A Comparison of Parallel Approaches for Algebraic Factorization in Logic Synthesis

Sumit Roy
Coordinated Science Laboratory
University of Illinois
1308 W. Main St., Urbana, IL 61801, USA
sroy@crhc.uiuc.edu

Prithviraj Banerjee
Center for Parallel & Distributed Computing
Northwestern University
2145 Sheridan Road, Evanston, IL 60208, USA
banerjee@ece.nwu.edu

Abstract

Algebraic factorization is an extremely important part of any logic synthesis system but is computationally expensive. Hence it is important to look at parallel processing to speed up the procedure. This paper presents three different parallel algorithms for algebraic factorization. The first algorithm uses circuit replication and uses a divide and conquer strategy. A second algorithm uses totally independent factorization on different circuit partitions with no interactions among the partitions. A third algorithm represents a compromise between the two approaches. It uses a novel L-shaped partitioning strategy which provides some interaction among the rectangles obtained in various partitions. For a large circuit like ex1010, the last algorithm runs 11.5 times faster over the sequential kernel extraction algorithms of SIS on 6 processors with less than 0.2% degradation in quality of the results.

1 Introduction

Multi-level logic synthesis is a very important phase of VLSI system design. The algorithms for logic transformations used in the SIS [7] system form the core of numerous university and industrial logic synthesis systems. The SIS synthesis tool is based on algebraic factorization and simplification of nodes in a Boolean network [2]; each of these operations are computationally expensive. Table 1 shows the number of times the algebraic factorization procedure is called in a typical script from the SIS package [7], the times spent in performing these factorizations and the total synthesis time. We observe that factorization, on an average, takes 61.45% of the total synthesis time. Hence we parallelize the algebraic factorization to speed up the synthesis process.

This research was supported in part by the Semiconductor Research Corporation under Contract SRC 95-DP-109 and the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research Office.

Table 1: Runtimes of several circuits and the time it takes in the kernel extraction routine. LC = Literal Count

MCNC Circuits	Size (LC)	Factorization Invoked	Total Fac. Time(sec)	Total Syn. Time(sec)
dalu	3588	10	34.2	283.1
seq	17938	16	295.1	900.2
des	7412	10	425.7	1800.4
spla	24087	12	1771.4	2681.6
ex1010	14952	9	5841.7	7950.6
total	67997		8368.1	13615.9

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

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

$$F = af + bf + ag + cg + ade + bde + cde,$$
$$G = af + bf + ace + bce, \quad H = ade + cde \qquad (1)$$

In the network, $\mathcal{N}$, $a + b$ is a common factor that can be factored out from F and G to produce an sum of products(*SOP*) expression with 8 less literals. The new expressions for F and G are shown below:

$$F = fX + deX + ag + cg + cde,$$
$$G = fX + ceX, \quad X = a + b$$

The literal count(*LC*), a first order estimate of circuit area, has reduced from 33 to 25. By repeatedly performing the factorization on a large network, it is possible to obtain great reductions.

Recently, several approaches have been reported on developing parallel logic synthesis algorithms. Theeuwen developed a parallel synthesis algorithm [8] for an Alliant FX-8, a shared memory multiprocessor. He restructured the program flow of the synthesis tool reported in [9] and then parallelized each of the synthesis operation individually. Although he reported linear speedup, many of the circuits ran slower than SIS or produced worse quality than that produced by SIS.

De and Banerjee [5] reported on a parallel algorithm for synthesis using the transduction approach called PROPER-SYN. In another study, De and Banerjee 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 [3]. It was shown that the overall synthesis quality is significantly improved by this iterative repartitioning and resynthesis approach over the single partitioned approach without any interactions. In ProperMIS, De and Banerjee [4] parallelizes MIS by dividing up the search space across different processors but retains the global information of the circuit in all of them. This requires synchronization and update after every step of factorization and hence is not scalable.

