IEEE Transactions on Computer Aided Design, May 1994

A Portable Parallel Algorithm for Logic Synthesis using Transduction [1] [2]

Kaushik De[†] **Balkrishna Ramkumar**[‡] **Prithviraj Banerjee**[§]

[†] LSI Logic Corporation
1551 McCarthy Blvd.
MS E-192
Milpitas, CA 95035

Dept. of Elec. & Comp. Engg.
Univ. of Iowa
Engg. Building, Capitol St.
Iowa City, IA 52242

[§] Coord. Science Lab.
Univ. Of Illinois
1308 W. Main St.
Urbana, IL 61801

Corresponding Author:
Prithviraj Banerjee
EMAIL: banerjee@crhc.uiuc.edu
Phone: (217)-333-6564
FAX: (217)-244-5685

Category: LOGIC SYNTHESIS

[1]A preliminary version of this paper appeared in ICCAD 92

[2]Acknowledgement: This research was supported in part by Semiconductor Research Corporartion under grant SRC 91-DP-109, in part by NASA under grant NAG 1-613, and in part by the National Science Foundation PYI grant NSF MIP 87-57563

IEEE Transactions on Computer Aided Design, May 1994

A Portable Parallel Algorithm for Logic Synthesis using Transduction

ABSTRACT

Combinational logic synthesis is a very important phase of VLSI system design. But the logic synthesis process requires large computing times if near optimal quality of the logic network is desired. Parallel processing is fast becoming an attractive solution to reduce the computational time. Recently, researchers have started to investigate parallel algorithms for problems in logic synthesis and verification. Much of the work in parallel algorithms for CAD reported to date, however, suffers from a major limitation. The parallel algorithms proposed for the CAD applications are designed with a specific underlying parallel architecture in mind. Moreover, incompatibilities in programming environments also make it difficult to port these programs across different parallel machines. As a result, a parallel algorithm needs to be developed afresh for every target parallel architecture. The ongoing project of ProperCAD offers an attractive solution to that problem. It allows the development and implementation of a parallel algorithm on the CHARM runtime system such that it can be executed in all the parallel machines without *any* change in the program. In this paper, we describe a portable parallel algorithm for logic synthesis based on the Transduction method, called ProperSYN. This algorithm uses an asynchronous message driven data-flow model of computation, with no explicit synchronizing barriers separating different phases of parallel computation as used in many previously developed parallel algorithms. Our algorithm is therefore more scalable to large numbers of processors The algorithm has been implemented and it runs on a variety of parallel machines. We present results on several benchmark circuits for shared memory MIMD machine like Sequent Symmetry and Encore Multimax, distributed memory MIMD machine like the Intel/860 hypercube and distributed processing systems like networks of SUN workstations.

IEEE Transactions on Computer Aided Design, May 1994

A Portable Parallel Algorithm for Logic Synthesis using Transduction

1 Introduction

Combinational logic synthesis deals with the optimization of logic to realize a specific combinational function in either two level or multilevel form, and typically optimizes the area or delay of the resultant circuit. Many efficient algorithms for two-level logic minimization such as ESPRESSO [1] and for multilevel logic optimization such as SOCRATES [2], MIS [3], SYLON-XTRANS [4], and BOLD [5] have been developed recently. Logic synthesis for medium sized circuits have large computing times and memory requirements. For larger circuits of the future, sequential algorithms will not be able to address their complexity.

Parallel processing is fast becoming an attractive solution to reduce the large amounts of time spent in VLSI circuit design. This has been recognized by several researchers in VLSI CAD as is evident in the recent literature [6]. Recently, researchers have started to investigate parallel algorithms for problems in logic synthesis and verification. Galivanche and Reddy [7] have proposed a parallel implementation of two-level network synthesis based on ESPRESSO. The authors presented parallel versions of the procedures in two-level synthesis, such as reduction, expansion and complementation of cubes, and reported speedups of about 7 on an 8 processor shared memory multiprocessor. Zipfel has proposed a parallel multi-level synthesis algorithm using algebraic factorization procedures of MIS in [8]. He has developed parallel algorithms for kernel extraction and cube extraction procedures within the MIS synthesis program, however, the results of speedups on an 8 processor shared memory multiprocessor were less than 2 on 8 processors. Hachtel et al. [9] have proposed a parallel tautology checking algorithm with application to logic verification. Ma et al. [10] have proposed parallel logic verification algorithms based on implicit enumeration. Both the above work on parallel logic verification were proposed for shared memory multiprocessors and reported good speedups. Lim et al. has recently proposed a parallel logic synthesis algorithm for logic synthesis using the transduction method [11] [12], suitable for execution on a shared memory multiprocessor. Speedups of about 6 were reported on an 8 processor Encore multiprocessor.

