PARALLEL ALGORITHMS FOR ALGEBRAIC FACTORIZATION
IN LOGIC SYNTHESIS

BY

SUMIT ROY

B.Tech., Indian Institute of Technology, Kharagpur, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

ABSTRACT

Algebraic factorization is an extremely important part of any logic synthesis system. The time requirements for algebraic factorization for large circuits are quite large; hence it is important to look at parallel processing to speed up the procedure. Unfortunately, since the algebraic factorization procedure is inherently sequential in nature, it is not at all obvious how one should proceed in parallelizing this key step in logic synthesis. This thesis presents a detailed study of three different parallel algorithms for algebraic factorization based on kernel extraction. The first algorithm uses circuit replication and uses a divide-and-conquer strategy to follow the same search path as in the sequential algorithm. Although it gets comparable quality to the serial algorithm, it suffers from poor speedup. A second algorithm uses totally independent factorization on different circuit partitions with no interactions among the partitions. This algorithm provides large speedups but suffers in quality as the number of partitions increases. A third algorithm represents a compromise between the above approaches. It uses a novel L-shaped partitioning strategy that provides some interaction among the rectangles obtained in various partitions. It produces better results than the parallel algorithm using independent partitions but produces lower speedups. For a large circuit like ex1010, the last algorithm runs 11.5

times faster than the sequential kernel extraction algorithms of SIS on 6 processors with less than 0.2% degradation in the quality of the results. Also, our parallel algorithms could be extended to timing-driven and low-power-driven synthesis if they are based on rectangle covering.

DEDICATION

To my parents and my brother

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Prithviraj Banerjee, for his advice and assistance during this work.

I would like to express my thanks to the members of the ProperCAD and Paradigm projects both for their technical help in issues of parallel computing and their camaraderie. I would especially like to thank John Chandy, my officemate, who helped me learn how to program on the ProperCAD platform for which there was little documentation.

I owe a great debt to my parents for their invaluable love and understanding during my graduate education.

This work was supported in part by the Semiconductor Research Corporation and in part by the Advanced Research Project Agency. The National Center for Supercomputer Applications provided support by providing access to their computing resources.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Motivation

Multilevel logic synthesis is a very important phase of VLSI system design. Efficient implementations for multilevel logic optimization include SOCRATES [1], SYLON-XTRANS [2], BOLD [3] and MIS [4]. The algorithms for logic transformations used in the MIS/SIS [5] system form the core of numerous university and industrial logic synthesis systems. Many of the transformation operations used in modern synthesis systems are computationally expensive. For large circuits used in industry today, the synthesis times can extend to several hours. In view of the availability of multiprocessor workstations by numerous vendors, parallel processing is fast becoming a desirable solution to reduce the large amount of time spent in VLSI circuit synthesis.

The MIS/SIS synthesis tool is based on algebraic factorization and simplification of nodes in a Boolean network [4]; each of these operations is computationally expensive. Algebraic factorization forms the core of the tool and is invoked several times during the

Table 1.1: Run times of several circuits and the time it takes in the kernel extraction routine.

MCNC Circuits	Size (Lit. Count)	Factorization Invoked	Total Factorization Time (in seconds)
dalu	3588	10	34.2
seq	17938	16	295.1
des	7412	10	425.7
spla	24087	12	1771.4
ex1010	14972	9	5841.7

synthesis. Table 1.1 shows the number of times the algebraic factorization procedure is called in a typical script from the SIS package [5] and the times spent in performing these factorizations. Hence, we need to speed up the factorization operation by using parallel processing.

The problem of algebraic factorization can be explained through the following example.

Example 1.1 *Let* $\{\mathcal{N} = F,\ G,\ H\ \}$ *be a combinational network.*

$$\begin{aligned} F &= af + bf + ag + cg + ade + bde + cde, \\ G &= af + bf + ace + bce, \\ H &= ade + cde \end{aligned} \tag{1.1}$$

In the network, $\mathcal{N}$, a+b is a common factor that can be factored out from F and G to produce a sum of products expression with eight fewer literals. The new expressions for

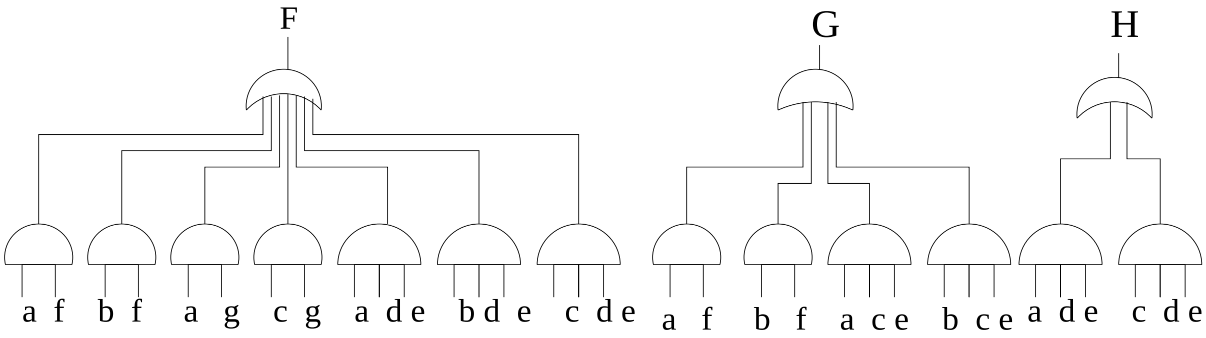

Figure 1.1: Circuit for Equation (1.1).

F and G are shown below:

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

$$G = fX + ceX,$$

$$X = a + b$$

The savings in area can be seen from the circuit before and after the factoring operation in Figures 1.1 and 1.2, respectively. Based on the estimation that a gate with n inputs requires $n + 1$ units of area, we observe a reduction from 62 to 47 units in the factorized circuit. By repeatedly performing the factorization on a large network, it is possible to obtain great reductions, but it is computationally expensive. Thus, we will use parallel processing to speed up the process.

The work described in this thesis is part of a larger project dealing with developing parallel algorithms for various operations in logic synthesis. In this work, we focus on algebraic factorization using the kernel extraction procedures in the MIS/SIS system. Since the kernel extraction process is inherently sequential in nature, it is not at all

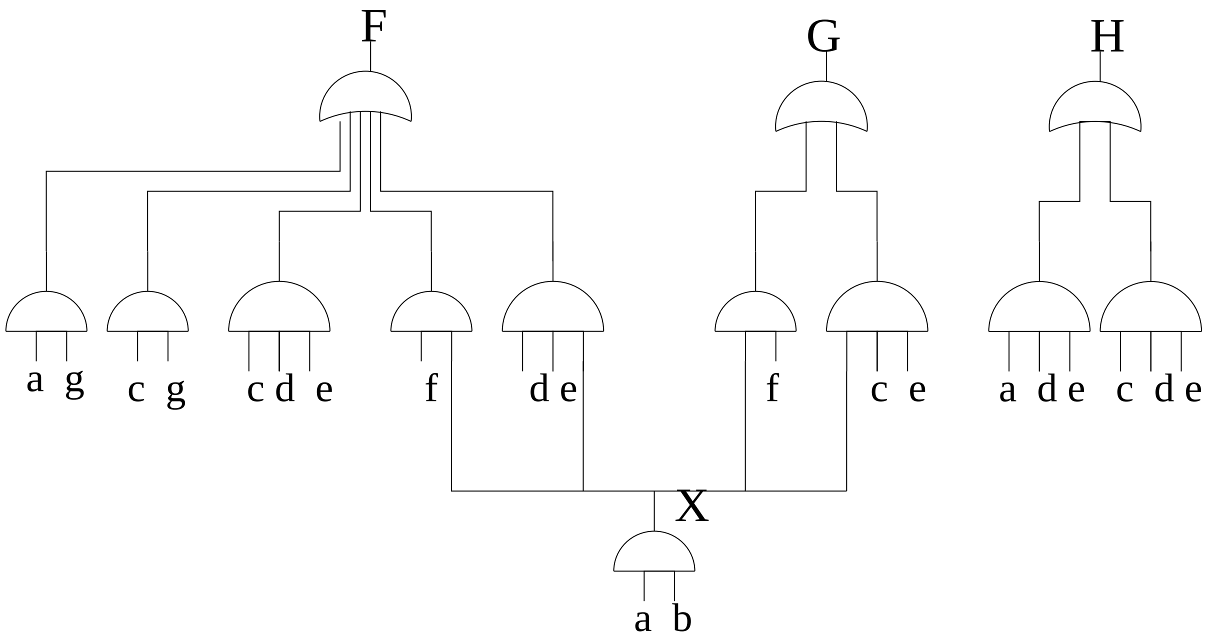

Figure 1.2: The above circuit after factoring out $a + b$.

obvious how one should proceed in parallelizing this key procedure in logic synthesis. This thesis presents a detailed study of various parallel algorithms for kernel extraction.

1.2 Previous Research

Recently, several approaches have been reported on developing parallel logic synthesis algorithms. Galivanche and Reddy [6] have proposed a parallel implementation of a 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 seven on an 8-processor shared memory multiprocessor. Zipfel has proposed a parallel multilevel synthesis algorithm using algebraic factorization procedures of MIS in [7]. He has developed parallel algorithms for kernel extraction and cube extraction procedures within the MIS synthesis programs; however, the results of the speedups on an 8-processor shared memory multiprocessor were less than two on eight processors. Hachtel and Moceyunas [8] have proposed a parallel tautology checking algorithm with applications to logic verification. Ma et al. [9] have proposed parallel logic verification algorithms based on implicit enumerations. Both of the above works on parallel logic verification were proposed for shared memory multiprocessors and reported good speedups. Lim et al. have proposed a parallel logic synthesis algorithm using the transduction method [10], [11], suitable for execution on a shared memory multiprocessor. Speedups of about 6 were reported on an 8-processor Encore multiprocessor.

In another study, De, Ramkumar and Banerjee [12] reported on a parallel algorithm for synthesis using the transduction approach called ProperSYN. Parallel approaches were described for the key steps in transduction, namely, function evaluation, pruning, gate merging, gate substitution, and others. This approach did not consider the algebraic factorization operation that is addressed in this work. An obvious way of speeding up the algebraic factorization and the other techniques of a logic synthesis application is to generate a large number of logic partitions of a given circuit and to synthesize each partition independently. The results of the individual partitions can then be merged back. This approach was presented by De and Banerjee in ProperPART [13]. Unfortunately, such an approach has the problem that with an increasing number of partitions, even though the synthesis time goes down, the quality of the overall network degrades. The degradation occurs because each synthesis procedure of a partition only synthesizes within a partition by treating it as an independent block. It does not take any global information into consideration during minimization. De and Banerjee also reported a novel iterative approach to parallel synthesis where portions of a circuit are repartitioned and resynthesized along different sets of processors in an implementation called ProperPART [13]. It was shown that the overall synthesis quality is significantly improved by this iterative repartitioning and resynthesis approach over the original partitioned approach without any interactions. The study did not isolate the effects of parallel algebraic factorization in this partitioned approach.

A recent study that specifically addressed the parallel algorithms for synthesis based on the MIS approach was reported by De et al. in an approach called ProperMIS [14]. ProperMIS provided parallel approaches to the kernel extraction, cube extraction, node simplification, and resubstitution operations of the MIS system. For the parallel kernel extraction step, ProperMIS divides up the search space across different processors but retains the global information of the circuit in all of them. Hence, it requires synchronization and update after every step of factorization. Although the global view produces comparable quality to that obtained by MIS, ProperMIS suffers from poor speedups and memory scalability.