In this paper, we focus on algebraic factorization using the kernel extraction procedures in the SIS system and try to achieve speedup by parallelizing it by partitioning the circuit in a novel way. We introduce interactions between the partitioned circuit using a novel L-shaped repartitioning. Performing kernel extraction on a partitioned circuit give us speedups while the L-shaped partitioning takes care of the quality. The outline of the paper is as follows. In Section 2, we review the sequential algorithm for algebraic factorization in SIS. The following three sections describe the three parallel algorithms for kernel extraction. Section 3 describes an approach using replicated circuit structure but a divide and conquer search. Section 4 describes a parallel kernel extraction on partitioned circuits. An unique approach using L-shaped partitioning of the co-kernel cube matrix is described in Section 5. We conclude the paper in Section 6 by comparing the various parallel approaches.

2 Overview of serial algebraic factorization

In this section, we will review some basic definitions as given in [2] to be used later in this paper. A *literal* is a variable or its negation. A *cube* is a set C of literals such that $x \in C$ implies $\overline{x} \notin C$. An *expression* is a set f of cubes. An expression f is *cube-free* if no cube divides f evenly. The *primary divisors* of an expression f form a set of expressions $D(f) = \{f/C | C \text{ is a cube}\}$. The *kernels* of an expression f are the expressions $K(f) = \{g | g \in D(f)$ 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* ($KC\ matrix$) is a sparse matrix with rows representing co-kernels and columns representing kernel-cubes. A non-zero element B(i,j) of the matrix corresponds to a cube of the function formed by the union of co-kernel i and kernel-cube j. In the network described in Equation 1, the kernels (and co-kernels) of **G** are ce+f(a, b), a+b(f,ce).

Algebraic factoring takes a sum of product (*SOP*) realization of a circuit as its input, searches for common subexpressions and extracts these subexpressions to produces a SOP expression with a smaller literal count. When the subexpression is a cube (kernel) then the factoring is called *cube extraction* (*kernel extraction*). Since the algorithms for kernel extraction and cube extraction are almost similar, we will be dealing with one of them, namely the kernel extraction algorithm. Henceforth we will be using algebraic factorization and kernel extraction interchangeably.

In [1] this optimization problem is mapped into a *minimum-weighted rectangle covering problem*, which is to find a set of rectangles with minimum total weight such that all 1's are covered. A rectangle (R,C) of the KC matrix B, where $b_{ij} \in \{0, 1, *\}$, is a subset of rows R and a subset of columns C such that $b_{ij} \in \{1, *\} \forall i \in R, j \in C$. The rectangle in a KC matrix corresponds to a set of cubes which can be divided by the kernel formed by the set of cubes of the column C. Thus the problem of kernel extraction reduces to finding a rectangle cover of the KC matrix with maximum total gain, where gain indicates the literal savings when a kernel is extracted.

3 Parallel kernel extraction using replicated circuit

In this section, we describe an approach originally presented in [4], which assumes replicated circuit information in all the processors and follows the same search path as that in SIS. It derives concurrency by subdividing the search space of the optimization problem. The circuit as well as the entire KC matrix is replicated in all the processors. The nodes of the circuit are conceptually partitioned among the processors to divide the task of generating the KC 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 KC matrix. With the help of a novel labeling scheme for the kernels, all the processor succeed in obtaining exactly the same representation of the KC matrix.

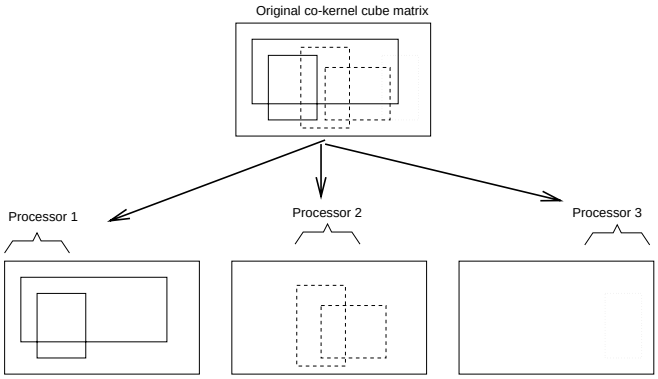

Figure 1: Decomposing a search space for finding the maximum-valued rectangle among processors.