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

This limitation has serious consequences. A parallel algorithm needs to be developed afresh for every target parallel architecture. This is compounded by the length of the software development cycle, which, for parallel programs, is considerably longer than for sequential programs. As a result, parallel programs are considerably costlier to develop than sequential programs.

There are two goals of the ProperCAD project [13] (Portable object-oriented parallel environment for CAD applications). The first goal is to develop an environment for writing portable parallel programs for VLSI CAD applications that will run on a range of parallel machines including shared memory multiprocessors such as the Encore Multimax and Sequent Symmetry, distributed memory multicomputers such as the Intel iPSC/860, and networks of workstations. This is being accomplished in two phases. The first phase involved the use of the CHARM language and runtime system which is based on extensions to the C language [14, 15] on top of which we developed CAD application specific libraries to support the development of parallel CAD applications. The second phase involves the integration of the CAD libraries and the machine independent runtime support in C++ in a truly object-oriented manner. The other goal of ProperCAD is to develop the new parallel algorithms for CAD around existing sequential algorithms with a strict interface defining the interactions between the two. This will enable the parallel algorithms to benefit from future improvements in the sequential algorithm.

The ProperCAD approach to the design of parallel CAD algorithms is illustrated in Figure 1. A parallel algorithm is designed around an existing uniprocessor algorithm by identifying modules in the uniprocessor code and designing a well-defined interface between the parallel and sequential code.

Parallel execution in the ProperCAD environment, based on the CHARM runtime system [14, 15], follows a coarse-grained data-flow model and is message driven. Conceptually, a pool of messages is created where each message may either request the creation of a new *object* or be destined for an existing *object*. An *object* can only create new messages or be waken up by messages sent to it at designated entry points. Processes are not allowed to block, or explicitly make receive requests from other processors.

Every message thus represents some work. The machine specific code (see Figure 1) is responsible for dynamically distributing the messages across the available processors. A choice of queuing strategies are also available to the programmer to influence the order in which messages in the pool are processed. The strategy may be depth-first, breadth-first, or priority-based. In the priority-based strategies, the programmer provides priorities with every message, which determines the order in which the messages in the pool are processed.

One process is provided per processor for data abstraction for one type of data [13]. All the processes in that processor can access that data using access routines. These special processes are called *work managers.*

In this paper, we present a new parallel algorithm for logic synthesis using the Transduction method [4] on the first phase of the ProperCAD project, namely, the one based on the CHARM system [14, 15] and is named as ProperSYN.. The successful implementation of SYLON-XTRANS [16] and its variants [17] have proven that the Transduction method is a powerful tool in the synthesis of multi-level networks. However, the Transduction

