit. The methodology for parallel synthesis is independent of the partitioning algorithm used.

The entire system is developed as a part of the ProperCAD project [25], based on the CHARM run time system [26, 27]. As a result, this system is portable across a variety of parallel machines, but we will report results on only a networks of workstations. Given a combinational circuit, it first partitions the circuit into N partitions, using a partitioning algorithm chosen by the user. The partitioning is performed on a single processor. We have not looked into the problem of parallelizing the partitioning algorithm because it is beyond the scope of this thesis. Also, the partitioning time forms a small fraction of the total synthesis time. If a suitable parallel partitioning algorithm is available, that algorithm can be applied to partition the circuit in parallel using multiple processors.

After the partitioning is performed, objects are created for each of the partitions and the objects are distributed to different processors by the CHARM run time system. When a partition object is picked up by a processor, that partition is synthesized by a combinational synthesis algorithm. We have used the MIS algorithm [16] to synthesize the individual partitions, but we could have used any other synthesis algorithm such as the Transduction method [17] as well. After the completion of synthesis on all of the partitions, all of the synthesized partitions are merged to form the synthesized circuit. This system is implemented as a part of the ProperCAD project [25] and based on the CHARM runtime system and named as ProperPART.

Table 4.3 Comparison of quality (literal count in sum-of-products form) and run time in one processor (in sec) obtained by applying ProperPART with one-pass approach with that obtained by applying MIS 2.2 on the entire circuit

CKT	Init Lit Cnt	MIS 2.2		ProperPART (one-pass)			
				4 Partitions		8 Partitions	
		Lit	Time	Lit	Time	Lit	Time
seq	17973	1367	7907.40	2369	687.55	2974	503.08
des	7657	4024	2784.00	4526	1718.01	4892	1374.40
k2	3063	-	-	1611	2397.37	2003	372.84
C7552	6144	2691	652.60	2948	443.29	2980	334.27
C6288	4800	3729	1037.20	3711	428.37	3695	315.18
C5315	4386	2010	241.70	2284	234.48	2431	199.29
duke2_berger	1314	-	-	826	162.60	920	72.38

4.4.2 Experimental results

In this subsection, we compare the experimental results obtained by applying the one-pass approach on various *ISCAS* and *MCNC* benchmark circuits.

4.4.2.1 Comparison with MIS-II

In Table 4.3, we compare the literal counts (in sum-of-products form) and the run times obtained by running MIS 2.2 with those obtained by running the one-pass approach of ProperPART on the benchmark circuits. The run times for ProperPART as well as MIS 2.2 presented in this table are obtained by running on a *uniprocessor SUN4 workstation.* The runtime for ProperPART for the one-pass approach on any circuit includes the initial partitioning time, parallel synthesis time for various partitions (on uniprocessor, partitions were synthesized one by one) and the final merge time. A "-" in any table in this chapter means it either ran out of memory or it could not finish in 40 hours.

One can observe that the quality of the final synthesized circuit obtained by ProperPART is not as good as that obtained by running MIS 2.2 on the entire circuit. It is also clear that the quality of the synthesized circuit goes down as the number of partitions increases. But on some circuits, MIS 2.2 could not be run on the entire circuit because either it ran out of memory or it could not finish after running for a long time. For those circuits, this partitioning approach is the only solution to the problem of synthesizing them. One can also observe that the run time for the one-pass approach of ProperPART for a large circuit is much smaller than that for MIS 2.2 on the same circuit. Also, the run time for the one-pass approach becomes smaller as the number of partitions increases. This is due to the fact that each of the partitions becomes simpler in terms of the complexity of the synthesis operations.

4.4.2.2 Speedup results

We will now present the speedup results obtained by applying the one-pass approach of ProperPART on network of SUN4 workstations for various benchmark circuits. The results for 4 partitions are presented in Table 4.4 and the results for 8 partitions are presented in Table 4.5. One can observe that the speedup results are reasonably good for most of the circuits. Only the circuit $k2$ performed poorly for 4 partitions in terms of the speedup result. This is due to the fact that in spite of our effort during partitioning to make the synthesis time for all of the partitions equal, the synthesis time for one of the partitions became much larger than that of other partitions. As a result, the run time of ProperPART is dominated by the synthesis time of one partition, which is executed in one processor. Hence, there is not much reduction in run time by using a greater number of processors.

4.5 Iterative Approach of Synthesis

4.5.1 Methodology

The major limitation of the one-pass approach described in the last section is that the quality of the circuit is not optimal. This is due to the reason that the synthesis is performed on only one partition at a time. Hence, there will be no sharing of common

Table 4.4 Run-time(speedup) results obtained by applying ProperPART with one-pass approach on a network of SUN4 workstations for 4 partitions

CKT	1 Proc. Sec(spd)	2 Proc. Sec(spd)	4 Proc. Sec(spd)
seq	687.55(1.0)	417.22(1.7)	236.20(2.9)
des	1718.01(1.0)	1157.29(1.5)	717.70(2.4)
k2	2397.37(1.0)	2210.11(1.1)	2123.63(1.1)
C7552	443.29(1.0)	335.99(1.3)	231.78(1.9)
C6288	428.37(1.0)	262.79(1.6)	184.28(2.3)
C5315	234.48(1.0)	168.51(1.4)	132.56(1.8)
duke2_berger	162.60(1.0)	130.39(1.3)	99.44(1.6)