While all of the previous approaches described parallel algorithms for logic synthesis operations in an integrated framework, it was unclear whether each of the basic operations was being successfully parallelized. It is important to perform this analysis since often in a real design environment, a user will perform many interactive operations on a large design by invoking different operations in different orders. Each of the synthesis operations needs to be parallelized individually. In this thesis, we therefore perform a detailed study of three parallel algorithms for algebraic factorization of a combinational network. The different parallel approaches studied are:

- a parallel algorithm that assumes replicated circuit information on all of the processors and performs a divide-and-conquer search operation across the processors,
- a parallel algorithm using kernel extraction on independent circuit partitions without any interaction among the partitions and
- a parallel algorithm on a partitioned circuit representation that allows interactions among the partitions.

The last two approaches are memory scalable, in that it is possible to perform synthesis operations on circuits that run out of memory on a single processor. Memory scalability is extremely important for large logic designs that are designed by industry. Before describing the above algorithms, we will review the serial algorithm for algebraic factorization in Chapter 2

1.3 Thesis Outline

This thesis is organized as follows. In Chapter 2, we review the sequential algorithm for algebraic factorization in MIS. The following three chapters describe the three algorithms for partitioned kernel extraction. Chapter 3 describes an approach using replicated circuit structure but a divide-and-conquer search. A scalable parallel kernel extraction algorithm without any interaction between the processors is described in Chapter 4. An unique approach using L-shaped partitioning of the co-kernel cube matrix is described in Chapter 5. We conclude the thesis in Chapter 6 by comparing the various parallel approaches and provide directions for future research in this area.

2. OVERVIEW OF SERIAL ALGEBRAIC FACTORIZATION

2.1 Definitions

In this section, we will review some basic definitions as given in [4] to be used later in this thesis.

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 $x \in C \Rightarrow \overline{x} \notin C$. (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 $a + b\overline{c}$.

The *support* of an expression f is the set of literals $sup(f)$ that are present in the expression of f.

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

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

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

In other words, the kernels of an expression f are the cube-free primary divisors of f. The cube C used to obtain kernel $k = f/C$ is called the *co-kernel* of k.

The *co-kernel cube matrix* is a sparse matrix with rows representing co-kernels and columns representing kernel-cubes. A nonzero element A(i,j) of the matrix corresponds to a cube of the function formed by the union of co-kernel i and kernel-cube j.

Example 2.1 *Let* $\{\mathcal{N} = F,\ G,\ H\ \}$ *be a combinational network.*

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

$$G = af + bf + ace + bce,$$

$$H = ade + cde \tag{2.1}$$

In the above network, a, b, c, …are variables, a, a', b, b', …are literals, and **af, bf, ade, …** are cubes. The *support-set* of G is *a, b, c, e* and *f*, while *F, G and H* are all cube-free since no cube divides them evenly. The circuit is shown in Figure 1.1. The kernels (and co-kernels) of **G** are ce+f(a, b) and a+b(f,ce). The co-kernel cube matrix

			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	.	.	.	.

Figure 2.1: Co-kernel cube matrix for Equation (2.1).

for the above network is shown in Figure 2.1, where the top two rows denote the cubes of the kernel and their indices. Columns 2 and 3 show the co-kernels (*e.g., a, b, de, ...*) and their indices (*1, 2, 3, ...*), whereas column 1 denotes the associated functions (F, G, H). The entry at (3, 2) is the index of the cube **bde** for function F, and all places where 6 appears, e.g., (2,4) in the matrix, correspond to the same cube **bde**.

The network, $\mathcal{N}$, in Example 2.1 will be used by all of the examples in the rest of the thesis. For reference, the cubes of the original expressions in Equation (2.1) are numbered from 1 to 13, i.e., $af = 1$, $bf = 2$, $\ldots$, $cde = 13$.

2.2 Algebraic Factoring

Algebraic factoring takes a sum of product (SOP) realization of a circuit as its input, searches for common subexpressions and extracts these subexpressions to produce an

SOP with a smaller literal count. The gain in the literal count of the SOP is illustrated in Example 1.1. The basic technique consists of two steps:

1. find the kernel, i.e., the subexpression and

2. factor out this subexpression from the SOP.

In the rest of this thesis, we will refer to step 2 as *dividing* the network by the kernel or *extracting* the kernel from the network, and hence the above problem of algebraic factorization is known as *kernel extraction.* The kernel extraction problem reduces to applying these two steps, namely kernel matrix creation and kernel extraction, repeatedly in an order so as to minimize the final SOP representation. In [15] this optimization problem is mapped onto a *minimum-weighted rectangle-covering problem.*

A rectangle (R, C) of the co-kernel cube matrix B, where $b_{ij} \in \{0, 1, *\}$, is a subset of rows R and a subset of columns C such that $b_{ij} \in \{1, *\} \forall i \in R, j \in C$. For example, {(3,4,9,10)(1,2)} is a rectangle, but {(3,4)(1,2,3)} is not. The **value** of a rectangle is the savings in the literal count if the kernel represented by it is extracted from the network. The value of the above rectangle is eight, which indicates that if the kernel $(a + b)$ is extracted from $\mathcal{N}$, it will have eight fewer literals in its SOP representation. The *minimum-weighted rectangle-covering problem* is to find a set of rectangles (not necessarily disjoint) with minimum total weight such that all 1's are covered. In the logic synthesis context, a rectangle in the co-kernel cube matrix corresponds to a set of cubes that can be divided by the kernel, where the kernel is the set of cubes of the column C and signifies the common subexpression. The weight of the rectangle *(R, C)* is the total number of literals in the network required to represent these cubes after extracting the kernel from the network and is defined as

$$w(R,C) \; = \sum_{i \in C} w_i^c + \sum_{j \in R} (w_j^r + 1) \tag{2.2}$$

where the first term sums up the literals in all of the cubes in the kernel, and the latter adds up the literals in the co-kernels. An extra literal count on the second summation corresponds to the new variable replacing the kernel. The value of a rectangle measures the gain in literal count if the kernel represented by it is extracted from the network. If V_{ij} is the number of literals in cube $b(i,j)$, then the value of the rectangle *(R, C)* is defined by Equation (2.3). The weights and the values of a rectangle are best demonstrated by Example 2.2.

$$v(R,C) = \sum_{i \in R, j \in C} V_{ij} - w(R,C) \tag{2.3}$$

Example 2.2 *Let* (R,C) *be the rectangle* {*(3,4,9,10)(1,2)*} *in the matrix in Figure 2.1.* The cubes (indices of the cubes) of the rectangle are af(1), bf(2), ade(5), bde(6) of function F and af(8), bf(9), ace(10), bce(11). The kernel of the rectangle is $a + b$ represented by column C = (1,2), and the co-kernels are de, f of F and ce, f of G.

$$w(R,C) = \sum(1+1) \; + \; \sum((2+1)+(1+1)+(2+1)+(1+1)) \; = \; 12$$

$$v(R,C) = \sum(2+2+3+3+2+2+3+3) \; - \; 12 \; = \; 8$$

Thus the value of the rectangle is eight literals, which indeed reflects the gain in the literal count if we extract the kernel, $a+b$, from the network $\mathcal{N}$.

Thus the problem of kernel extraction reduces to finding a rectangle cover of the co-kernel cube matrix with maximum total value. If the rectangles are not disjoint, then the above algorithm is called *overlapping kernel extraction* in [15], and this approach is used in MIS. The algorithm is given below in pseudo C code.

```
KERNEL_EXTRACTION(network N)
do
        B ← generate_co_kernel_cube_matrix(N)
        value ← perform_kernel_extraction(B)
   while (value > 0)
```

```
PERFORM_KERNEL_EXTRACTION(co_kernel_cube_matrix B)
best_rect ← NONE
while (best_rect→value > 0)
        while (generate_next_rectangle(B, rect) ≠ NO_MORE)
                if   (rect→value > best_rect→value)
                        best_rect ← rect
        divide_network_by_kernel(best_rect)
        value ← value + best_rect→value
return value
```

The *generate_co_kernel_cube_matrix* generates all kernels from the network N to create B. A greedy approach is used to find the rectangle cover for B in *perform_kernel_extraction*. It generates all prime rectangles and keeps extracting the rectangle with the largest value. The kernel from the best rectangle is extracted from the network in *divide_network_by_kernel*, and the values of all of the cubes of the best rectangle are set to zero in the co-kernel cube matrix, indicating that they are covered. The process of selecting a good rectangle and extracting the rectangle is iterated while the value of the best rectangle is positive, i.e, until no factor further reduces the number of literals in the network.

Example 2.3 *Let $\mathcal{N}$ of Example 2.1 be the given network. The co-kernel cube matrix B is given in Figure 2.1.*

The kernel extraction proceeds in the following manner. In the first iteration, the best rectangle found is {(3,4,9,10)(1,2)} with value eight, and the kernel X = a+b is extracted. The values of the cubes 1, 2, 5, 6, 8, 9, 10 and 11 are made zero. The next best rectangle is {(3,6,11)(1,3)} with value three, which overlaps with the previous rectangle. The kernel Y = a+c is extracted. The algorithm stops since no more rectangles with positive gain can be found, and the transformed network $\mathcal{N}$ has 22 literals instead of 33. The resulting circuit is shown in Figure 2.2, and the new network is given below.

$$F = deX + fX + deY + gY,$$

$$G = ceX + fX,$$

$$H = deY,$$

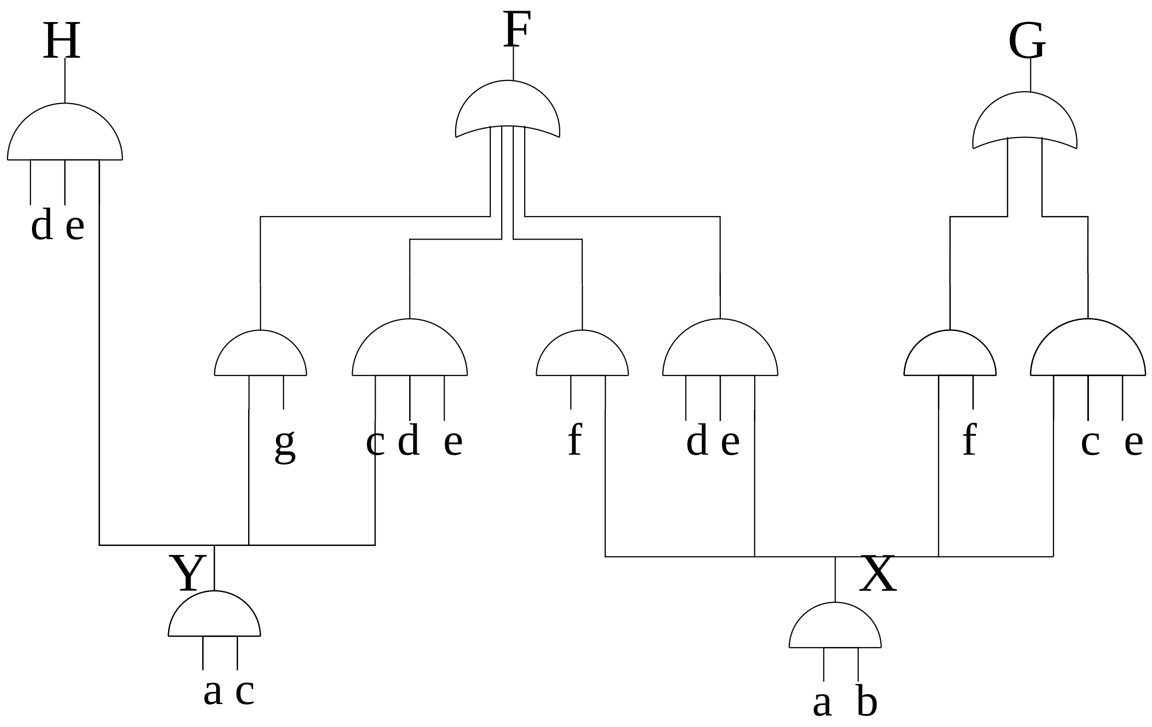

Figure 2.2: Resultant circuit after kernel extraction.

$$
\begin{aligned}
X &= a + b, \\
Y &= a + c
\end{aligned}
\tag{2.4}
$$

2.3 Summary

In this chapter we reviewed some basic definitions required in the rest of the thesis. We explained the serial kernel extraction algorithm in MIS/SIS, which will be the underlying framework for all of the parallel kernel extraction algorithms discussed in the thesis. The next three chapters describe three algorithms for parallel kernel extraction.