Once the KC 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 1. Processor 1 gets the rectangles whose leftmost columns are in the left third of the area, processor 2 gets the second third and so

on. It consists of a top-down traversal of the tree to generate all 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 rectangles starting from a certain column depending on its processor id. The processor which owns the root of the search tree identifies the best rectangle and broadcasts it to all processors. Each processor then goes on to divide its native copy of the matrix with the node and repeats the above process till it finds no more profitable rectangle.

3.1 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 summarized in Table 2. The **S** column indicates how many times faster the parallel executions are compared to the single processor run. For the circuits, *spla* and *ex1010*, the program did not terminate after 10000 seconds and some ran out of memory. The results demonstrate that the quality obtained by the multiprocessor runs are comparable to that obtained by a single processor run, which is expected since every processor maintains global information about the KC matrix. Although the multiprocessor works with global information, the difference in the quality between the sequential and distributed run is due to the different search path they might have taken. 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 the processors. The difference in the quality and the time for the sequential run of the above algorithm and SIS is due to the different scripts which were used by them. Also the above algorithm is not scalable since the circuit and the KC matrix is replicated on all processors. The next section aims at relaxing the tight coupling existing between the data in all the processors, thereby hoping to save on the redundant work.

MCNC	Initial	2 processors		4 processors		6 processors	
Circuits	LC	LC	**S**	LC	**S**	LC	**S**
dalu	3588	2139	1.46	2139	1.83	2139	1.97
des	7412	6092	1.82	6094	2.99	6092	3.56
seq	17938	2650	1.64	2632	2.36	2633	2.54
spla	24087	-	-	-	-	-	-
ex1010	13977	-	-	-	-	-	-
overall	28938	10881	1.64	10885	2.39	10864	2.44

Table 2: Results of Parallel Kernel Extraction Using Circuit Replication

4 Parallel kernel extraction using circuit partitioning without interaction

In this section, we perform kernel extraction on partitioned networks independently. Since we optimize the partitioned circuit only, we give up on search for rectangles encompassing more than one partition; hence we minimize such overflowing rectangles. The circuit is mapped to a graph, by transforming the nodes to vertices and the fanin-fanout relation between node pairs into edges. We apply a min cut based graph partitioning algorithm [6] to partition the circuit into n parts. Partitioning the logic nodes of a circuit corresponds to dividing the KC matrix horizontally into row slices as illustrated in Figure 2. 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 KC matrix and perform kernel extraction. Conceptually, each processor confines its search for the best kernel in a rowwise partition of the actual co-kernel cube matrix as illustrated in Figure 2.

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

Figure 2: Partitioned co-kernel cube matrix for Eq. 1

Example 4.1 When the min cut partitioner is applied to $\mathcal{N}$ of Example 1.1 , it divides the circuit into $\{F\}$ and $\{G, H\}$. The corresponding co-kernel cube matrix is shown in Figure 2. The second and the third column represents the co-kernels and the row indices respectively. Similarly, the first and the second row indicates the kernel-cube and column indices. If we perform kernel extraction on each of these matrices independently, the resultant network is given by Equation 2. The transformed network has 26 literals instead of 22 produced by the kernel extraction routine in SIS.

$$G = ceZ + fZ, \quad H = deY, \quad Z = a + b, \quad Y = a + c,$$
$$F = deX + fX + ag + cg + cde, \quad X = a + b \quad (2)$$

If we perform kernel extraction on each of these matrices independently, the resultant network is given by Equation 2, which has 26 literals instead of 22 produced by the sequential kernel extraction routine in SIS.

This 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. $\{$(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 be the best rectangles in their partitions individually. The rectangle {(6,11)(1,3)} was not detected since the partial rectangle {(6)(1,3)} did not have a positive gain.