Table 4.5 Run-time(speedup) results obtained by applying ProperPART with one-pass approach on a network of SUN4 workstations for 8 partitions

CKT	1 Proc. Sec(spd)	2 Proc. Sec(spd)	4 Proc. Sec(spd)	8 Proc. Sec(spd)
seq	503.08(1.0)	279.77(1.8)	137.15(3.7)	126.49(4.0)
des	1374.40(1.0)	812.14(1.7)	485.39(2.8)	393.50(3.5)
k2	372.84(1.0)	285.70(1.3)	142.48(2.6)	125.18(3.0)
C7552	334.27(1.0)	242.24(1.4)	135.47(2.5)	113.85(3.0)
C6288	315.18(1.0)	188.80(1.7)	96.95(3.3)	68.32(4.6)
C5315	199.29(1.0)	140.75(1.4)	79.73(2.5)	65.44(3.1)
duke2_berger	72.38(1.0)	49.29(1.5)	28.89(2.5)	22.37(3.2)

logic among the nodes which are in different partitions. That can potentially degrade the quality of the resultant synthesized circuit.

Very large circuits cannot be synthesized as a whole due to prohibitive run times and memory requirements. As a result, one cannot improve the quality of the synthesized circuit by performing synthesis again on the entire synthesized circuit. Hence, we have devised an iterative procedure to improve the quality of the circuit. The main idea is to allow synthesis among certain constrained sets of nodes which are in different partitions at one time, and this procedure is repeated a certain number of times. This iterative procedure is explained with an example in Figure 4.3 with 4 partitions. The partitions are numbered from 1 to 4 in the figure.

Figure 4.3(a) shows the first phase of the iteration. This is the same as the one pass approach described in the last subsection. It shows that each partition is synthesized independently. In this phase, a node η in a particular partition can share logic with only those nodes which are in the same partition as η. After the first phase, we obtain the synthesized version of the four partitions of the circuit. In the second phase, as shown in Figure 4.3(b), we partition each of the four partitions obtained in the last phase into two parts, and mark them as A and B. Then we merge the partitions $1A$ and $2A$ to form the new partition 1 and merge $1B$ and $2B$ to form the new partition 2. Similarly, we merge the partitions $3A$ and $4A$ to form the new partition 3 and merge $3B$ and $4B$ to form the new partition 4. Now these new partitions (1 to 4) are synthesized independently. In this phase, one-half of the nodes of partition 1 are synthesized with one-half of the nodes of partition 2, and the other half of the nodes of partition 1 are synthesized with the other half of the nodes of partition 2. This will allow some logic sharing among the nodes in partitions 1 and 2. The same is true for partitions 3 and 4. This can potentially improve the quality of the circuit, but will never degrade the quality. In the third phase, as shown in Figure 4.3(c), each partition is partitioned again into two parts. But this time, partitions 1A and 3A (1B and 3B) are paired and partitions 2A and 4A (2B and 4B) are paired and the same procedure is repeated. In the fourth phase, as shown in