3. PARALLEL KERNEL EXTRACTION USING REPLICATED CIRCUIT

In this chapter, we describe an algorithm based on the approach presented in [14], which assumes replicated circuit information in all of the processors and follows the same search path as that in MIS. It derives concurrency by subdividing the search space of the optimization problem. We have leveraged off the parallel kernel extraction algorithm from the integrated parallel synthesis tool providing other logic synthesis techniques such as simplification, cube extraction and resubstitution. We have studied it for the purpose of a comparison to the other two parallel algorithms described in this thesis.

3.1 Parallel Algorithm

The circuit as well as the entire co-kernel cube matrix is replicated in all of the processors. The nodes of the circuit are conceptually partitioned among the processors to divide the task of generating the co-kernel cube matrix. All processors generate kernels for their nodes and broadcast them to the rest of the processors. Upon receiving them, a processor adds these kernels to its copy of the co-kernel cube matrix. With the help of

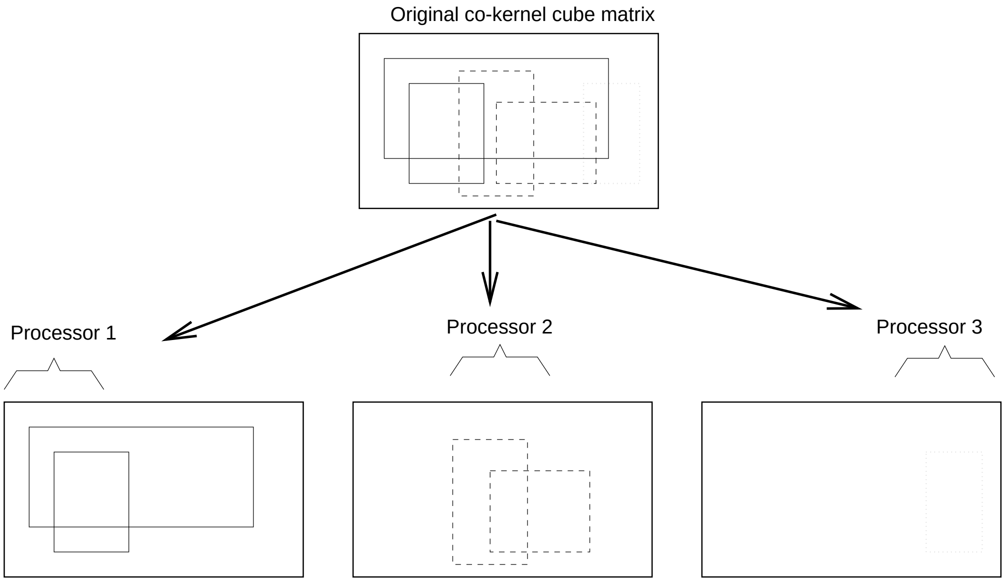

Figure 3.1: Decomposing a search space for finding the maximum-valued rectangle among processors. Processor 1 gets rectangles whose leftmost columns are in the left third of the area, processor 2 gets the second third, and so on.

a novel labeling scheme for the kernels, all of the processors succeed in obtaining exactly the same representation of the co-kernel cube matrix.

Once the co-kernel cube matrix has been generated, we have to perform a rectangle-cover on it, which involves iteratively generating all possible rectangles and selecting the maximum-valued rectangle. The search for the maximum-valued rectangle can be performed in parallel by a divide-and-conquer approach and is conceptually illustrated in Figure 3.1. It consists of a top-down traversal of the tree to generate all of the rectangles and their values. The best rectangle is then identified by each processor and propagated to the parent processor. The above search tree is distributed among the processors conceptually by restricting each processor to work with the rectangles starting from a certain column depending on its processor identity. The processor that owns the root of the search tree identifies the best rectangle and broadcasts it to all of processors. Each processor then goes on to divide its native copy of the matrix with the node and repeats the above process until it finds no more profitable rectangle. We describe the parallel algorithm below.

```
PARALLEL_KERNEL_EXTRACTION_REPLICATED(network N_i)
B_i ← generate_co_kernel_cube_matrix(N_i)
B = All_to_all_gather(B_i)
if  (myproc = MASTER_PROCESSOR)
    do
        best_rect ← find_best_rectangle(1, B.col)
        if  (best_rect→value > 0)
            divide_network_by_kernel(best_rect)   #PARALLEL
            B_temp ← generate_kernels_for_new_node(best_rect)
            B ← add_kernels_to_original_matrix(B, B_temp)   #PARALLEL
            Barrier_Sync()
    while (best_rect→value > 0)
```

```
FIND_BEST_RECTANGLE(start, end)
if  (Search_Expected(B, start, end) > WORK_THRESHOLD)
    mid = (start + end)/2
    best_rect_0 ← find_best_rectangle(start, mid)   #ANY_PROCESSOR
```

```
        best_rect_1 ← find_best_rectangle(mid, end)   #ANY_PROCESSOR
        if   (best_rect_0→value > best_rect_1→value)
             return best_rect_0
        else  return best_rect_1
    else  best_rect ← NONE
          while (generate_next_rectangle(B_recur, start, end, rect) ≠ NO_MORE)
                if   (rect→value > best_rect→value)
                     best_rect ← rect
          return best_rect
```

Although the circuit is replicated in all of the processors, each of them creates the kernels for only a portion of the nodes (in network N_i) that it owns. Each processor sends its own submatrix to others in an all to all gather to create its copy of the co-kernel cube matrix B corresponding to the entire network N. Once each processor has the entire matrix, the MASTER_PROCESSOR starts the search recursively for the best rectangle on B. The *find_best_rectangle* distributes the search among all of the processors in a divide-and-conquer fashion. Once the best rectangle is found, it is broadcast to all of the processors. Each processor extracts the rectangle from the entire matrix. The master processor also finds out the kernels for the new nodes created during the extraction. All of the processors then add it to its copy of the co-kernel cube matrix B. The above process continues until there is no profitable rectangle.

The *find_best_rectangle* procedure splits the matrix, B_{recur}, until the size of the matrix is less than a given threshold of work. It is invoked on the master processor but keeps subdividing and distributing it to other processors with the help of the ANY_PROCESSOR directive. The divide-and-conquer search terminates when the children processors have performed a reduction. Since the master processor started the recursion, it is the root of the recursion and has the best rectangle information at the end of the reduction.

3.2 Experimental results

The above parallel algorithm was implemented on a SUN SPARC server 1000E shared memory multiprocessor and tested on various MCNC BENCHMARK circuits. The kernel extraction operation was executed on 2, 4 and 6 processors, and the results are summarized in Table 3.1. The "Spd" column indicates how much faster the parallel executions are compared to the single processor run. For the circuits *spla* and *ex1010*, the program did not terminate after 10000 s, and some ran out of memory. The results demonstrate that the quality obtained by the multiprocessor runs is comparable to that obtained by a single processor run, which is expected since the above algorithm strictly follows the same search path. The poor speedup is due to the synchronizations after every extraction step and the redundant work required to maintain the consistency among the replicated circuit information in all of the processors. Also, the above algorithm is not scalable since the circuit and the co-kernel cube matrix are replicated on all processors. Our work aims

Table 3.1: Results of parallel kernel extraction using circuit replication. Time is in seconds and quality is in literal counts (LC).

MCNC	Initial	1 processor		2 processors			4 processors			6 processors		
Circuits	LC	Time	LC	Time	LC	Spd	Time	LC	Spd	Time	LC	Spd
dalu	3588	61.1	2139	41.9	2139	1.46	33.3	2139	1.83	31.1	2139	1.97
des	7412	478.9	6092	263.8	6092	1.82	164.9	6094	2.99	134.7	6092	3.56
seq	17938	8024.4	2633	4907.4	2650	1.64	3400.5	2632	2.36	3160.8	2633	2.54
spla	24087	-	-	-	-	-	-	-	-	-	-	-
ex1010	13977	-	-	-	-	-	-	-	-	-	-	-
	-	-	10864	-	10881	1.64	-	10885	2.39	-	10864	2.44

to relax the tight coupling existing between the data in all of the processors, thereby hoping to save on the redundant work.

3.3 Summary

In this chapter, we described a parallel kernel extraction algorithm that requires replication of the entire circuit information. Individual portions of the co-kernel cube matrix are created in parallel and then distributed to produce a copy of the entire matrix in each processor. The search for the best rectangle is then performed in a divide-and-conquer fashion to obtain concurrency. Once the best rectangle is found, each processor extracts the kernel, and the search continues until there are any profitable rectangles. The search path followed by the above algorithm is independent of the number of processors. Thus the quality of the result is comparable even if the number of processors increases.

But the above algorithm does not produce good speedups, and there are a couple of reasons for it. First, it performs redundant work while trying to maintain identical copies of the co-kernel cube matrix. Every processor extracts the best rectangle from the network, whereas in the serial code, it is extracted only once. Second, there is a communication overhead paid for sending updates done to any part of the circuit and broadcasts performed when a new kernel is generated. Finally, there is a synchronization that introduces idle time and hence reduces the efficiency of the algorithm.

Also, the algorithm is not scalable both due to memory and performance. Each processor has a copy of the entire circuit and the co-kernel cube matrix, which prevents memory scalability. Also, there is a serial bottleneck created by the master processor when it tries to create kernels from the new nodes. Thus the above algorithm is not expected to scale with the problem size. The next two chapters describe parallel approaches that try to prevent redundant work and achieve scalability.

4. PARALLEL KERNEL EXTRACTION WITHOUT INTERACTION

In this chapter, we perform kernel extraction on partitioned networks independently. Unlike the previous algorithm, which tries to parallelize the entire execution of the kernel extraction, we try to develop a parallel algorithm for the inner loop of *KERNEL_EXTRACTION* (statements 2 and 3) described in Chapter 2, thereby giving more flexibility to the end user.

Since no interaction is allowed between the various processors, a good circuit partitioning scheme is necessary to ensure quality control on synthesis of the partitioned circuits. Thus the main thrust of this chapter is on various circuit partitioning schemes that are suitable for producing good quality results on kernel extraction on the partitioned circuits only.

4.1 Inorder Partitioning

An obvious way to partition the circuit is to divide the circuit as it is read in so as to produce balanced partitions. If the total number of nodes are known, then we can trivially accomplish the above by assigning the nodes to a processor until the total number of nodes in that processor exceeds the average load for any processor. The rationale for such a partitioning scheme, apart from being very simple to implement, is that when designers describe their circuit in the form of a netlist, they start from one part of the circuit, create the netlist for that part and then move on to a different portion of the circuit. Thus by partitioning the circuit by its natural order, we extract the implicit locality information of the circuit. Figure 4.1 explains the concept. It gives a description of a circuit, *example*, in a blif format. For every node, the node names following the *.names* are the inputs to the node, and the last entity is the node name. For example, the first node name is A, and B and C are its inputs. The node represents $\mathrm{A} = \mathrm{B}\overline{C}$. The total number of nodes is 13. If we wish to partition the circuit among four processors, each block should have three nodes, except for the first block that has four nodes. While reading the circuit, the inorder partitioner puts the first 4 nodes into the first block, the next three into the next block and so on. The final partition is shown in Figure 4.1.