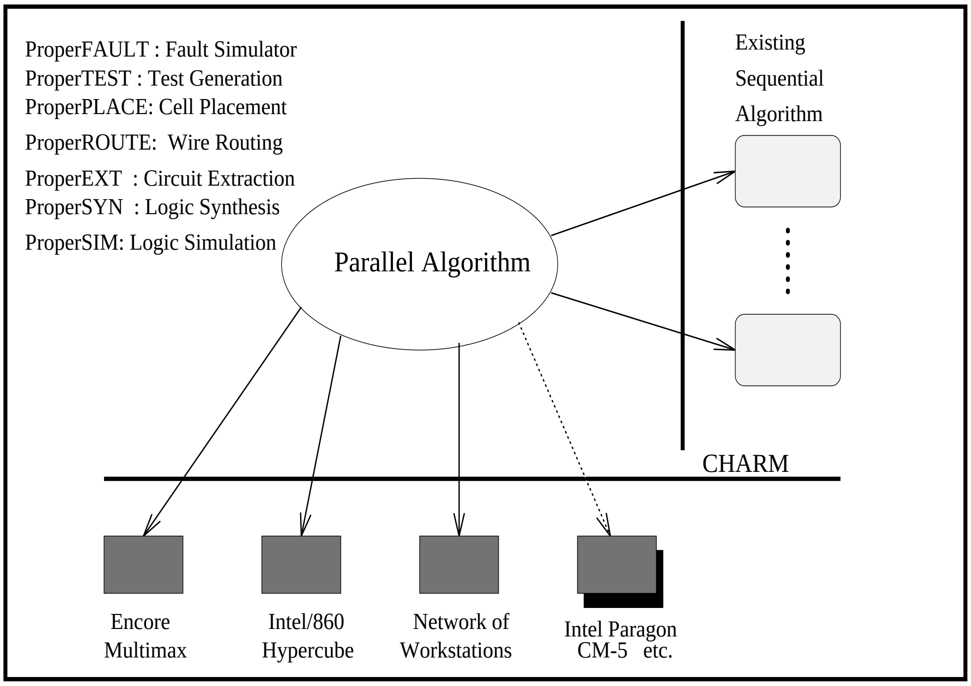

Figure 1: An overview of the ProperCAD project, listing the application under development

method has the disadvantage of large execution times for large networks. We believe that an efficient parallel algorithm for logic synthesis using the transduction method will make the runtimes manageably small.

The parallel algorithm for logic synthesis in the ProperSYN system is novel and different from the parallel algorithm described by Lim et al [11] [12]. Lim's algorithm is suitable for shared memory multiprocessors, and uses a shared circuit description over all processors and explicit barrier synchronization between phases of transformations. Our algorithm uses an asynchronous message driven computing model with no synchronizing barriers separating phases of parallel computation, and is therefore inherently more scalable to larger number of processors. Scalability also depends on other important aspects like *load balancing* and *scheduling*. To achieve good load balancing and scheduling, we create an order of magnitude more jobs than the number of processors available. Those jobs are automatically distributed by the CHARM runtime system. Since the number of jobs is large and the size of job is small, the average work performed by each processor tends to be the same. While creating jobs, we took into consideration the communication to computation ratio to make sure that the communication overhead remains low. Also, our algorithm is portable across a wide variety of parallel architectures. Finally, our algorithm is built around a well defined sequential algorithm interface, so that we can benefit from future expansion of the sequential algorithm. Some preliminary results of this algorithm were reported in [18].

Section 2 provides a brief background on the Transduction method of logic synthesis. Section 3 describes the portable parallel algorithm for logic synthesis developed in the ProperSYN system. Section 4 describes the results of the *same* parallel algorithm on different parallel processing platforms, namely shared memory

MIMD multiprocessors such as the Sequent, the distributed memory MIMD multicomputers such as the Intel i860 hypercube, and distributed environments such as a network of SUN workstations.

2 Overview of the Transduction Method

2.1 Terminology

In this paper, we will consider loop-free multi-level combinational networks consisting of simple gates such as AND, OR, NAND, NOR and NOT gates. However, it is conceptually trivial to extend the methods discussed in our paper to handle more complex gates like XOR and XNOR.

Let g be the number of gates in a multilevel network, $V = \{v_1, v_2, \ldots, v_g\}$ be the set of gates in the network, and $C = \{c_{ij}\}$ be the set of connections where c_{ij} connects the output of gate v_i to an input of gate v_j. A gate v_i is an *immediate predecessor* of v_j if there exists a connection c_{ij}. In that case v_j is an *immediate successor* of v_i. If there exists a sequence of gates through which one can reach the gate v_{k_t} from the gate v_{k_1}, then v_{k_t} is a *successor* of v_{k_1} and v_{k_1} is *predecessor* of v_{k_t}.

A network can be viewed as a directed acyclic graph, consisting of the gates as the nodes in the graph and the connections as the edges in graph. The Transduction method is based on the concept of *sets of permissible functions* [4]. The output function of a gate can be any member of its set of permissible functions without changing the functionalities of the primary outputs of the network. Hence, conceptually, sets of permissible functions are equivalent to the observability don't cares. There are two types of permissible functions, the *maximum set of permissible functions (MSPF)* and the *compatible set of permissible functions (CSPF)*. As its name suggests, the MSPF of a gate or connection contains the largest set of functions which the output of a gate can take without changing the functions of the primary outputs. On the other hand, the CSPF of a gate or connection is a subset of its MSPF, calculated based on some ordering of the inputs at each gate. However, the computation time of CSPF is much shorter than that required for MSPF, though it is less effective. The methods of computation of MSPF and CSPF can be found in [4]. For our parallel algorithms reported in this paper we used CSPF. But one can use MSPF as well without any conceptual change of our algorithms if he/she is ready to accept larger runtimes and memory requirements for better quality of the circuits.