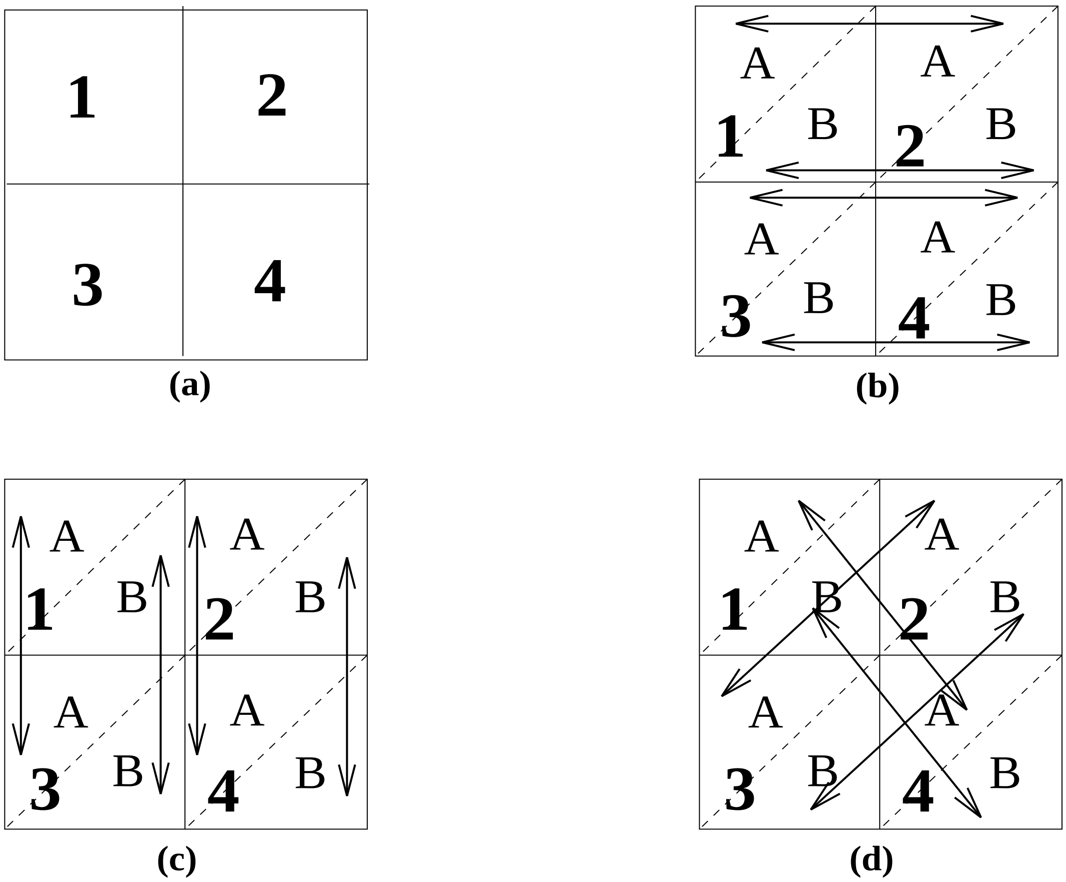

Figure 4.3 An example of iterative approach of synthesis using the partitioning approach with 4 partitions

Figure 4.3(d), partitions 1A and 4A (1B and 4B) are paired and partitions 2A and 3A (2B and 3B) are paired and the same procedure is repeated.

We have to generate the pairing of different partitions for different phases of this iterative approach. We will assume that the number of partitions, N, is a power of 2, i.e., N = 2^k where k is a positive integer. We need to generate the pairing in such a way that each partition is paired with different partitions in different phases of this iterative approach. Also, in any phase, any particular partition is involved in only one pairing. Then, it is obvious that the number of phases is the same as the number of partitions, N. Also, the number of pairings in any phase is N.

For example, for 4 partitions, the pairings at different phases are given as

Phase 2: [(1A, 2A), (1B, 2B), (3A, 4A), (3B, 4B)]

Phase 3: [(1A, 3A), (1B, 3B), (2A, 4A), (2B, 4B)]

Phase 4: [(1A, 4A), (1B, 4B), (2A, 3A), (2B, 3B)]

Phase 1 is the same as the one-pass approach, i.e., all of the individual partitions are synthesized independently. Hence, phase 1 is not listed here. For all future listings of phases, we will omit phase 1 for the sake of brevity.

A *pairing* is defined to be a list of two numbers. For example, *(1, 2)* is a pairing. It is implicit that the underlying synthesis partitions are (1A, 2A) and (1B, 2B) for this pairing.

A *phase* is defined to be a list of pairings. For example, *[(1, 2), (3, 4)]* is a phase.

A *phase_list* is defined to be a list of phases. For example, {*[(1, 2), (3, 4)], [(1, 3), (2, 4)], [(1, 4), (2, 3)]*} is a phase_list.

The operation *phase_merge* on a set of phase_lists creates a single phase_list by merging individual phases of the set of phase_lists on a one-to-one basis. For example, consider the following two phase_lists:

```
   phase_list1              phase_list2
{[(1, 2), (3, 4)]        {[(5, 6), (7, 8)]
 [(1, 3), (2, 4)]         [(5, 7), (6, 8)]
 [(1, 4), (2, 3)]         [(5, 8), (6, 7)]
 [(1, 5), (2, 6)]         [(3, 7), (4, 8)]
 [(1, 6), (2, 5)]         [(3, 8), (4, 7)]
 [(1, 7), (2, 8)]         [(3, 5), (4, 6)]
 [(1, 8), (2, 7)]}        [(3, 6), (4, 5)]}
```

If we perform the operation phase_merge on phase_list1 and phase_list2, the resultant phase_list becomes:

```
{[(1, 2), (3, 4), (5, 6), (7, 8)]
 [(1, 3), (2, 4), (5, 7), (6, 8)]
 [(1, 4), (2, 3), (5, 8), (6, 7)]
 [(1, 5), (2, 6), (3, 7), (4, 8)]
 [(1, 6), (2, 5), (3, 8), (4, 7)]
 [(1, 7), (2, 8), (3, 5), (4, 6)]
 [(1, 8), (2, 7), (3, 6), (4, 5)]}
```

The operation *phase_union* on a set of phase_lists merges the lists to form a single phase_list. For example, consider the following two phase_lists, phase_list1: {[(1, 2), (3, 4)]} and phase_list2: {[(1, 3), (2, 4)], [(1, 4), (2, 3)]}. If we perform the operation phase_union on phase_list1 and phase_list2, the resultant phase_list becomes: {[(1, 2), (3, 4)], [(1, 3), (2, 4)], [(1, 4), (2, 3)]}.

The operation *gen_phaselist* generates a phase_list, given two lists of numbers. The main idea behind this operation is to generate a phase_list in such a fashion such that any pair in any phase of the phase_list contains one number from the first list of numbers and the other number from the second list of numbers. It also generates all possible phases subject to the constraint that any number from any of the two lists, e.g., the number ν from the list λ, is paired with different numbers from the other list in the different phases of the phase_list. It is assumed that the number of items in the two lists are the same. It is also assumed that the number of items in a list is a power of 2, i.e., the number of items is of the form 2^k, where k is a nonnegative integer. For example, consider the two lists $L1$ and $L2$. The list $L1$ contains the numbers 1 and 2, and the list $L2$ contains the number 3 and 4. Then the operation gen_phaselist on the lists $L1$ and $L2$ will generate the following phase_list: {[(1, 3), (2, 4)], [(1, 4), (2, 3)]}.