Another significant problem 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 2 smaller rectangles {(3,4)(1,2)} and {(9,10)(1,2)}. Hence the kernel, $a + b$, was duplicated in both the partitions in Equation 2

4.1 Experimental results

The parallel algorithm for kernel extraction on independent partitions was implemented on a SUN SPARC Server 1000E shared memory multiprocessor. The results of this experiment are tabulated in Table 3. 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 an average, for 6 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 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. In the next section we will discuss an algorithm which tries to take care of the vertical interactions as well and thereby minimize the quality degradation.

MCNC Circuits	Initial	2 processor		4 processor		6 processor	
	LC	LC	**S**	LC	**S**	LC	**S**
dalu	3588	2877	2.23	2972	5.5	3022	8.68
des	7412	6650	2.25	6650	3.13	6658	3.70
seq	17938	9367	1.42	9378	4.95	9455	4.79
spla	24087	17954	2.17	18289	7.21	18484	9.66
ex1010	13977	11838	2.16	11897	9.65	11968	16.30
average	1.00	0.726	2.05	0.734	6.09	0.740	8.63

Table 3: Results of Parallel Kernel Extraction Using Circuit Partitioning

5 Parallel kernel extraction using partitioning with interactions

While the parallel algorithm based on circuit replication wasted a lot of time to maintain the global picture in all the processors, the complete information helped in producing good quality. The second parallel algorithm 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 third and final algorithm 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. This algorithm is restricted to shared memory architectures only, unlike the previous two.

5.1 L-shaped partitioning of KC matrix

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, has duplicate entries like a, b, c in Figure 2. 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. Secondly, in order to capture the overflowing rectangles, a processor needs to have kernels from other processor which are formed by its kernel-cubes. The following algorithm takes the **n** KC matrices, B_i, as the input, reorganizes the entries of the matrix and finally creates **n** L-shaped sub-matrices. The algorithm is given in pseudo code below:

L-SHAPED_PARTITION$(B_1, B_2, \ldots, B_n)$
1 ▷ distribute kernel cube ownership
2 $global\_cubes \leftarrow$ NIL;
3 **for** $i \leftarrow 1$ **to** n
4 **for each** $kernelcube, \mathbf{c}, \in B_i$
5 **if** $(\mathbf{c} \notin global\_cubes)$
6 $local\_cubes[i] \Leftarrow \mathbf{c}$;
7 $global\_cubes \Leftarrow \mathbf{c}$;
8 ▷ add overlapping sub blocks B_{ij}
9 **for** $i \leftarrow 1$ **to** n
10 **for** $j \leftarrow 1$ **to** n
11 $B_{ij} \leftarrow \{\mathbf{c}$ or $c \in (local\_cubes[j] \cap B_i)\}$
12 $B_j \leftarrow B_j \cup B_{ij}$

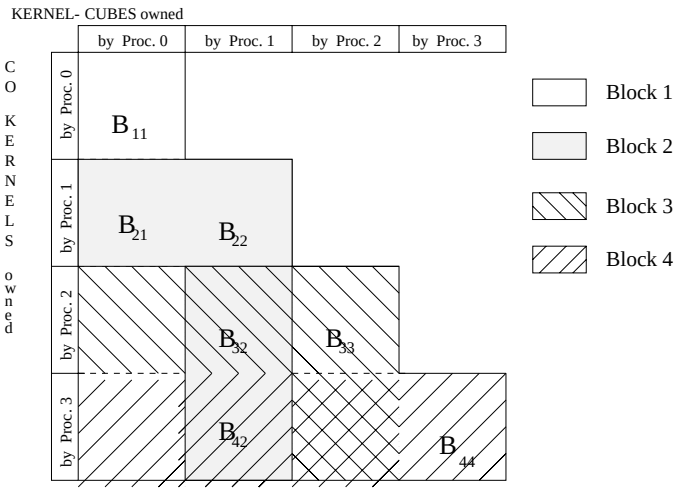

Figure 3: KC matrix after L shaped partitioning

The first part in the L-shaped partitioning algorithm produces disjoint partitions of the kernel cubes on a greedy basis. The matrix, B_0 owns all its cubes, B_1 owns all its cubes