2.2 Types of Transformation

There is a suite of transformations and optimizations that can be applied to the network with the help of the CSPFs of the gates and the connections. The Transduction method consists of four main transformations, namely, *gate substitution*, *gate merging*, *generalized gate substitution* and *gate input reduction*. It also *prunes* redundant connections in the network.

In *gate substitution*, one searches for a pair of gates, v_i and v_j, such that the output function of v_i is a member of the CSPF of v_j. In this case, one can replace all the output connections of v_j by new connections from v_i. Gate v_j, along with the fanout free input cones of v_j are deleted from the network.

In *gate merging*, one searches for a pair of gates, v_i and v_j, such that the intersection of their CSPF is nonempty. One can then try to synthesize a new gate, v_k, whose output function is a member of the intersection of the CSPFs of v_i and v_j. This is achieved by connecting the inputs of v_k to a minimal set of existing gates in the network through the connectability condition [4]. After this, v_i and v_j can both be deleted from the network, along with the fanout free input cones of them.

A more general version of gate substitution is *generalized gate substitution*. In this transformation, if all the output connections of a gate, v_i, can be replaced by output connections from a set of existing gates in the network, then the gate v_i can be removed from the network.

Finally, in *gate input reduction*, one tries to reduce the number of connections in the network. Here, for every gate v_i in the network, a new gate v_i' is tried to be synthesized whose output function is a member of the CSPF of v_i and and has less number of input connections than v_i. The inputs to v_i' are from the outputs of a minimal set of existing gates in the network and such connections can be found by using the effectively connectability condition [4]. With these transformations, the Transduction method is able to produce near optimal networks [16].

3 Parallel Algorithms for Transduction Method

3.1 Parallelism using Partitioning

An obvious way of extracting significant speedup out of the logic synthesis application is to generate a large number of partitions and to synthesize each partition independently. The results of the individual partitions can then be merged back. Unfortunately, such an approach has the problem that with increasing number of partitions, the quality of the overall network degrades. Specifically, it has been shown that even if the individual partitions are synthesized perfectly such that they are each completely irredundant, the resultant network can still have redundancies when the synthesized partitions are reconnected to form a larger network, [19]. This is 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. In the interest of better quality, one should, therefore, choose a minimum number of partitions. Then, one is forced to resort to parallelism of operations within each partition which is much harder to exploit. One may not get good speedups within a partition. There is clearly a tradeoff between result quality and execution time as determined by an optimal number of partitions. Such a theory needs to be developed but is outside the scope of this paper.

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

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

Figure 2: The partitioning algorithm

3.2 Our Approach using Work Decomposition

In our ProperSYN algorithms, we do not use the partitioning approach. Although, in our approach the logic network is divided into a set of non-overlapping partitions, the partitioning is for *parallelization purposes only*, the transformations and the optimizations are performed on the entire network.

We used a very simple partitioning strategy. The user specifies the maximum number of gates per partition, *max_gates*. Initially all primary outputs of the given circuit are put in a list called *cone_tips*. Then a node η is picked from the list *cone_tips* and the transitive fanins of η which do not have their partition assigned yet are put into a new partition Π in breadth-first manner until the number of nodes in Π reaches *max_gates* after which a new partition Π' is formed. Then all the fanin nodes to nodes in Π are put at the end of *cone_tips*. This procedure continues until all the nodes in the network are assigned to some partition. The algorithm is shown in Figure 2.

For each partition, one *object* is created. Processes that perform various operations on these objects are distributed to different processors using some load balancing methods supported by the CHARM runtime system, as depicted in Figure 3. Typically, we create a lot of small partitions such that load balancing is good. But we make sure that the amount of computation required by each partition is roughly an order of magnitude