4.1.1 Experimental results

We partitioned the circuits using the following scheme and put the nodes in separate processors. We then perform kernel extraction on the partitioned networks independently on each of them. The run time and the quality for various MCNC circuits are shown below in Table 4.1 on a SUN SPARC Center 1000 shared memory multiprocessor. We used SIS to compare our results on one processor using the *gkx -bo1* option. Although

```
.model  example
.input L M N Q R
.output A B F G
```

```
.names C D A
10   1

.names D E B
1-   1
-0   1

.names O P C
- 1  1

..names O P D
10    1
01    1
```

```
.names  O P E
11    1

.names E H  F
00    1

.names H I  G
0-   1
-0   1
```

```
.names J P  H
11   1

.names J K   I
10   1
01   1

.names L M J
11  1
```

```
.names M N K
10   1

.names Q R O
11   1
00   1

.names Q R P
1-  1
-1  1
```

Figure 4.1: Inorder partition of a circuit in a blif format.

we got good speedups, there was a quality degradation of 2 to 3% as compared to the SIS results. The advantage of the method is that it is scalable, produces great speedups and is very simple to implement. However, we can see a degradation in the quality of the synthesized circuit, and it gets worse with increasing processors. As the number of processors increases, each partition is confined to a smaller portion of the entire circuit information and thus unable to make any global optimization. Also, since in many of the circuits the fanins of a node are not confined to its locality, the above locality assumption is no longer valid. Finally, when it comes to the end of a block, it chooses a node just by the order, without paying attention to whether there exists any other node that fans out to many nodes of this block. Hence, to overcome these drawbacks, we try a min-cut-based partitioning scheme in the next section.

Table 4.1: Results of parallel kernel extraction on a SUN SPARC Center 1000 using inorder partitioning.
Time is in seconds and quality is in literal counts (LC).

MCNC	Initial	SIS (1 proc)		2 processors			4 processors			6 processors		
Circuits	LC	LC	Time	LC	Time	Spd	LC	Time	Spd	LC	Time	Spd
dalu	3588	2837	16.5	2925	11.2	1.47	2988	5.72	2.88	3026	3.57	4.62
des	7412	6648	26.3	6651	14.0	1.87	6666	10.45	2.52	6650	7.45	3.53
seq	17938	9373	73.8	9350	60.6	1.22	9477	27.48	2.68	9427	21.26	3.47
spla	24087	17716	419.1	17984	289.3	2.20	18312	105.74	3.96	18750	264.75	6.63
ex1010	13977	11833	4508.9	12190	2138.0	1.50	12581	577.87	7.33	12810	392.87	24.28
average	1.00	0.723		0.733		1.65	0.747		3.87	0.756		8.51

4.2 Min-cut Partitioning

In a circuit partitioned approach to parallel kernel extraction, since we optimize the partitioned circuit only, we give up on the search for rectangles encompassing more than one partition. Thus the objective in this approach is to partition the circuit so as to minimize such overflowing rectangles. We solve this problem by mapping a circuit onto a weighted graph and then partition it to reduce overflowing rectangles. The vertices of the graph are the logic nodes of the network, and the hyperedges represent the fanin-fanout relationship between the nodes. The weight on a vertex is an estimate of the computation associated with the corresponding logic node. A previous study [13] has shown that the literal count of the SOP form of the node is a good approximation of the computation accompanying it; thus it is used as the weight of the node. Once we have derived the weighted graph from the circuit, we try to partition it so as to produce good quality results.

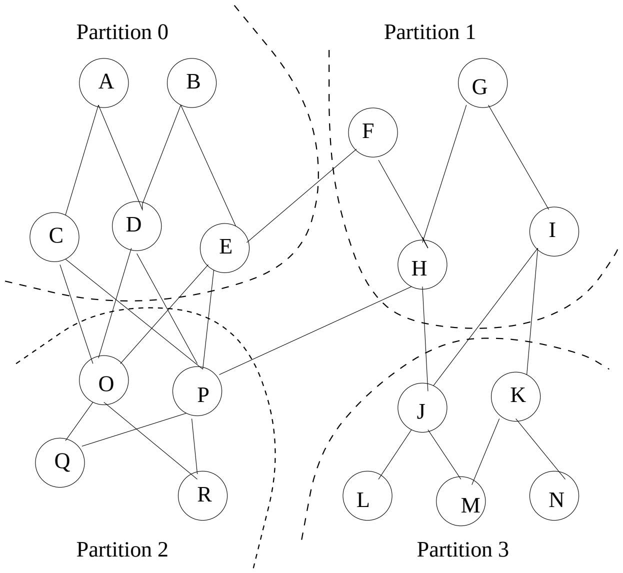

Figure 4.2: Min-cut graph partitioning of circuit in Figure 4.1 with multi-net cost.

A key observation in choosing the cost for the min-cut partitioning scheme is that if a rectangle is not enclosed within the submatrix of a processor, then there exist at least two vertices in two different processors having a common support set. Thus to minimize such overflowing rectangles, we need to partition the graph such that any hyperedge is confined to as few partitions as possible, ideally one partition. We use a variant of the linear-time network partitioning [16] developed by Sanchis [17] to decompose the logic network into blocks with a potentially balanced amount of computation. The cost function used for each net is equal to one less than the number of blocks it spans. Minimizing the cut set with this cost function tries to place nodes sharing the support set in the same block, thereby minimizing the rectangles from spilling. Figure 4.2 shows the partitioning for the circuit description of Figure 4.1, assuming each vertex has unit weight.

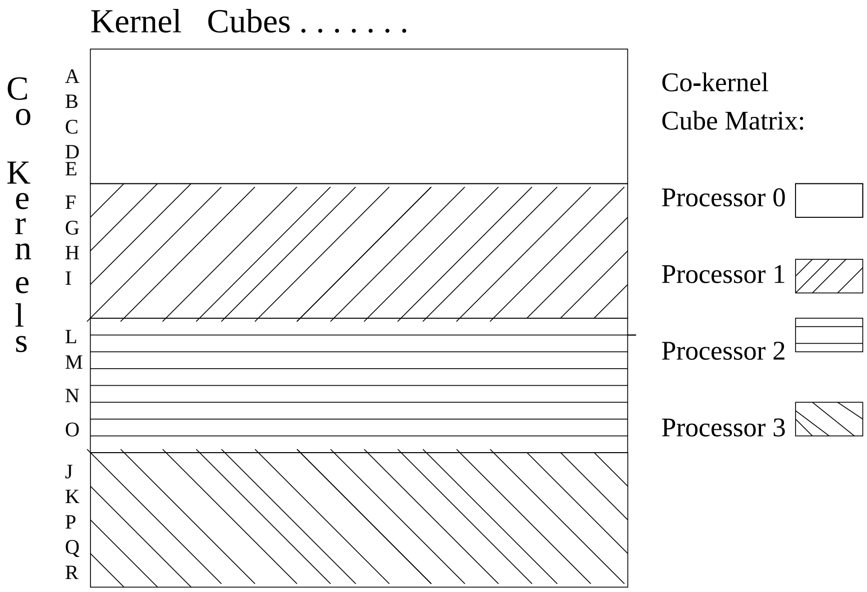

Figure 4.3: Conceptual search space for each processor.

After partitioning the graph, we distribute the nodes among the processors based on the partition. Each processor then goes on to create the co-kernel cube submatrix from its nodes. Finally, they independently perform kernel extraction on their own submatrix. The conceptual search space for each of the processors is shown in Figure 4.3 by the different shaded portions of the co-kernel cube matrix of the entire circuit. Example 4.1 illustrates the above idea.

Example 4.1 *Let Equation (2.1) be the given network.*

The above noninteractive partitioned kernel extraction proceeds by first mapping the network $\mathcal{N}$ onto weighted graph $\mathcal{G}$ with $(F(17), G(10)\ H(6))$ as its vertex (weight of the vertex). The two-way partitioned graph is shown in Figure 4.4.

Thus the balanced partition produced by the two-way network partition algorithm is $G_0 = (G, H)$ and $G_1 = (F)$. The partitioned circuit is shown in Figure 4.5. When

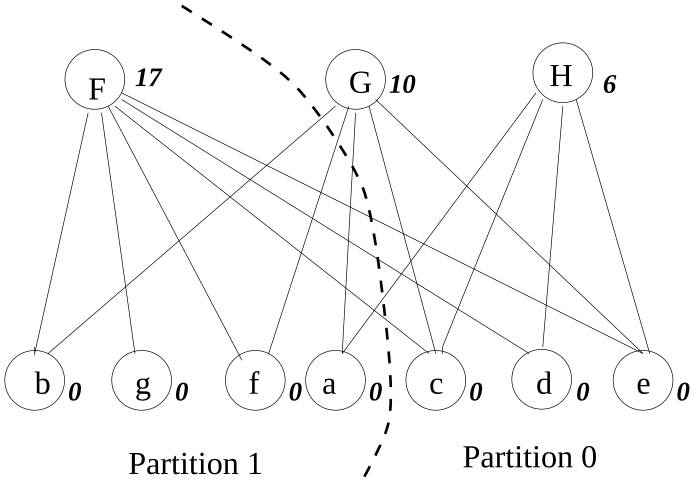

Figure 4.4: Partitioned graph for circuit in Figure 1.1.

the kernel matrix is created on the partitioned circuit, the submatrix for processor zero, B_0, corresponds to the bottom half of the co-kernel cube matrix in Figure 2.1 containing rows 7 to 11, whereas B_1 for processor one contains the upper half. Next we perform kernel extraction on the two submatrices B_0 and B_1, shown in Figure 4.6, separately, which leads to the following transformed network:

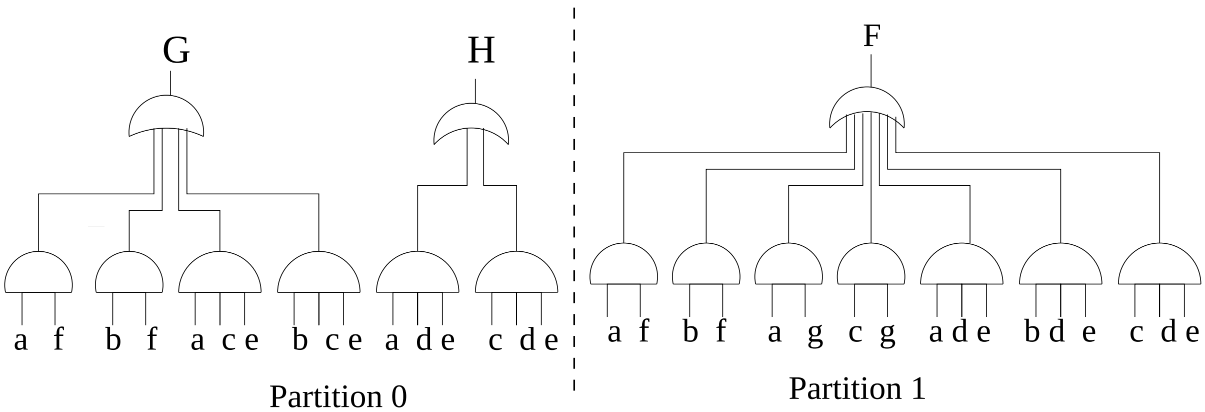

Figure 4.5: Partitioned circuit for Equation (2.1).

			a	b	c	de	f	g		
			1	2	3	4	5	6		
F	a	1	.	.	.	5	1	3	.	B
F	b	2	.	.	.	6	2	.	.	l
F	de	3	**5**	**6**	7	.	.	.	.	o
F	f	4	**1**	**2**	.	.	.	.	.	c
F	c	5	.	.	.	7	.	4	.	k
F	g	6	*3*	.	*4*	.	.	.	.	1

			a	b	c		f		ce	
			1	2	3		5		7	B
G	a	7	.	.	.	.	8	.	10	l
G	b	8	.	.	.	.	9	.	11	o
G	ce	9	**10**	**11**	.	.	.	.	.	c
G	f	10	**8**	**9**	.	.	.	.	.	k
H	de	11	*12*	.	*13*	.	.	.	.	0