Let us denote *first(L)* to be the first number of a list of numbers L. We denote *first_half(L)* to be the the first half of a list of numbers and *last_half(L)* to be the last half. Then the operation gen_phaselist can be described recursively as follows:

$$
gen\_phaselist(L_1, L_2) = \begin{cases} \{[(first(L_1), first(L_2))]\} & \|L_1\| = 1 \\ \left.\begin{array}{c} \left(\begin{array}{l} \texttt{gen\_phaselist}(first\_half(L_1), first\_half(L_2)) \\ \quad \texttt{phase\_merge} \\ \texttt{gen\_phaselist}(last\_half(L_1), last\_half(L_2)) \end{array}\right) \\ \texttt{phase\_union} \\ \left(\begin{array}{l} \texttt{gen\_phaselist}(first\_half(L_1), last\_half(L_2)) \\ \quad \texttt{phase\_merge} \\ \texttt{gen\_phaselist}(last\_half(L_1), first\_half(L_2)) \end{array}\right) \end{array}\right\} & \text{otherwise} \end{cases}
$$

Using these operations, one can generate the complete phase_list, given the number of partitions N. The algorithm for generating the complete phase_list is given in Figure 4.4. After the complete phase_list is derived, it indicates the pairings of the partitions needed at different phases of the iterative approach.

It can be observed that in the iterative approach, the very first partitioning and the very last merging are performed by only one processor. During the other phases, N partitions are bipartitioned and then they are merged to form N new partitions according to the pairings listed for that phase. Those operations can be performed in parallel by distributing those jobs to different processors. Also, the synthesis on different partitions can be performed in parallel by distributing them to the different processors. Another important feature of this iterative approach is that the size of the partitions handled by the synthesis algorithm remains approximately the same at different phases of this iterative approach, i.e., the partition sizes do not grow. This is due to the fact that we are bipartitioning and merging in different combinations in different phases, but we are not merging two existing partitions to form a larger partition. Hence, it is possible to apply this approach to the large circuits which can not be handled as a whole by the synthesis algorithms.

Algorithm: complete_phaselist (N)
/* For generating phaselist, given # partition N */

```
for(i = 1; i <= N; i++)
    list[i] = i;
phase_list = NULL;
for(i = N; i >= 2; i = i/2)
    phase_list = phase_union(complete_phaselist_recur(list, i), phase_list);
return(phase_list);
```

Algorithm: complete_phaselist_recur (list, sz)
/* Recursively bipartition the list unless size of the list reaches */
/* 'sz'. Then it generates phase_list using gen_phaselist algorithm */

```
if(size(list) == sz)
    phase_list = gen_phaselist(first_half(list), last_half(list));
else
    phase_list = phase_merge(complete_phaselist_recur(first_half(list), sz),
                         complete_phaselist_recur(last_half(list), sz));
return(phase_list);
```

Figure 4.4 The algorithm for generating complete phase_list when the number of partitions is supplied

Table 4.6 Comparison of quality (literal count in sum-of-products form) and run time in one processor (in sec) obtained by applying ProperPART with iterative approach with that obtained by applying MIS 2.2 on the entire circuit

CKT	Init Lit Cnt	MIS 2.2		ProperPART (Iterative) 4 Partitions		ProperPART (Iterative) 8 Partitions	
		Lit	Time	Lit	Time	Lit	Time
des	7657	4024	2784.00	4383	2707.40	4659	3185.37
k2	3063	-	-	1500	2655.66	1692	989.44
C7552	6144	2691	652.60	2824	834.20	2893	1225.26
C6288	4800	3729	1037.20	3673	1091.37	3652	1566.78
C5315	4386	2010	241.70	2179	534.61	2320	884.69
duke2_berger	1314	-	-	774	276.08	840	370.63

4.5.2 Experimental results

4.5.2.1 Comparison with MIS-II

In Table 4.6, we compare the literal counts (in sum-of-products form) and the run times (on uniprocessor) obtained by running MIS 2.2 with those obtained by running the iterative approach of ProperPART on the benchmark circuits. The run times for ProperPART as well as MIS 2.2 presented in this table are obtained by running on a *uniprocessor SUN4 workstation.* The run time for ProperPART for the iterative approach on any circuit includes the initial partitioning time, parallel partitioning-merge-synthesis times at different phases (on uniprocessor, done one by one sequentially) and the final merge time. As mentioned earlier, a "-" in any table means it either ran out of memory or it could not finish in 40 hours.

One can observe that the quality of the synthesized circuit obtained by the iterative approach is always better compared to the quality obtained by the one-pass approach. But the quality is not as good as that obtained by applying MIS 2.2 on the entire circuit, whenever it is possible to run MIS 2.2 on the entire circuit. But for two circuits, *k2* and *duke2_berger*, MIS 2.2 could not run on the whole circuit. The run times for the iterative approach are comparable to the run times of MIS 2.2. This shows that this iterative approach is good for really large circuits, for which MIS can not be run on the entire circuit. It can also be observed that the run time for the iterative approach increases as the number of partitions increases. This is due to the fact that the number of phases for the iterative approach increases as the number of partitions increases, which in turn increases the run time.