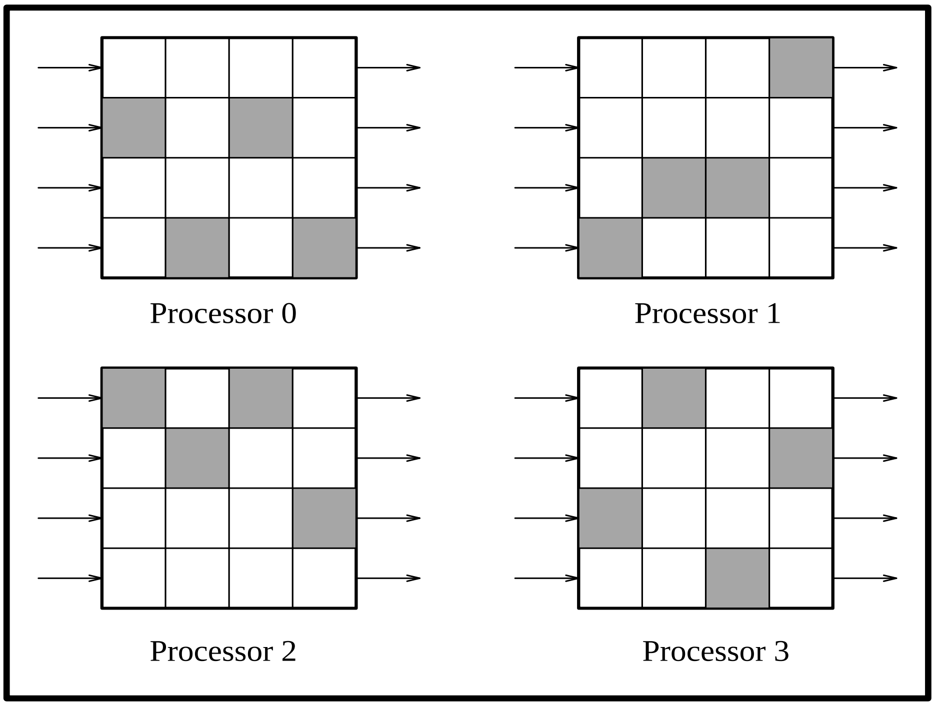

Figure 3: Partitions of a Network and the "ownership" of each processor

higher than that of communication time for sending a message between objects. We will denote the object for a partition p as *partition_object* p. Each processor has a copy of the network, but has the responsibility for the shaded regions of the network. Effectively, we can say that each processor "owns" some parts of the network.

The quality of the resultant circuit does not vary with the partitioning strategy beyond the statistical variation shown in Section 4.2 because the partitioning is only used for the purpose of division of work. The evaluation time for the output function and the CSPF depends on the partitioning strategy. But determining a good partitioning strategy for the minimization of the evaluation times for the output function and the CSPF is a non-trivial procedure and can increase the overall runtime of any synthesis algorithm. The evaluation times for the output functions and the CSPFs is are small part of the total runtime of the algorithm. Hence, we decided to use a simple partitioning strategy.

3.3 Parallel Evaluation of Output Function

For optimization purposes, we need to evaluate the output functions at each gate and connection in the network. The output functions need to be expressed in terms of the primary inputs of the network. We compute output functions using binary decision diagrams (BDD) [20].

A major problem we have to deal with is the fact that since different processes perform different optimizations simultaneously on the network, the output functions and the CSPFs keep changing. It is important to provide some coherence mechanism among various applications of these optimizations in parallel. To do