Figure 4.6: Partitioned co-kernel cube matrix for Equation (2.1).

$$
\begin{aligned}
G &= ceZ + fZ, \\
H &= deY, \\
Z &= a + b, \\
Y &= a + c, \\
F &= deX + fX + ag + cg + cde, \\
X &= a + b
\end{aligned}
\tag{4.1}
$$

The quality degradation can be observed by comparing the circuits in Figures 1.1 and 4.7. The transformed network has 26 literals instead of 22 produced by the sequential

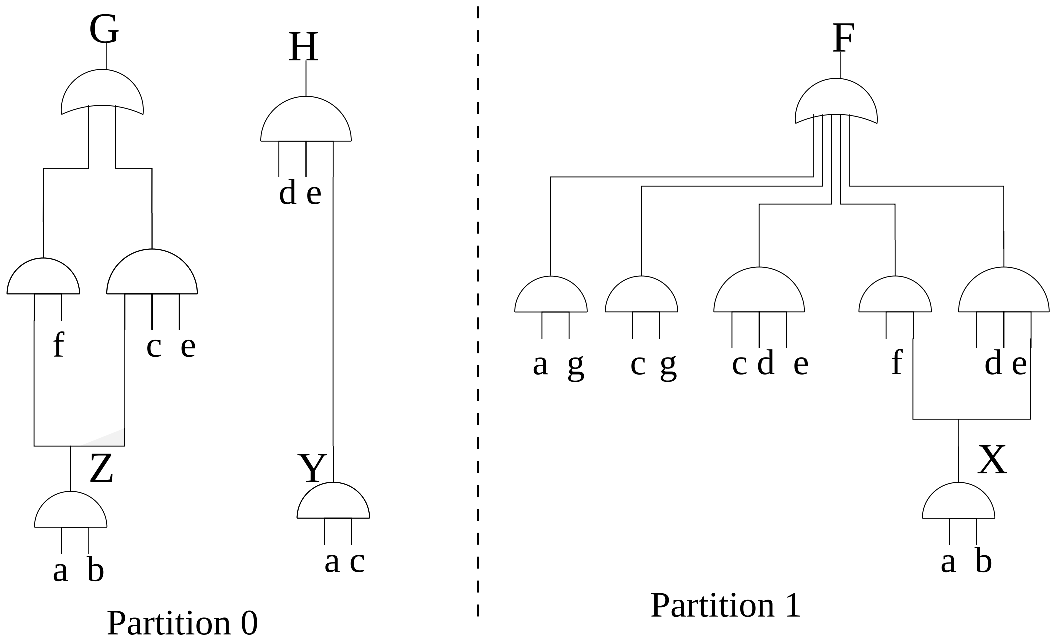

Figure 4.7: Factored network for the partitioned circuit in Figure 4.5.

kernel extraction routine in MIS. The quality degradation shows that performing kernel extraction on a graph-based partitioned circuit has certain drawbacks.

First, a rectangle encompassing more than one partition cannot be detected. Rectangle {(6,11)(1,3)} was not detected since it was not contained in any single partition. Although such a rectangle may be separately detected as sets of individual rectangles (like the rectangle {(3,4,9,10)(1,2)} being detected as {(3,4)(1,2)} and {(9,10)(1,2)}), it is very rare since the partial rectangles may not individually be the best rectangles in their partitions. The rectangle {(6,11)(1,3)} was not detected since the partial rectangle {(6)(1,3)} did not have a positive gain.

Another significant drawback is the duplication of kernels in the different partitions. Assume that a rectangle is split up between different partitions and that each sub-rectangle turns out to be the best rectangle for that block. Once the rectangles are extracted and the corresponding networks are divided, the different partitions will introduce separate nodes for the same kernel as that of the big rectangle. The rectangle {(3,4,9,10)(1,2)} was decomposed into two smaller rectangles {(3,4)(1,2)} and {(9,10)(1,2)}. Hence, the kernel, $a + b$, was duplicated in both of the partitions in Figure 4.7.

Node duplication not only increases area, but also adversely affects the quality of the later iterations of the *perform_kernel_extraction* procedure. If a duplicated kernel is a part of a higher level kernel, then the partitioned approach will fail to identify it later since the same logic is represented by two different nodes, and the association is lost between them. Assume that the node G in Equation (2.1) contained another cube, *cf*, to form

$$G = af + bf + ace + bce + cf$$

Then after the first iteration of kernel extraction, the network will have

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

$$G = eZ + fZ + cf \tag{4.2}$$

where X and Z refer to the same kernel, $a + b$. Then when we are performing the next iteration of kernel extraction, we will miss the kernel, $X + c$, $(Z + c)$, in F and G since X and Z are now different. Thus we lose on the quality of the transformation on the subsequent iterations due to the duplication of the kernel. The following chapter describes a novel repartitioning algorithm based on the co-kernel cube matrix that tries to eliminate all of these drawbacks.

After partitioning the graph, we distribute the nodes of the circuit based on the partitions. Then we allow each processor to independently create its own co-kernel cube matrix and perform kernel extraction. Conceptually, each processor confines its search for the best kernel in a slab of the actual co-kernel cube matrix as illustrated in Figure 4.3.

4.2.1 Experimental results

The parallel algorithm for kernel extraction on independent partitions was implemented on a SUN SPARC Server 1000E shared memory multiprocessor. To compare our approach, we used SIS to perform kernel extraction using the *gkx -bo1* option in it. The results of this experiment are tabulated in Table 4.2. It is clear that the algorithm loses some quality by not considering the vertical rectangles. The main advantage of this algorithm is that it is extremely fast and memory scalable. On average, for six processors we get a speedup of 8.63, but there is a quality degradation of 2% from that of SIS. We get super linear speedups since we are searching a fewer number of rectangles. For certain circuits, like *ex1010*, it shows a speedup of 16.30 times over the sequential algorithm , SIS, with only 1% quality degradation. The above algorithm produces better quality results than that obtained by the inorder partitioning, since the locality of the nodes in a partition is better in a min-cut partition than inorder partition.

Table 4.2: Results of parallel kernel extraction on a SUN SPARC Center 1000 using circuit partitioning.
Time is in seconds and quality is in literal counts (LC).

MCNC	Initial	SIS (1 proc)		2 processor			4 processor			6 processor		
Circuits	LC	LC	Time	LC	Time	Spd	LC	Time	Spd	LC	Time	Spd
dalu	3588	2837	16.5	2877	7.4	2.23	2972	3.0	5.50	3022	1.9	8.68
des	7412	6684	26.3	6650	11.7	2.25	6650	8.4	3.13	6658	7.1	3.70
seq	17938	9373	73.8	9367	51.8	1.42	9378	14.9	4.95	9455	15.4	4.79
spla	24087	17716	419.1	17954	193.2	2.17	18289	58.1	7.21	18484	43.4	9.66
ex1010	13977	11833	4508.9	11838	2082.3	2.16	11897	467.4	9.65	11968	276.7	16.30
average	1.00	0.723		0.726		2.05	0.734		6.09	0.740		8.63

4.3 Summary

In this chapter, we discussed a partitioned approach to parallel kernel extraction. The given circuit was partitioned into P parts for P processors, using two approaches:

1. inorder partitioning and

2. min-cut-based graph partitioning.

Then each processor performs kernel extraction independently on their part of the circuit. The prime advantage of this approach is that it is scalable and gives good speedup since there is no overhead cost of communication or synchronization. Also, it is easy to implement, and it can be upgraded to a newer release of SIS without much effort. The main disadvantage comes from the quality degradation as the number of processors increases since the ratio of the local to the global information keeps decreasing monotonically. As illustrated by Figure 4.3, the search space of each processor is confined to the horizontal slabs; thus a tall rectangle overflowing the boundary of these slabs can never be found. In the next chapter, we will discuss an algorithm that tries to take care of the vertical interactions as well. Although inorder partitioning is very fast, it produces worse quality results as compared to the min-cut-based partitioning. Hence, although inorder-based partitioning is very easy to implement, we should use min-cut-based partitioning for better locality of nodes in a partition and hence better quality.

5. PARALLEL KERNEL EXTRACTION WITH INTERACTIONS

While the parallel algorithm described in Chapter 3 based on circuit replication wasted much time maintaining the global picture in all of the processors, the complete information helped in producing good quality. The second parallel algorithm described in Chapter 4 decomposed the problem into individual circuit partitions, but in the process the search for rectangles was restricted to only horizontal slices of the co-kernel matrix. The motivation of the algorithm described in this chapter is to try to decompose the problem such that it does not require any synchronization or updates but provides enough information on each processor to produce good quality results.

5.1 L-shaped Partitioning Algorithm

We will now present a new n-way partitioning algorithm such that it can also detect the rectangles not confined to a single partition as well as avoid the duplication of kernels in different processors. Duplicate kernels were generated by different processors since the kernel-cubes, i.e., the top row of the co-kernel matrices, have duplicate entries like a, b, c

in Figure 4.6. Hence, we will try to produce a disjoint partition of the kernel-cubes such that multiple processors do not search for kernels with the same cubes. Also, in order to capture the overflowing rectangles, a processor needs to have kernels from other processors that are formed by its kernel-cubes. The following algorithm takes the **n** co-kernel cube matrices, B_i, as the input, reorganizes the entries of the matrix and finally creates **n** L-shaped submatrices. The algorithm is given below.

L-SHAPED_PARTITION($B_1, B_2, \ldots, B_n$)

```
▷ distribute kernel cube ownership
global_cubes ← NIL;
for i ← 1 to n
    for each kernelcube, c, ∈ B_i
        if  (c ∉ global_cubes)
            local_cubes[i] ⇐ c;
            global_cubes ⇐ c;
▷ add overlapping sub blocks B_ij
for i ← 1 to n
    for j ← 1 to n
        B_ij ← {c or c ∈ (local_cubes[j] ∩ B_i)}
        B_j ← B_j ∪ B_ij
```

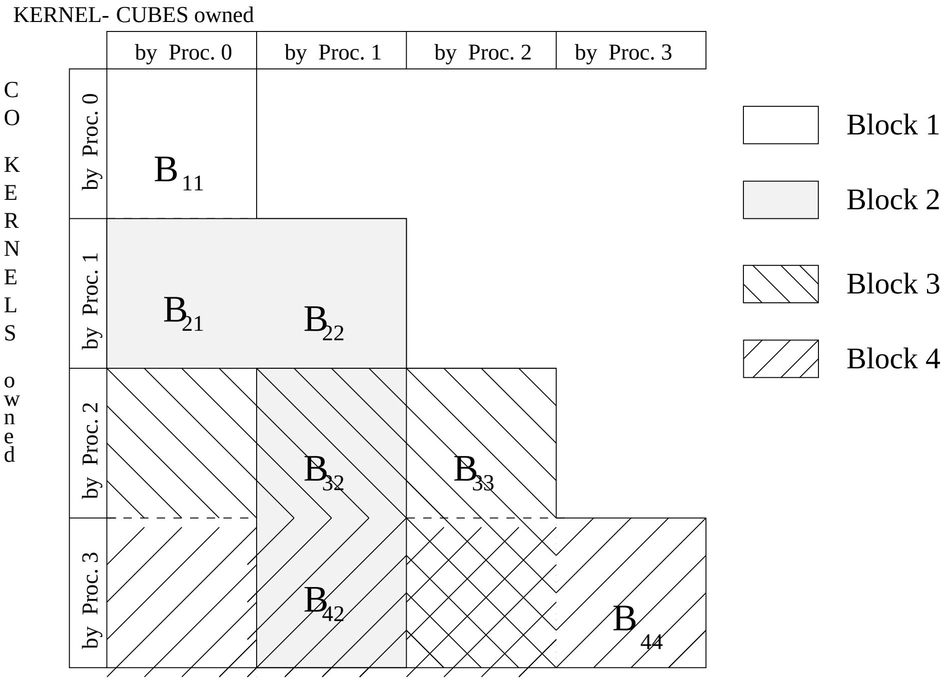

Figure 5.1: Co-kernel cube matrix after L-shaped partitioning.