4.5.2.2 Speedup results

We will now present the speedup result obtained for the iterative approach for various benchmark circuits by applying ProperPART on a network of SUN4 workstations. The run times and speedup results with 4 partitions are presented in Table 4.7 and the results for 8 partition are presented in Table 4.8. The speedup results are very good for most of the circuits. Another point to be observed is that the speedup results presented for the iterative approach is much better than those obtained for the one-pass approach (presented in the last section). This is due to the fact that the fraction of all of the works which can be performed in parallel is much greater for the iterative approach than for the one-pass approach. In each phase of the iterative approach, N partitioning, merging and synthesis are performed, where N is the number of partitions. Those operations can be performed in parallel. Also, there are N phases in the iterative approach. As a result, the speedup results are better with a larger number of partitions, as is obvious from the results in Table 4.7 and 4.8.

4.6 Conclusions

In this chapter, we have presented the partitioning approach of parallel logic synthesis. Given a combinational circuit, the circuit is partitioned into N partitions and then those partitions are synthesized in parallel by using multiple processors. Then the synthesized partitions are merged to form the synthesized circuit. This approach is especially suitable for the very large circuits which cannot be handled as a whole by any synthesis algorithm due to prohibitive run times or memory requirements. Since the completion time for

Table 4.7 Run-time(speedup) results obtained by applying ProperPART with iterative approach on a network of SUN4 workstations for 4 partitions

CKT	1 Proc. Sec(spd)	2 Proc. Sec(spd)	4 Proc. Sec(spd)
des	2707.40(1.0)	1672.64(1.6)	1014.64(2.7)
k2	2655.66(1.0)	2348.56(1.1)	2194.99(1.2)
C7552	834.20(1.0)	536.34(1.6)	391.40(2.1)
C6288	1091.57(1.0)	602.62(1.8)	365.58(3.0)
C5315	534.61(1.0)	323.98(1.6)	216.44(2.5)
duke2_berger	276.08(1.0)	193.44(1.4)	133.39(2.0)

Table 4.8 Run-time(speedup) results obtained by applying ProperPART with iterative approach on a network of SUN4 workstations for 8 partitions

CKT	1 Proc. Sec(spd)	2 Proc. Sec(spd)	4 Proc. Sec(spd)	8 Proc. Sec(spd)
des	3185.37(1.0)	1721.43(1.8)	962.44(3.3)	652.44(4.9)
k2	989.44(1.0)	605.44(1.6)	308.06(3.2)	210.80(4.7)
C7552	1225.26(1.0)	685.30(1.8)	365.83(3.4)	241.66(5.1)
C6288	1566.78(1.0)	822.84(1.9)	416.21(3.8)	240.07(6.5)
C5315	884.69(1.0)	494.14(1.8)	265.15(3.3)	168.14(5.3)
duke2_berger	370.63(1.0)	213.25(1.7)	116.35(3.2)	77.44(4.8)

synthesis in parallel is bounded by the largest completion time for synthesis among all of the partitions, the completion time for synthesis in parallel is minimized if the maximum synthesis time among all of the partitions is minimized. That can be achieved if the synthesis times for all of the partitions are approximately the same. We have presented a partitioning algorithm that uses a heuristic method to make the synthesis times for all of the partitions to be approximately the same. The results show good speedup results for most of the circuits.

Since in this approach the partitions are synthesized independently, in most of the cases, the quality of the synthesized circuit will not be as good as it would be if the entire circuit as a whole is synthesized (whenever it is possible to synthesize the entire circuit as a whole). Hence, we have devised an iterative approach to improve the quality of the synthesized circuit by performing synthesis at different phases. At each phase, only certain sets of nodes are allowed to perform synthesis together. The results show that the quality of the synthesized circuit improves modestly by using this iterative approach over that obtained by the one-pass approach.

CHAPTER 5

CONCLUSIONS

5.1 Summary of Thesis Work

The complexity of the VLSI circuit is increasing very rapidly and the existing Computer-Aided Design (CAD) algorithms for VLSI will not be able to handle large circuits in the near future due to prohibitive run times. Researchers have started using parallel processing to speed up the computational time for VLSI design. Parallel algorithms have been developed for almost all applications of VLSI design. Logic synthesis and verification is a very important phase of VLSI design and it is computationally very complex. Some research has been performed on developing parallel algorithms for logic synthesis and verification.

However, most of the parallel algorithms developed for VLSI applications have one major drawback. They have been developed for one specific parallel architecture. As a result, they are very difficult to port to different parallel machines. This limitation has serious consequences. The parallel algorithm for any given application has to be developed afresh for every target parallel architecture. This is very expensive procedure because the software development cycle for parallel programs is considerably longer than for sequential programs.

The ongoing ProperCAD project [25] at the University of Illinois offers an encouraging solution to this problem. It allows the user to develop the parallel algorithms using the CHARM language and run-time system such that the programs developed are portable across a variety of parallel machines, both shared and distributed memory machines.