which are not owned by B_0. Similarly B_i owns all 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 own sub-matrix, B_j. This overlapping helps to capture the overflowing rectangles. The partition described above gives rise to the matrix in the shape of L, as shown in Figure 3, 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. Once the partition is done, the overlapping kernel extraction procedure is called on each of this L-shaped KC matrices sequentially.

The L-shaped partitioned algorithm was executed on several MCNC Circuits on a single processor. The results are tabulated in Table 4 along with that from SIS. The effectiveness of the L-shaped partitioning was shown by the negligible degradation in the quality. This encouraged us to use the L-shaped decomposition for our parallel algorithm for kernel extraction.

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

Table 4: Results of Kernel Extraction using SIS and L-shaped partitioning on a single processor

5.2 Parallel algorithm for kernel extraction

In a parallel environment, the creation of the KC matrix is a more involved process. We want to create the L-shaped KC matrix in all the processors and the overlapping portions , i.e. the non-diagonal blocks, B_{ij}, have to be 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 the processors start from the same index, say 0. To keep the row and column labeling consistent across all processors, we start the row and column index with an *offset* which is a factor of the processor id. So the index of the first kernel in processor 2 will be 200001 while that in processor 5 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 [4].

Once the kernels have been generated, all the processors send the cubes of all 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 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} which has to go to processor j. The submatrix B_{ij} is sent to processor j to create the different L-shaped matrices. Example 5.1 illustrates the above mechanism.

Example 5.1 *Let $\{G, H\}$ and $\{F\}$ be the 2-way partition for the network, $\mathcal{N}$, of Example 1.1. The corresponding KC matrix looks similar to that shown in Figure 2, but with the indices of the kernel cubes and co-kernels of the KC matrix are marked according to the above strategy.*
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
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 4 shows the resulting KC matrices in the 2 processor.

			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 4: Co-kernel cube matrix for processor 0 and processor 1

5.3 Consistency issues

After solving the data decomposition for parallel processing, we need to study the consistency issues which arise due to concurrent operation on replicated portions of the data.

Assume that all the processors start finding the best rectangle in their respective L-shaped KC matrix and 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. The other processor will extract the kernel once it has completed one iteration of kernel extraction.

An important issue arises from the concurrent evaluation of the overlapping portions of the KC matrix by more than one processor and is illustrated by Example 5.2.

Example 5.2 Suppose processor 0 is evaluating the rectangle, {(3,4, 100003, 100004)(1,2)}, and finds the gain to be 8 whereas processor 1 evaluates the value of the rectangle {(100001, 100002)(5,100005)} to be 3. Since 8 and 3 are the highest gains of kernels of 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 saving of 3 instead of the expected 11 as is illustrated below.

$$G = ceX + fX, \quad H = ade + cde, \quad X = a + b,$$
$$F = XY + ag + cg + cde + deX + fX, \quad Y = de + f$$

Example 5.2 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 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 5.2.

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 with a+b to get F' thereby saving 8 literals instead of 3.

$$F' = XY + ag + cg + cde,$$

Now we return to the observation in Example 5.2, where concurrent evaluation of the same set of cubes by two processors were 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 saves zero. Once a processor speculates a cube to be in its best rectangle, it makes its value zero and but stores the value of the cube in an attribute called *trueval*. Any other processor which 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 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 its cubes modified accordingly. Then let {(3,4,100003, 100004)(1,2)} be considered next. Since the values of the cubes 8, 9 10, 11 are zero, the value of the rectangle turns out to be -2. Thus although the later rectangle is a much bigger and profitable one, the best rectangle indicates to 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 (**V** is the current value and **T** is the true value of the cube).