The first part in the L-shaped partitioning algorithm produces disjoint partitions of the kernel-cubes in a greedy basis. The matrix B_0 owns all of its cubes, and B_1 owns all of the cubes that are not owned by B_0. Similarly, B_i owns all of its cubes not owned by the previous *i-1* matrices, namely B_0, B_1 …, B_{i-1}. This partitioning tries to prevent duplication of the kernels. Once the cubes have been partitioned, processor i identifies the submatrix B_{ij} from B_i containing cubes owned by processor j. Processor i sends this submatrix to processor j to add it to its submatrix, B_j. This overlapping helps to capture the overflowing rectangles. The partition described above gives rise to the matrix in the shape of an “L”, as shown in Figure 5.1; hence we call our strategy an L-shaped partition. Initially, each processor had the horizontal slab, and the vertical leg is added later to create the “L” shape. Example 5.1 illustrates the above algorithm.

Example 5.1 *Let us consider the network of Example 2.1 and the network partition from Figure 4.5.*

Kernel-cubes (index) of B_0 = {a (1), b (2), c (3), ce (4), f (5)}

Kernel-cubes (index) of B_1 = {a (6), b (7), c (8), de (9), f (10), g (11) }

After the *distribute_cube_ownership* routine,

Global_cubes = {a (1), b (2), c (3), ce (4), f (5), de (9), g (11)}

Local_cubes[0] = {a (1), b (2), c (3), ce (4), f (5)}

Local_cubes[1] = {de (9), g (11) }

The resultant matrices after the completion of *add_overlapping_submatrix* are shown in Figure 5.2. The overlapping submatrix B_{10} is the left-bottom quarter of the matrix in the above figure. Hence, B_0 consists of the upper diagonal block B_{00} and B_{10}, while the block B_1 consists of B_{10} and the lower diagonal block B_{11}.

Once the partition is done, overlapping kernel extraction is called on each of these L-shaped co-kernel cube matrices sequentially. Example 5.2 illustrates the working of the above algorithm.

Example 5.2 *Given the network from Example 2.1, the L-shaped partitioner creates the submatrices B_0 and B_1 of Example 5.1.*

Kernel extraction on B_0 produces the kernels X = a + b and Y = a + c. No more profitable kernels could be found once X and Y are extracted from the network since the values of all of the remaining entries become zero. Kernel extraction on B_1 leaves the

	Kernel-cubes owned by block 0					Kernel-cubes owned by block 1	
	a	b	c	ce	f	de	g
	1	2	3	4	5	9	11
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	3	4	.	.	.	.	.
F f 9	1	2	.	.	.	.	.
F c 10	.	.	.	.	.	7	4
F g 11	3	.	4	.	.	.	.

B_0 B_1

Figure 5.2: Co-kernel cube matrix after L-shaped partitioning for Equation (2.1).

network unchanged. The final transformed network is exactly the same as that obtained by MIS in Example 2.3, and the synthesized circuit is identical to Figure 2.2.

Although B_1 did not produce any profitable rectangle for this trivial example network, this would not be true for a real circuit.

5.1.1 Experimental results

The above serial algorithm was implemented in C++ on a single processor of a SUN SPARC Center 1000. The L-shaped partitioned algorithm was executed on several MCNC BENCHMARK circuits on a single processor. The results are tabulated in Table 5.1 along with that from SIS. The effectiveness of the L-shaped partitioning is shown by the negligible degradation in the quality. Thus we have succesfully taken care of the

Table 5.1: Results of kernel extraction using SIS and L-shaped partitioning on a single processor.
Quality is in literal counts (LC).

MCNC Circuits	Initial LC	SIS LC	2 way LC	4 way LC	6 way LC
misex3	1661	1142	1143	1147	1144
dalu	3588	2837	2837	2837	2851
des	7412	6648	6648	6648	6648
seq	17938	9373	9471	9464	9455
spla	24087	17716	17716	17727	17702
average	1.000	0.690	0.691	0.692	0.691

overlapping rectangles and have prevented the duplication of kernels. The above results encouraged us to use the L-shaped decomposition for our parallel algorithm for kernel extraction.

5.2 Shared Memory Parallel Algorithm

In a parallel environment, the creation of the co-kernel cube matrix is a more involved process. We want to create the L-shaped co-kernel cube matrix in all of the processors and the overlapping portions; i.e., the nondiagonal blocks, B_{ij}, have to be the same in all of them. In the sequential algorithm, when a new kernel (co-kernel) is generated, a new row is assigned for that kernel, and the row number corresponding to that kernel is noted in a table. Similarly, when a unique kernel-cube is generated, a new column is assigned for that cube, and the column number corresponding to that cube is noted in a table.

Since kernels are generated concurrently, two different kernels will get the same row index if all of the processors start from the same index, say zero. To keep the row and column labeling consistent across all processors, we start the row and column index with an *offset* that is a factor of the processor id. Thus the index of the first kernel in processor 2 will be 200001 while that in processor 5 will be 500001. Hence, the labeling of the rows will be consistent in all of the processors irrespective of the order in which the kernels are generated. A similar labeling strategy was used in [14].

Once the kernels have been generated, all of the processors send the cubes of all of its kernels, i.e, the top column of their co-kernel cube matrix, to the master processor. *Distribute_cube_ownership* partitions the cubes of all of the processors to remove duplicate cubes existing between the different processors. It produces a mapping between the global cube index and the local cube index, which is sent to the individual processor for reorganizing the rows in their submatrices, B_i. In the process, processor i identifies the portion of the matrix B_{ij} that has to go to processor j. The submatrix B_{ij} is sent to processor j to create the different L-shaped matrices. Example 5.3 illustrates the above mechanism on the partitioned circuit in Figure 4.5.

Example 5.3 *Assume the partitioned circuit in Figure 4.5.*

Kernel-cubes (index) of B_0 = {a (1), b (2), c (3), ce (4), f (5)}

Kernel-cubes (index) of B_1 = {a (100001), b (100002), c (100003), de (100004), f (100005), g (100006) }

Processor 1 sends its kernel-cubes to processor 0 to produce the global_cubes

			a	b	c	ce	f	de	g	
			1	2	3	4	5	100005	100006	
			PROC 0							
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	100001	.	.	.	.	1	5	3	P
F	b	100002	.	.	.	.	2	6	.	R
F	de	100003	5	6	7	.	.	.	.	O
F	f	100004	1	2	.	.	.	.	.	C
F	c	100005	.	.	.	.	.	7	4	
F	g	100006	3	.	4	.	.	.	.	1

Figure 5.3: Co-kernel cube matrix for processor 0 and processor 1.

Global_cubes = {a (1), b (2), c (3), ce (4), f (5), de (100005), g (100006)}

Local_cubes[0] = {a (1), b (2), c (3), ce (4), f (5)}

Local_cubes[1] = {de (100004), g (100006) }

Local_cube_index $\Rightarrow$ Global_cube_index = {(100001, 1), (100002, 2), (100003, 3), (100004, 4), (100005, 5) }

Upon receiving the mapping from its local cube index to global cube index, processor 1 identifies the submatrix B_{01} and sends it to processor 0, which attaches it to its existing matrix to produce B_0. Figure 5.3 shows the resulting co-kernel cube matrices in the two processors.

After solving the data decomposition for parallel processing, we need to study the consistency issues that arise due to concurrent operation on replicated portions of the data. Assume that all of the processors start finding the best rectangle in their respective L-shaped co-kernel cube matrix and start extracting it from the network. Whenever a rectangle encompasses nodes owned by more than one processor, we send the partial rectangle to the other processors to divide its nodes. If we do not take care of the message before finishing with the kernel extraction in the other processor, then the partial rectangle could become obsolete and nonprofitable as illustrated by Example 5.4.

Example 5.4 *Let the co-kernel cube matrix of Figure 5.3 be the given matrix.*
Assume processor 0 has found {(3, 4, 100003, 100004)(1,2)} as its best rectangle. But it cannot extract {(100003, 100004)(1,2)} from node F since it is in processor 1. Also, it needs to reset the cubes in {(100003, 100004)(1,2)} to zero since they are covered. So processor 0 sends the partial rectangle {(100003, 100004)(1,2)} to processor 1. But processor 1 does not update the values until it finishes all of its kernel extraction. At the end of it, the network is transformed to

$$F = aX + bX + aY + cY$$

Now when it tries to extract the rectangle {(100003, 100004)(1,2)} from the function F, it fails since the cubes of the original network have been transformed.

To prevent the above problem, after every division of the network with the best rectangle, each processor checks if any message has arrived. If so, it divides the network with the received rectangle. We also set the weights of the cubes of this rectangle to zero since it has already been divided by some kernel. The parallel kernel extraction algorithm based on the L-shaped partitioning is provided below.

PARALLEL_KERNEL_EXTRACTION(*network* N_i)

```
▷ generate, partition and distribute co-kernel cube matrices
B_i ← generate_co_kernel_cube_matrix(N_i)
if   (myproc ≠ MASTER_PROCESSOR)
       send_message(B_i)
else   receive_message(B_i)
Barrier_Sync()
if   (myproc = MASTER_PROCESSOR)
       Lshaped_partition(B_1, B_2, ..., B_n)
▷ perform kernel extraction
do
        while (generate_next_rectangle(B, rect) ≠ NO_MORE)
                if   (rect→value > best_rect→value)
                        best_rect ← rect
        divide_network_by_kernel(best_rect.mypart)
        value ← value + best_rect→value
        send_message(best_rect.overflowing)
        if   (pending_message)
                receive_message(others_best_rect)
                divide_network_by_kernel(others_best_rect)
   while (best_rect→value > 0 or message_pending)
```

The above algorithm takes in the partitioned network N_i as the input. Each processor creates the co-kernel cube matrix B_i from the partitioned network N_i and sends the required information to the MASTER_PROCESSOR. The master processor then performs the L-shaped repartitioning based on the algorithm described in Section 5.1. At the end of line 8, each processor has its L-shaped co-kernel cube matrix and is ready to perform kernel extraction on its B_i. Each loop iteration of the kernel extraction, line 11 to 20, achieves two tasks. First, each processor finds out the best rectangle in its L-shaped matrix and extracts the kernel. It also sends the rectangle if it overflows to other processors' nodes. The second task is to receive overflowing rectangles from other processors and extract them from their network. Each processor repeats these two tasks until there is no more profitable rectangle or no message pending from other processors. The above algorithm describes the baseline of the parallel kernel extraction. Details of the algorithms are excluded for readability purposes, but the interesting issues involved in them are discussed below.

5.2.1 Consistency issues

An important issue arises from the concurrent evaluation of the overlapping portions of the co-kernel cube matrix by more than one processor and is illustrated by Example 5.5.

Example 5.5 *Let the co-kernel cube matrix in Figure 5.3 be the given matrix.*

Suppose processor 0 is evaluating the rectangle, $\{(3,4,\ 100003,\ 100004)(1,2)\}$, and finds the gain to be eight whereas processor 1 evaluates the value of the rectangle $\{(100001, 100002)(5,100005)\}$ to be three. Since eight and three are the highest gains for any kernel in B_0 and B_1, respectively, each processor goes on to divide its network with the corresponding kernels. The final representation after extracting the above rectangles from the network leads to a savings of three instead of the expected 11 as is illustrated below.

$$\begin{aligned}
G &= ceX + fX, \\
H &= ade + cde, \\
X &= a + b, \\
F &= XY + ag + cg + cde + deX + fX, \\
Y &= de + f
\end{aligned} \tag{5.1}$$

Example 5.5 shows that if two processors share a cube, although both processors expect to gain from including the cube in its best rectangle, only one benefits from it. The sharing of cubes leads to an even deeper problem. To analyze it, let us assume that processor 1 extracts its own rectangle $\{(100001, 100002)(5, 100005)\}$ first and then the partial rectangle $\{(100003, 100004)(1,2)\}$ from processor 0. After extraction with de + f, $F = aY + bY + ag + cg + cde$. Before performing the division in processor 1, the cubes *ade, bde, af* and *bf* are added to F to make it divisible by $a+b$. As some of these cubes could have been covered in a previous extraction by some rectangle overlapping with the current one, these cubes are added to the node to ensure that the current rectangle can indeed divide the node. After the addition, the kernel a + b is extracted from F to give the form shown in Example 5.5. Thus we add cubes to the expression and extract subexpressions from these cubes without any gain. It would have been better if we had not added cubes to perform the extraction in the first place.