In this thesis, we have developed a parallel algorithm for combinational logic synthesis using the Transduction method. The Transduction method uses the concept of a *set of permissible functions* to perform various logic transformations to reduce the size of the logic circuit. The logic transformations used in the Transduction method are *gate substitution*, *gate merging*, *generalized gate substitution*, *gate input reduction* and *connection pruning*. Each individual transformation has been parallelized, and they have been integrated into a single system.

We have also developed a parallel algorithm for combinational logic synthesis using the MIS approach [16]. The key features for the MIS approach are: 1) algebraic factoring and 2) node simplification. The transformations which perform algebraic factoring are *kernel extraction*, *cube extraction* and *resubstitution*. The node simplification is performed by using two-level logic minimization techniques, and the don't cares which arise due to the structure of the multilevel logic are used in the two-level minimization process. The parallel algorithms for all of those transformations have been developed, and they have been integrated into a single system.

The parallel algorithms developed for the Transduction method as well as the MIS approach have three major contributions. First, the parallel algorithms developed use an *asynchronous* message-driven computing model with no synchronizing barriers separating phases of parallel computation. Priorities are assigned to different transformations to guide the synthesis procedure. Second, our algorithms are *portable* across a wide variety of parallel architectures, both *shared* and *distributed* memory machines. Finally, our algorithms are built around well-defined *sequential algorithm interfaces*, so that the parallel algorithm can benefit from the future expansion of the sequential algorithms. For example, we have *reused nearly 80%* of the sequential code from the MIS synthesis program in our parallel implementation.

The parallel synthesis algorithms based on the Transduction method and the MIS approach, described in this thesis, cannot be applied to very large circuits as a whole due to very high memory requirements. For those circuits, one has to partition the circuit and synthesize the partitions independently. Since the partitions are synthesized independently, they can be synthesized in parallel by using multiple processors. We have developed a new partitioning algorithm which is suitable for partitioning a given logic circuit such that the partitions are of almost equal size while maintaining the gross structural features of that circuit. We have developed a parallel synthesis system based on this partitioning approach. We have also devised an iterative approach which performs synthesis in different phases to improve the quality of the synthesized circuit.

5.2 Future Research Directions

In the ProperSYN system described in Chapter 2 of this thesis, the output functions and the CSPFs are obtained on a demand-driven basis. All data have an "owner" processor, and whenever any other processor needs those data, it requests them from the "owner" processor. The "owner" processor sends a copy of the data to the processor which has requested them. As a result, many copies of the same data may be residing in different processors. Moreover, after a processor has consumed the data it has requested, it does not delete them from its memory because it may need them again. For large combinational circuits, the output functions and the CSPFs will be very large and will require much memory space. Due to multiple copies of the same data, we may not be able to handle large circuits because of the memory constraints. The problem of multiple copies of the same data can be solved up to certain extent by using caching techniques, in which the BDDs of a particular part of the network will be permanently stored in that processor which "owns" that part of the network. If any other processor needs those BDDs, it will first look for them in its local "cache" and, if not found, ask the "owner" processor to send them. If the local "cache" size is small, the number of extra copies of a BDD will be small. In that case, we will be able to handle larger-sized circuits than we can handle with our present implementation. To find out the optimal size of the cache, some research has to be performed. Also, the cache replacement policy has to be tuned to the synthesis algorithm for better performance.

To handle really large circuits, one has to take more drastic actions. The circuit can be levelized and the output functions and the CSPFs can be stored in memory only for the nodes in every Lth level, i.e., the output functions and CSPFs can be stored for the nodes in levels k, $L + k$, $2L + k$, $3L + k$, etc., where k is a parameter for the program. If the output function or the CSPF of an intermediate node is needed at some phase of computation, those data can be recomputed starting from the level at which data are available. One needs to recompute at most L - 1 levels to obtain any data. It is also required to have a small cache to store the recently recomputed data to avoid repeated recomputation of the same data. Clearly, the cache replacement algorithm has to be tuned to the synthesis algorithm for better performance. The larger the value of L, the smaller is the memory requirement, but the larger is the run time due to more recomputations. Some research has to be performed in this direction to observe the performance of such a system.

The use of a *master* processor in ProperSYN and ProperMIS prevents them from becoming truly scalable systems. For a large number of processors, the procedure of asking permission from *one master* processor will become the sequential bottleneck of the parallel program. Hence, a *multiple masters* model has to be developed to avoid that problem, a problem which is very difficult due to the fact that the effect of a logic transformation is propagated all the way to the primary inputs and the primary outputs. As a result, it is very difficult to assign multiple *master* processors, one for each "zone" of the circuit. It is a very interesting future research problem.

In this thesis, we have looked only at the combinational logic synthesis algorithms and developed parallel algorithms for them. In the future, the algorithms for sequential logic synthesis have to be investigated for the purpose of parallelization. The problems of optimal state assignment, optimal state minimization, etc., are computationally very expensive. Hence, parallel algorithms have to be developed for those problems as well.

REFERENCES

[1] P. Banerjee, "Parallel Algorithms for VLSI CAD," *Tutorial Proceeding, Design Automation Conference*, 1988.

[2] P. Banerjee, M. H. Jones, and J. S. Sargent, "Parallel Simulated Annealing Algorithms for Standard Cell Placement on Hypercube Multiprocessor," *IEEE Transactions on Parallel and Distributed Systems*, pp. 91–106, January 1990.