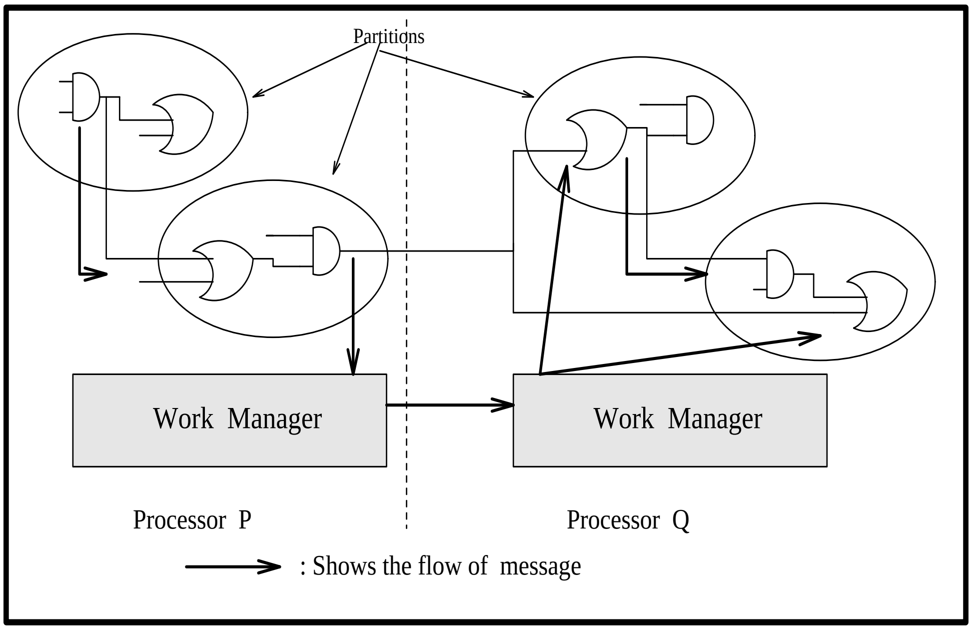

Figure 4: Evaluation of Output Functions and the Flow of Messages

that with each BDD, we keep a tag, called the *version*. Also, with each gate and connection, we keep the current version number of the output function and the CSPF. A BDD is "valid", if its version number is current. Any transformation in the network changes the output functions and the CSPFs in some parts of the network. The version numbers are used to prevent any "illegal" transformation done using an "invalid" BDD. Initially, the versions of output functions and the CSPF are set to 1. If due to any transformation, the output function or CSPF of a gate or connection becomes "invalid", the corresponding version number attached to it is incremented by 1.

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

After the output function of a gate ζ is evaluated in a partition, it is checked if its output function is needed in any other partition. If the other partition that needs the output function is in the same processor, that *partition_object* is informed about the availability of the output function of ζ. But if it is in a different processor, then the current *partition_object* invokes the *work manager* in the current processor, which in turn transmits the BDD to the *work manager* of the destination processor and then the *work manager* in the destination processor sends a message to the corresponding processes in that processor about the availability of the output function of ζ. We use the *work manager* to ensure that only one copy of the BDD is sent across the processor boundary. The procedure is illustrated in Figure 4.

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

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

3.4 Parallel Evaluation of CSPF

The flow of evaluation of CSPF moves from the primary outputs to the primary inputs. If the output function of a primary output is available, its CSPF can be calculated immediately. If no *external don't care* [21] is specified for that primary output, then the CSPF is the same as the output function, otherwise the CSPF is the union of the output function and the external don't care.

The CSPF of an input connection to a gate can be computed if the output functions of all the input connections to that gate is available as well as the CSPF of that gate. Then the CSPF of the connection is computed as a function of the type of the gate, the CSPF of the gate, the output functions of the sibling connections and the output function of the connection itself [4]. The CSPF of a gate can be computed if the CSPF of all the output connections are available. The CSPF of a gate is the set intersection of the CSPFs of its output connections, [16].

The partitions which have any primary outputs start evaluation of the CSPF at the primary outputs. After evaluation of an input connection, it is checked if it is connected to the output of a gate which is in a different partition and the partition is in the same processor; then that *partition_object* is informed about the availability of the CSPF of the connection. If the partition is in a different processor, the CSPF is transmitted to the other processor through the *work managers* of the two processors similar to the case of the evaluation of output functions. The procedure is explained in Figure 5.

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

3.5 Parallel Gate Substitution

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

Let us denote p is the number of *partition_objects* created. Then the *partition_object* x will create another