To avoid the above problem, we perform an extra check while performing the division. If at the point of division, the extraction is profitable assuming the cost of the kernel is zero, the division is performed after adding the nodes; otherwise it is performed on the existing node representation. Referring to the last example, after extracting the rectangle $\{(100001, 100002)(5,100005)\}$ and setting the values of the cubes in it to zero, we find the gain from extracting the rectangle $\{(100003,100004)(1,2)\}$ to be negative. Hence, in this case, we do not add the cubes to F but divide the existing F by a+b to get $F = XY + ag + cg + cde$, thereby saving eight literals instead of three.

Now we return to the observation in Example 5.5, where concurrent evaluation of the same set of cubes by two processors was producing false hopes of literal savings in both of them. Therefore, we need to devise some mechanism by which only one processor gets to save the value of the cube while the rest save zero. Once a processor speculates a cube to be in its best rectangle, it makes its value zero and stores the value of the cube in an attribute called *trueval*. Any other processor that subsequently tries to include that cube into their rectangle will find its value to be zero since it is speculatively covered by the best rectangle of some processor. Later on, if the owning processor finds a better

Table 5.2: Various states of a cube during extraction.

state	value	trueval	Description of the state
FREE	X	X	cube not covered by any best rectangle
COVERED	0	X	cube covered by some best rectangle, but not divided
DIVIDED	0	0	covered by some rectangle and divided

rectangle, it copies back the value of the cube from its trueval, thereby making the cube available for other processors. But the search for the best rectangle in a processor gets biased by the order of the generation of the rectangle.

Consider the kernel extraction of B_0 in processor 0. Let {(1,2)(4,5)} be the first rectangle, which is made the best rectangle and the values and trueval of all of its cubes are modified accordingly. Then let {(3,4,100003, 100004)(1,2)} be considered next. Since the values of the cubes 8, 9, 10 and 11 are zero, the value of the rectangle turns out to be -2. Thus although the latter rectangle is a much bigger and profitable one, the best rectangle is the first one. Before solving the problem, we try to look at the various states of the cube based on the attributes in Table 5.2.

Let us try to qualify the state COVERED by adding another attribute representing the processor that is speculating on covering it. When the same processor tries to find the value of the cube, it is returned the true value since the cube has not yet been divided. But if some other processor asks for the value, it is returned zero since that cube has already been covered by the best rectangle of the owning processor and in the near future it is going to be divided. The rationale is that the nonowner cannot change the best rectangle of the owning processor, and so for all practical purposes, the cube has been covered. But if the owning processor asks for the value of the cube, it should get the true value since the cube has not been divided, and the considered rectangle can replace its current best rectangle. Thus including the attributes *value, trueval* and *owner* in the cubes succeeds in making the search independent of the order of rectangles considered.

But as it is true for all speculative mechanisms, there are situations for which the speculation turns out to be wrong. Let us assume that the first rectangle considered by the processor 0 is $(R_0,\ C_0) = \{$(3,4,100003,100004)(1,2)$\}$ while processor 1 is evaluating $(R_1,\ C_1) = \{$(100001,100002)(5,100005)$\}$ at the same time. If processor 1 resets the value of the cubes in $(R_1,\ C_1)$ before processor 0 evaluates the value of $(R_0,\ C_0)$, then it will find the value of it to be -2. When processor 0 evaluates $\{$(1,2)(4,5)$\}$ next, it replaces $(R_0,\ C_0)$ since the value of it is three. Now $(R_1,\ C_1)$ in processor 1 gets replaced by $\{$(100001, 100005)(100005,100006)$\}$. Thus the final transformed network looks like the following:

$$\begin{aligned}
G &= ceX + fX, \\
H &= ade + cde, \\
X &= a + b, \\
F &= aY + bY + ag + cg + cde, \\
Y &= de + g
\end{aligned} \tag{5.2}$$

The final literal count is 27 instead of 22 produced by the serial algorithm. The reason for the degradation is that $(R_0,\ C_0)$ evaluates to a lesser value since it finds that some other processor is speculating on a best rectangle containing half of its cubes. Hence, it gives way to a rectangle with an *evaluated* gain higher than it, provided $(R_1,\ C_1)$ is extracted from the network. But later, $(R_1,\ C_1)$ is replaced by a better rectangle in the processor. At this point, $(R_0,\ C_0)$ will evaluate to be the most profitable in processor 0, but it has already been considered before, so no changes occur. Fortunately, such a situation occurs very rarely in practice.

A final issue with the concurrent evaluation of kernel extraction is illustrated by the following situation in Example 5.6.

Example 5.6 *Let the co-kernel cube matrix of Figure 5.3 be the given matrix.*

Let processor 0 find {(3,4,100003, 100004)(1,2)} to be the best rectangle and proceed with extracting it from the network. Hence, it divides the nodes it owns, namely G, by {(3,4)(1,2)} and sets the *value* and *trueval* of its cubes to zero. It asks processor 1 to divide its network by {(100003, 100004)(1,2)} and continues with the extraction of the next best rectangle from its submatrix, B_0. If processor 1 is busy at this time doing its kernel extraction, the requested extraction is delayed. In the mean time if processor 0 considers the rectangle {(100003, 100004)(1,2)}, its value will evaluate to be three and might be selected as the best rectangle, missing the actual profitable rectangle {(5,100003, 100006)(1,3)} having the same value three. Since all of the cubes of {(100003, 100004)(1,2)} are owned by processor 0 from the previous iteration, the rectangle evaluates to three.

The above problem arises due to a delay in the division of the two parts of the rectangle since it is distributed across two processors. Although the decision has been taken and the cube is definitely going to be covered, the cube value when asked is returned the actual value instead of zero since it is asked by the owning processor. To avoid this, we add a version number to the cube. As soon as the best rectangle for a processor is decided, it advances the version number for that processor. Any further queries on the cubes owned by this processor in any earlier versions will visualize the cube as being covered.

A consistency issue arises when one processor tries to own the cube that has already been locked by some other processor or by this processor in some previous version. The problem comes when it later finds a better rectangle and tries to free the cubes. The processor reassigns the value of all of the cubes it owns. In the process, it erroneously reassigns value to cubes that have been covered by the best rectangle of some other processor or by its best rectangle from some previous iteration. The error is avoided by adding the attribute FREE to the cube. Thus only if a cube is FREE is it owned by a processor; otherwise it is left as it is.

Example 5.7 Let us assume that processor 0 has found $\{(3,4, 100003, 100004)(1,2)\}$ as its best rectangle. So it owns the cubes 1, 2, 5, 6, 8, 9, 10 and 11. Later the processor evaluates $\{(100003, 100006)(1, 3)\}$ and includes it to be its best rectangle, thereby putting 3, 4, 5 and 7 under its ownership. So when it gets replaced by some other rectangle, it frees cube 5, which is still supposed to be covered by $\{(3,4, 100003, 100004)(1,2)\}$.

Table 5.3: Results of parallel algorithm with L-shaped partitioning on a SUN SPARC Center 1000 shared memory multiprocessor.
Time is in seconds and quality is in literal counts (LC).

MCNC	Initial	SIS (1 proc)		2 processors			4 processors			6 processors		
Circuits	LC	LC	Time	LC	Time	Spd	LC	Time	Spd	LC	Time	Spd
dalu	3588	2837	16.5	2874	8.3	1.99	2935	3.9	4.23	3025	2.4	6.88
des	7412	6648	26.3	6658	10.1	2.60	6656	8.4	3.13	6653	2.9	9.07
seq	17938	9373	73.8	9274	65.2	1.13	9038	31.5	2.34	9255	22.0	3.35
spla	24087	17716	419.1	17709	289.3	1.45	17712	271.4	1.54	17717	264.8	1.58
ex1010	13977	11833	4508.9	11836	2138.0	2.11	11842	577.9	7.80	11865	392.9	11.48
average	1.000	0.723		0.721		1.86	0.719		3.81	0.724		6.47

5.2.2 Experimental results

The parallel algorithm for kernel extraction with the L-shaped partitioning with interactions was implemented on a SUN SPARC Server 1000E shared memory multiprocessor. We use SIS to compare our results by performing kernel extraction using the option *gkx -bo1*. Table 5.3 summarizes the quality and run times of executing this algorithm on 2, 4 and 6 processors and compares it with the equivalent kernel extraction technique of SIS. The results show that the L-shaped partitioning has been able to provide a good speedup, i.e., a speedup of 6.47 on average on 6 processors, without losing hardly any quality compared to SIS. Also, the above heuristics restricted quality degradation to less than 1% from the sequential code. In fact, in certain cases, like *seq* on four processors, it produces better results than SIS.

5.3 Shared Memory Parallel Kernel Extraction with Barrier

Although the quality produced by the above parallel algorithm is very close to the SIS result, they are not exact, and hence there are scopes of improving the quality control measures. Despite including all of the quality control techniques described in the previous section, each processor may not have the recent information on the overlapping portions of their matrices. A cube when trying to be accessed by two processors *simultaneously* can return the actual value of it; although ideally, one should find zero as its value since we do not use locks. The reason for not using locks is to avoid sequentializing the code. Also a situation similar to the caveat discussed in the speculative approach can lead to a wrong impression of the cube. All of these may lead to cubes sometimes being divided multiple times without providing true gain. Another reason is that the search for the best rectangle is local to each processor and hence may not be the global optimum. These are a few reasons for a quality degradation for moving from sequential to parallel.

All of these can be solved by synchronizing the execution in all of the processors. If we put a barrier at the end of the loop after line 13 and send the best rectangles to a central processor, it can decide on the best rectangle amongst them. It then broadcasts this rectangle to every processor, which extracts the corresponding kernel from their networks and updates their submatrices. The update prevents any consistency problems and allows everybody to start their search for the next best rectangle with the latest information. The modified algorithm is shown below:

```
PARALLEL_KERNEL_EXTRACTION_WITH_BARRIER(network N_i)
▷ generate, partition and distribute co-kernel cube matrices
B_i ← generate_co_kernel_cube_matrix(N_i)
if   (myproc ≠ MASTER_PROCESSOR)
     send_message(B_i)
else  receive_message(B_i)
Barrier_Sync()
if   (myproc = MASTER_PROCESSOR)
     Lshaped_partition(B_1, B_2, ..., B_n)
▷ perform kernel extraction with barrier
do
        while (generate_next_rectangle(B, rect) ≠ NO_MORE)
                if   (rect→value > best_rect→value)
                     best_rect ← rect
        if   (myproc ≠ MASTER_PROCESSOR)
             send_message(best_rect)
        else  for proc ← 1 to n − 1
                   receive_message(rect)
                   if   (rect→value > best_rect→value)
                        best_rect = rect
              Broadcast(best_rect)
```

```
21          if (myproc ≠ MASTER_PROCESSOR)
22              receive_message(best_rect)
23          divide_network_by_kernel(best_rect.mypart)
24          Barrier_Sync()
25      while (best_rect→value > 0)
```

The above algorithm is very similar to the one described earlier. We create the L-shaped submatrix B_i in each processor (Line 1 to 8). Each processor finds out the best rectangle in its own L-shaped submatrix and sends it to the master processor, which finds out the best rectangle among all of the processors. The master processor then broadcasts the global best rectangle to other processors. Once everybody has finished extracting the best rectangle, the next iteration starts and this repeats till any profitable rectangle is left.