[3] R. J. Brouwer and P. Banerjee, "PARAGRAPH: A Parallel Algorithm for Simultaneous Placement and Routing using Hierarchy," *European Design Automation Conference (EDAC-92)*, March 1992.

[4] A. Cossoto, F. Romeo, and A. Sangiovanni-Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells," *IEEE Transcations on Computer-Aided Design*, pp. 838–847, September 1987.

[5] R. M. Kling and P. Banerjee, "ESP: Placement by Simulated Evolution," *IEEE Transcations on Computer-Aided Design*, pp. 245–256, March 1989.

[6] S. C. Chang and J. JaJa, "Parallel Algorithm for Channel Routing in the Knock-Tree Model," *International Conference on Parallel Processing*, pp. 18–25, August 1988.

[7] K. P. Belkhale and P. Banerjee, "Parallel Algorithms for VLSI Circuit Extractions," *IEEE Transcations on Computer-Aided Design*, pp. 604–618, May 1991.

[8] G. Bischoff and S. Greenberg, "CAYNNE: A Parallel Implementation of the Circuit Simulator SPICE," *International Conference on Computer-Aided Design*, pp. 182–185, November 1986.

[9] G. K. Jacob, A. R. Newton, and D. E. Pederson, "An Empirical Analysis of Performance of a Multiprocessor-based Circuit Simulator," *23rd Design Automation Conference*, pp. 588–593, June 1986.

[10] J. M. Arnold and C. T. Terman, "A Multiprocessor Implementation of a Logic Level Timing Simulator," *International Conference on Computer-Aided Design*, pp. 116–118, November 1985.

[11] E. H. Frank, "Exploiting Parallelism in a Switch-Level Simulation Machine," *Design Automation Conference*, pp. 209–215, June 1986.

[12] S. Patil and P. Banerjee, "A Parallel Branch and Bound Algorithm Approach to Test Generation," *IEEE Transcations on Computer-Aided Design*, pp. 313–322, March 1990.

[13] R. B. Mueller-Thuns, D. G. Saab, R. F. Damiano, and J. A. Abraham, "Portable Parallel Logic and Fault Simulation," *International Conference on Computer-Aided Design*, pp. 506–509, 1989.

[14] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, "ESPRESSO-II: A New Logic Minimizer for Programmable Logic Arrays," *Custom Integrated Circuits Conference*, pp. 370–376, June 1984.

[15] A. J. de Geus and W. Cohen, "A Rule-based System for Optimizing Combinational Logic," *IEEE Design and Test*, pp. 22–32, August 1985.

[16] R. Brayton, R. Ruddel, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A Multiple-level Logic Optimization System," *IEEE Transactions on Computer-Aided Design*, pp. 1062–1081, November 1987.

[17] X. Xiang, *Multilevel Logic Network Synthesis Systems, SYLON-XTRANS.* Ph.D. dissertation, Univ. of Illinois, 1990.

[18] K. A. Barlett, D. Bostick, G. Hachtel, R. Jacoby, and M. Lightner, "BOLD: A Muliple-level Logic Optimization System," *International Conference on Computer Aided Design*, 1987.

[19] R. Galivanche and S. M. Reddy, "A Parallel PLA Minimization Program," *Design Automation Conference*, pp. 600–607, 1987.

[20] G. Zipfel, "Parallel Algorithm for Algebraic Factorization with Application to Multi-level Logic Synthesis," M.S. thesis, Univ. of Illinois, 1991.

[21] G. D. Hachtel and P. H. Moceyunas, "Parallel Algorithms for Boolean Tautology Checking," *ICCAD*, pp. 422–425, 1987.

[22] H. T. Ma, S. Devadas, and A. S. Vincentelli, "Logic Verification Algorithms and Their Parallel Implementations," *24th DAC*, 1987.

[23] C.-F. Lim, "Parallel Algorithm for Multi-Level Logic Synthesis Using the Transduction Method," M.S. thesis, Univ. of Illinois, 1991.

[24] C. F. Lim, P. Banerjee, K. De, and S. Muroga, "A Shared Memory Parallel Algorithm for Logic Synthesis," *The Sixth International Conference on VLSI Design*, January 1993.

[25] B. Ramkumar and P. Banerjee, "ProperCAD: A Portable Object-oriented Parallel Environment for VLSI CAD," *International Conference in Computer Design*, 1992.

[26] W. Fenton, B. Ramkumar, V. A. Saletore, A. B. Sinha, and L. V. Kale, "Supporting Machine Independent Programming on Diverse Parallel Architecture," *International Conference on Parallel Processing*, August 1991.

[27] L. V. Kale, "The Chare Kernel Parallel Programming System," *International Conference on Parallel Processing*, August 1990.

[28] B. Ramkumar and P. Banerjee, "A Portable Parallel Algorithm for VLSI Circuit Extraction," *International Parallel Processing Symposium*, 1993.

[29] B. Ramkumar and P. Banerjee, "Portable Parallel Test Generation for Sequential Circuits," *International Conference on Computer-Aided Design*, pp. 220–223, 1992.

[30] S. Kim, B. Ramkumar, and P. Banerjee, "ProperPLACE: A Portable Parallel Algorithm for Standard Cell Placement," Submitted to *International Symposium on Parallel Processing*, 1994.