Table 5: Various states of a cube during extraction

state	**V**	**T**	what a state implies
FREE	X	X	cube not covered by any best rectangle
COVERED	0	X	cube covered but not divided
DIVIDED	0	0	covered by some rect and divided.

Let us try to qualify the state COVERED by adding another attribute representing the processor which is speculating on covering it. When the same processor tries to find the value of the cube, it is returned the true value as 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 non-owner 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.

5.4 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 result by perform kernel extraction using the option *gkx -bo1*. Table 6 summarizes the quality and runtimes of executing this algorithm on 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 an average on 6 processors without losing hardly any quality compared to SIS.

$$Speedup = \frac{p^2}{(1 + \frac{\gamma.(p-1)}{2.\alpha.p})^2} \quad (3)$$

where p is the number of partitions and α and γ are sparsity factors for the initial KC matrix and L-shaped KC matrix respectively.(Proof omitted) Although, we do lesser work than the sequential algorithm, we are able to achieve comparable results since the L-shaped partitioning focuses the search for the best rectangle. The above heuristics of searching on the L-shape instead of only on the diagonal block improved quality degradation by 50% than that obtained by the algorithm described in the Section 4. In fact, in certain cases, like **seq** on 4 processors, it produces better results than SIS as it choose a better search path than SIS.

MCNC	Initial	2 processors		4 processors		6 processors	
Circuits	LC	LC	**S**	LC	**S**	LC	**S**
dalu	3588	2874	1.99	2935	4.23	3025	6.88
des	7412	6658	2.6	6656	3.13	6653	9.07
seq	17938	9274	1.13	9038	2.34	9255	3.35
spla	24087	17709	1.45	17712	1.54	17717	1.58
ex1010	13977	11836	2.11	11842	7.8	11865	11.48
average	1.000	0.721	1.86	0.719	3.81	0.724	6.47

Table 6: Results of Parallel Algorithm with L-Shaped Partitioning on Shared Memory Multiprocessor

6 Conclusion

Algebraic factorization is an extremely important but computationally intensive part of any logic synthesis system. It is used repeatedly in logic synthesis, hence it is important to look at parallel processing to speed up the procedure. This paper has presented a detailed study of three parallel algorithms for kernel extraction.

The first algorithm uses data 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, 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 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 represents a compromise between the two approaches. It uses a novel L-shaped partitioning strategy which provides some interaction among the rectangles obtained in various partitions. It produces better results than the parallel algorithm using independent partitions, but produces less 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. Otherwise, if quality is of utmost importance, then the parallel algorithm using L-shaped partitioning is the preferred algorithm. It outperforms the parallel algorithm with replicated approach in memory scalability (in terms of ability to synthesize very large circuits), and in runtimes. Thus we have successfully developed parallel algorithms for the minimum-weighted rectangle cover problem. Even though the specific implementation 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 the algorithms are formulated in terms of a rectangular cover problem.

References

[1] 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*, pages 66–69, Santa Clara, CA, Nov. 1987.

[2] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, CAD-6(6):1062–1081, Nov. 1987.

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

[4] 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*, pages 569–578, Santa Barbara, CA, Apr. 1995.

[5] K. De, B. Ramkumar, and P. Banerjee. A portable parallel algorithm for logic synthesis using transduction. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, 13(5):566–580, May 1994.

[6] L. A. Sanchis. Multiple-way network partitioning with different cost functions. *IEEE Trans. Comput.*, 42:1500–1504, 1993.

[7] 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. Technical Report UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, May 1992.

[8] F. Theeuwen. Logic optimization on a concurrent processing computer. In *Proceedings of the European Design Automation Conference*, pages 429–433, Glasgow, UK, Nov. 1990.

[9] F. Theeuwen, J.F.M., and P. van P.T.H.M. Automatic generation of boolean expressions in nmos technology. In *Digest of Papers, International Conference on Computer-Aided Design*, pages 332–334, Santa Clara, CA, Nov. 1985.