5.3.1 Experimental results

We included this modification in our previous algorithm and performed some experiments using a SUN SPARC Center 1000 on the MCNC BENCHMARK circuits to investigate its merits. The run times and the quality are summarized in Table 5.4. The quality did not seem to improve significantly inspite of the barrier. But the more interesting observation is that the run time increases, instead of decreasing, as we increase the processor. The increase in run time can be attributed to a couple of problems. First, since

Table 5.4: Results of overlapping kernel extraction with barriers on a SUN SPARC Center 1000 shared memory multiprocessor.
Time is in seconds and quality is in literal counts (LC).

MCNC	Initial	SIS		2 processor		4 processor		8 processor	
Circuits	LC	LC	Time	LC	Time	LC	Time	LC	Time
dalu	3588	2837	16.5	2924	39.44	2935	69.02	-	-
duke2	1746	1077	1.4	1071	3.00	1103	4.99	1236	22.05
misex3	1661	1142	2.1	1138	5.32	1142	7.71	1177	42.56
seq	17938	9373	73.8	9372	113.55	9487	221.52	-	-
t481	6823	5792	29.4	5927	19.54	5841	34.74	5895	253.51

all of the data structures are created in a shared memory, each processor gets blocked during a *shared malloc* system call, which sequentializes the execution. Also, the contention for the shared memory lock increases with increasing the number of processors, and hence the run time increases. Another reason for the slow down of the run time compared to the previous algorithm is that only one rectangle is extracted after every iteration whereas in the previous algorithm n best rectangles are extracted if there are n processors.

5.4 Distributed Memory Parallel Algorithm

In comparison to the parallel approach described in this chapter, the approach is slightly different for a distributed memory multicomputer architecture. The algorithms described in Chapter 3 and 4 are identical in shared memory and distributed memory multiprocessors. Hence, they were not separately described. In the shared memory multiprocessors, entries on the two different L-shaped co-kernel cube matrices would point to the same cube data if both of the entries implied the same cube. But in a distributed system, these two entries will point to two different address spaces. A lack of common address space may lead to inconsistencies since after updating one of the cubes, the change does not get immediately reflected in the cubes present in the other processors. Hence, we have to send a message to the other processor to update the other. Hence ownership of a cube is not immediately reflected as soon as the processor has acquired ownership. Thus the concept of ownership will not work for the distributed case.

But we should try to reduce the effect of the false gain as much as possible. One way is to add an uncertainty factor to the value of the cube present in the overlapping portion of the L-shaped co-kernel cube matrix. Thereby we can bias the search to find rectangles from the diagonal block of the matrix. We put a higher bias to the vertical regions than to the horizontal regions of the L-shaped since we have observed that most of the rectangles are tall and thin.

Another optimization is to prevent the processor from considering rectangles that do not have any owned cubes since such a rectangle will definitely be searched by the processor owning that block. Also, that processor might be able to find a bigger rectangle which might be preempted by finding the smaller rectangle.

Example 5.8 *Let the co-kernel cube matrix of Figure 5.3 be the given matrix.*

Assume processor 0 finds {(3, 4, 100003, 100004)(1,2)} as its best rectangle, while processor 1 finds {(100003, 100006)(1, 3)} as its best rectangle in their first iteration. So processor 1 sends a message to processor 0 asking it to reset the values of the cubes of $\{(100003, 100006)(1, 3)\}$ in B_0, which the latter does after it finishes the extraction of its own rectangle. In the next iteration, processor 0 does not find the next best rectangle, namely $\{(5, 100003, 100006)(1,2)\}$, since it has been preempted by the partial rectangle from processor 1.

A bigger rectangle was not found in the above example since a part of it was chosen by some other processor. To alleviate this problem, we do not allow a processor to consider rectangles that are entirely to the left of its diagonal block since that block is covered by some other processor.

5.4.1 Experimental results

The above parallel algorithm was implemented on an Intel Paragon distributed memory message passing multiprocessor. The results and run times of executing the above algorithm on four and eight processors of the Paragon are summarized in Table 5.5. The reported results and the run times of SIS are on one processor of the SUN SPARC 600 since SIS was not available on the Intel Paragon. Although the algorithm produces good speedups, we see a degradation of around 8-9% in the quality compared to that of MIS. The main reason is that more than one processor is including a cube in its best rectangle and thereby getting a false gain estimation. We tried some experiments on the value of the *uncertainty factor*. If we bring the uncertainty factor close to one, then we get rectangles that are long, which improves the quality.

Table 5.5: Results of overlapping kernel extraction on an Intel Paragon distributed memory multiprocessor.
Time is in seconds and quality is in literal counts (LC).

MCNC	Initial	SIS		4 processors		8 processors	
Circuits	LC	Time	LC	Time	LC	Time	LC
dalu	3588	125.4	3014	8.74	3208	4.17	3174
des	7412	94.0	6648	1.98	6904	3.95	6848
duke2	1746	5.4	1077	2.05	1172	0.83	1157
frg2	2851	24.5	2033	5.66	2334	1.30	2214
misex3	1661	7.3	1142	4.19	1255	1.45	1334
seq	17938	205.5	9373	104.64	10331	67.28	11442
t481	6823	47.8	5808	5.85	6533	5.35	6644
average	1.00		0.692		0.755		0.780

5.5 Summary

In this chapter, we discussed parallel algorithms for kernel extraction with some interactions between different processors. First, we discussed a novel L-shaped partition, which enables us to search tall rectangles not included in the best rectangle search discussed in Chapter 4. The sequential algorithm developed, based on this L-shaped partitioned matrix, produced comparable results even for eight partitions. However, we found out that direct parallization of it results in a poor quality result due to consistency issues. We solve the problem by two approaches:

1. including state information attributes in the individual cubes and

2. using barrier synchronization.

The first approach produced comparable results and good speedups. The other approach produced good quality results but showed slowdowns due to the synchronization after every iteration. Also, the latter approach extracted only one kernel in each iteration, whereas the former approach extracted P kernels for P processors. We also discussed a parallel kernel extraction algorithm for the distributed memory system. The quality degradation is mainly due to the communication latency, which makes the information present on a processor obsolete, and the kernel extraction algorithm is very sensitive to it. In the next chapter, we conclude our work and provide some directions for future research.

6. CONCLUSIONS

6.1 Summary of Thesis Work

Algebraic factorization is an extremely important part of any logic synthesis system and is used repeatedly in logic synthesis. The time requirements for performing algebraic factorization using kernel extraction for large circuits is quite large; hence it is important to look at parallel processing to speed up the procedure. Unfortunately, since the kernel extraction process is inherently sequential in nature, it is not at all obvious how one should proceed in parallelizing this key procedure in logic synthesis. Although some research has been performed on developing parallel algorithms for logic synthesis as a whole, there has been hardly any effort to separately develop parallel algorithms for this key optimization technique. This work has presented a detailed study of parallel algorithms for kernel extraction. It investigates the entire spectrum of parallel approaches.

We performed some detailed studies on a previously published algorithm [14] that uses data replication and uses a divide-and-conquer strategy to follow the same search path as in the sequential algorithm in Chapter 3. Although it obtains comparable quality to that of the serial algorithm, it suffers from poor speedup since the parallelism is restricted by the sequential dependencies and redundant work required in maintaining consistency between the global picture across all processors.

A second algorithm described in Chapter 4 uses totally independent factorization on different circuit partitions with no interactions among the partitions. This algorithm provides large speedups, like 16.3, but suffers in quality as the number of partitions increases.

A third algorithm described in Chapter 5 represents a compromise between the two approaches. It uses a novel L-shaped partitioning strategy that provides some interaction among the rectangles obtained in various partitions. It produces better results than the parallel algorithm using independent partitions but produces lower speedups. In *ex1010*, the L-shaped algorithm executes 11.48 times faster than SIS, with quality degradation of less than 0.2 %.

We have therefore discussed several algorithms for parallel kernel extraction. If speed is of primary importance and the user can tolerate some degradation in quality, then the independent partitioned approach would be preferable. We showed that if one uses a smart partitioning strategy, the quality degradation can be controlled. The partitioned approach without any interaction is perhaps the most practical parallel kernel extraction strategy.

If quality is of utmost importance, then the parallel algorithm using L-shaped partitioning is the preferred algorithm. It outperforms the parallel algorithm with a replicated approach in memory scalability (in terms of the ability to synthesize very large circuits) and in run times. The quality of the results is very close to the serial algorithm.

Thus we have successfully developed parallel algorithms for the *minimum-weighted rectangle-covering problem*. Even though the specific implementations of the above algorithms target area minimization via literal count measures, our methods can be directly applied to timing-driven and low-power-driven synthesis provided that the algorithms are formulated in terms of a rectangular-covering problem.

6.2 Future Research Directions

In the parallel kernel extraction algorithms using independent partitions in Chapter 4, the effect of other partitioning techniques on the quality will be investigated. Fanin-cone-based and fanout-cone-based partitioning, Chapter 7 of [18], can play an important part in the quality of the final synthesized circuit. In the independent approach, a global resubstitution on the entire circuit following a parallel kernel extraction could reduce the quality degradation. A parallel resubstitution algorithm needs to be developed for this purpose.

The quality of the interacting kernel extraction can be improved by adding some synchronizations and providing updates after a certain number of kernel extractions so that everybody can have a correct image of the co-kernel cube matrix. The main issue to be studied is to find out the right balance between the update frequency so as to provide consistent matrix information for as long a period as possible without paying too much for the synchronization.

In the future, we will investigate parallel algorithms for other key operations of logic synthesis, namely cube extraction, simplification, node resubstitution and technology mapping. Parallel algorithms for cube extraction can be developed in a similar manner as that of kernel extraction. In simplification, we can start with the partitioned circuit, simplify them independently in parallel, then repartition it and again simplify. Parallel resubstitution requires a parallel breadth-first-search algorithm to be implemented. In technology mapping, we can perform the matching phase, which is the most computationally intensive part, in parallel.

REFERENCES

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

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

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

[4] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. CAD-6, no. 6, pp. 1062–1081, Nov. 1987.

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

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

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

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

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

[10] C.-F. Lim, "A parallel algorithm for multi-level logic synthesis using the transduction method," M.S. thesis, University of Illinois at Urbana-Champaign, Sept. 1991, Tech. Rep. CRHC–91–27.

[11] C.-F. Lim, K. De, P. Banerjee, and S. Muroga, "A shared memory parallel algorithm for logic synthesis," in *Proceedings of the International VLSI Design Conference*, Bombay, India, Jan. 1993.

[12] K. De, B. Ramkumar, and P. Banerjee, "A portable parallel algorithm for logic synthesis using transduction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 13, no. 5, pp. 566–580, May 1994.

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

[14] K. De, J. A. Chandy, S. Roy, S. Parkes, and P. Banerjee, "Portable parallel algorithms for logic synthesis using the MIS approach," in *Proceedings of the International Parallel Processing Symposium*, Santa Barbara, CA, Apr. 1995, pp. 569–578.

[15] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multi-level logic optimization and the rectangular covering problem," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1987, pp. 66–69.

[16] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitiongs," in *Proceedings of the Design Automation Conference*, June 1982, pp. 175–181.

[17] L. A. Sanchis, "Multiple-way network partitioning with different cost functions," *IEEE Transactions on Computers*, vol. 42, pp. 1500–1504, 1993.

[18] P. Banerjee, *Parallel Algorithms for VLSI Computer-aided Design Applications*. Englewoods-Cliffs, NJ: Prentice Hall, 1994.