[31] X. Q. Xiang and S. Muroga, "SYLON-XTRANS: A Multilevel Logic Network Synthesizer," *International Workshop on Logic Synthesis*, 1989.

[32] Y. Matsunaga and M. Fujita, "Multi-level Optimization using Binary Decision Diagrams," *ICCAD*, pp. 556–559, 1989.

[33] K. De, B. Ramkumar, and P. Banerjee, "ProperSYN: A Portable Parallel Algorithm for Logic Synthesis," *International Conference in Computer-Aided Design*, pp. 412–416, 1992.

[34] K. De, B. Ramkumar, and P. Banerjee, "A Portable Parallel Algorithm for Logic Synthesis using Transduction," Submitted to *IEEE Transactions on Computer-Aided Design*, 1992.

[35] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The Transduction Method - Design of Logic Networks based on Permissible Function," *IEEE Transactions on Computers*, pp. 1404–1424, October 1989.

[36] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transductions on Computer*, pp. 677–691, August 1986.

[37] K. A. Barlett and et. al, "Multilevel Logic Minimization using Implicit Don't Cares," *IEEE Transactions on Computer-Aided Design*, pp. 723–740, June 1988.

[38] H. Savoj, H.-Y. Wang, and R. K. Brayton, "Improved Scripts in MIS-II for Logic Minimization of Combinational Circuits," *International Workshop on Logic Synthesis*, 1991.

[39] K. De and P. Banerjee, "Logic Partitioning and Resynthesis for Testability," *International Test Conference*, 1991.

[40] K. De and P. Banerjee, "Parallel Algorithms for Logic Synthesis using MIS Approach," Submitted to *Design Automation Conference*, 1994.

[41] K. De and P. Banerjee, "ProperMIS: Asynchronous Portable Parallel Algorithms for Factoring and Simplification in Logic Synthesis," Submitted to *IEEE Transactions on Computer-Aided Design*, 1993.

[42] R. L. Ruddel, *Logic Synthesis for VLSI Design.* Ph.D. dissertation, University of California, Berkeley, 1989.

[43] R. Brayton, R. Ruddel, A. Sangiovanni-Vincentelli, and A. Wang, "Multilevel Logic Optimization and the Rectangular Covering Problem," *International Conference on Computer Aided Design*, pp. 66–69, 1987.

[44] A. Saldhana, A. Wang, R. Brayton, and A. Sangiovanni-Vincentell, "Multilevel Logic Simplification Using Don't cares and Filters," *26th Design Automation Conference*, pp. 277–282, 1989.

[45] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., San Fransisco, California, 1979.

[46] A. Malik, R. Brayton, A. R. Newton, and A. Sangiovanni-Vincentelli, "Reduced Offsets for Minimization of Binary-valued Functions," *IEEE Transactions on CAD*, pp. 413–426, April 1991.

[47] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, vol. 49, pp. 291–307, 1970.

[48] C. M. Fiduccia and R. M. Mattheyses, "A Linear Time Heuristic for Improving Network Partitioning," *Proc. 19th Design Automation Conference*, pp. 175–181, 1982.

[49] P. R. Suaris and G. Kedem, "Quadrisection: An Approach to Standard Cell Leyout," *International Conference on Computer-Aided Design*, pp. 474–477, 1987.

[50] Y.-C. Wei and C.-K. Cheng, "Ratio Cut Partitioning for Hierarchical Designs," *IEEE Transactions on CAD*, pp. 911–921, July 1991.

[51] C. W. Yeh and C. K. Chung, "A General Purpose Multiple way Partitioning Algorithm," *Design Automation Conference*, pp. 421–426, 1991.

[52] H. Cho, G. Hachtel, M. Nash, and L. Setiono, "BEAT_NP: A Tool for Partitioning Boolean Networks," *Proc. International Conference on Computer Aided Design*, pp. 10–13, 1988.

[53] S. Dey, F. Berglez, and G. Kedem, "Corolla Based Circuit Partitioning and Resynthesis," *27th Design Automation Conference*, pp. 607–612, 1990.

[54] J. G. Udell and E. J. McCluskey, "Efficient Circuit Segmentation for Pseudo-exhaustive Test," *International Conference on Computer-Aided Design*, pp. 148–151, 1987.

[55] W. E. Donath, "The Partitioning of Computer Logic," in *Advances in VLSI CAD* (G. N. Rabbat, ed.), 1984.

VITA

Kaushik De was born in India, in 1965. He received his B. Tech (Hons.) degree in Electronics and Electrical Communication Engineering from the Indian Institute of Technology, Kharagpur, India, in 1987. He received his M.S. degree in Electrical Engineering from the University of Illinois at Urbana-Champaign, in 1991. He was employed with Texas Instruments from 1987 to 1989. Since 1989, he has been a research assistant with the Center for Reliable and High-Performance Computing at the University of Illinois. During the summer of 1991, he was employed by the IBM T. J. Watson Research Center, Yorktown Heights, New York.

After completing his doctoral dissertation, he will be joining LSI Logic Corporation, Milpitas, California. His research interests include Synthesis, Testing, Computed-Aided Design and Parallel Processing.

Kaushik De was recipient of the Institute Silver Medal from the Indian Institute of Technology, Kharagpur, India.