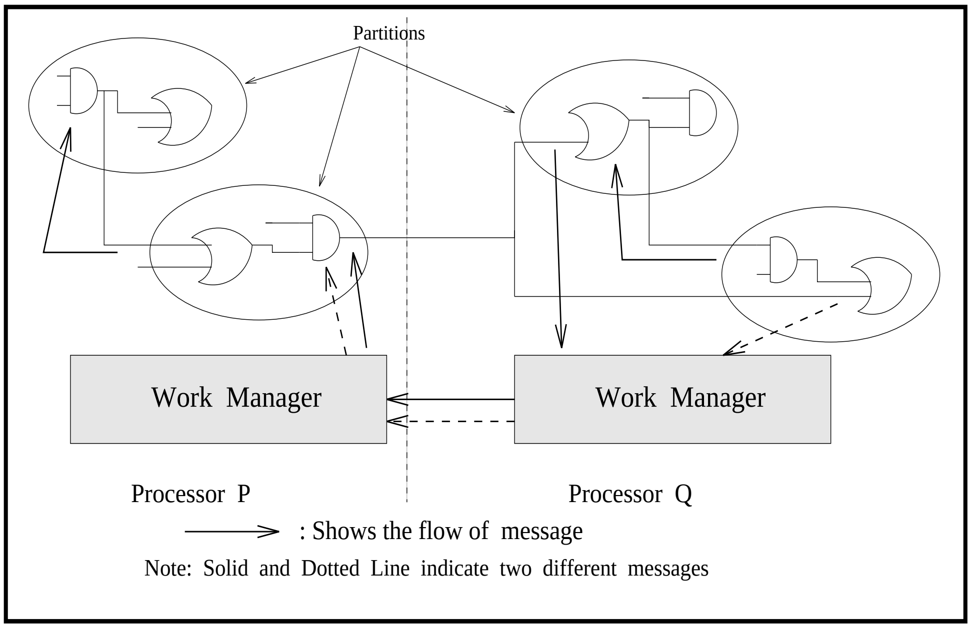

Figure 5: Evaluation of CSPF and the Flow of Messages

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

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

The procedure is depicted in Figure 6.

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

When the *gate_sub_object* denoted by (i, j) is awakened by a message, it starts by collecting the output

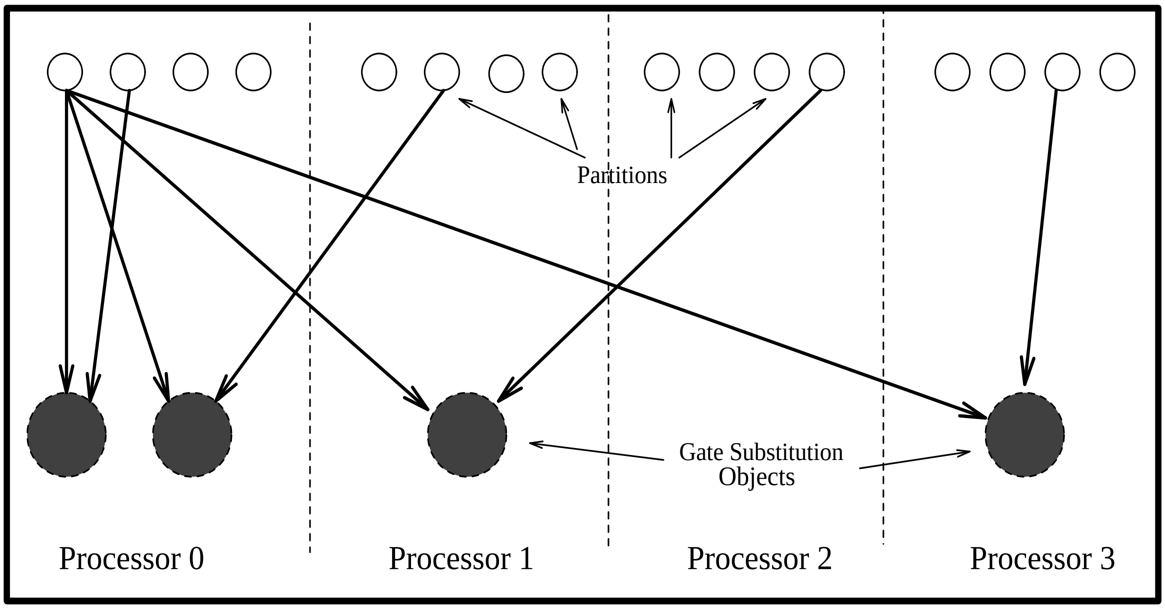

Figure 6: Creation of objects for gate substitution

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

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

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

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

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

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

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

Figure 7: Algorithm for Parallel Gate Substitution

of a gate in partition i. The algorithm is given in Figure 7.

We want to emphasize the fact that in our parallel algorithm, we check for all possible gate substitutions, as done in the sequential algorithm. We explain that fact with some examples. Consider the circuit shown in Figure 8(a) with 4 gates (A, B, C and D). If we perform all possible comparisons as shown using the directed edges, there are 12 comparisons in total. Let us assume that we partition the circuit into two partitions, where partition 1 contains A and B and partition 2 contains C and D. Then our parallel algorithm will generate 4 *gate_sub_objects*. In Figure 8(b), we have shown the comparisons performed by different *gate_sub_objects*. One can observe that total number of comparisons performed by them is 12, which is the same as the number of comparisons performed by the sequential algorithm.

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

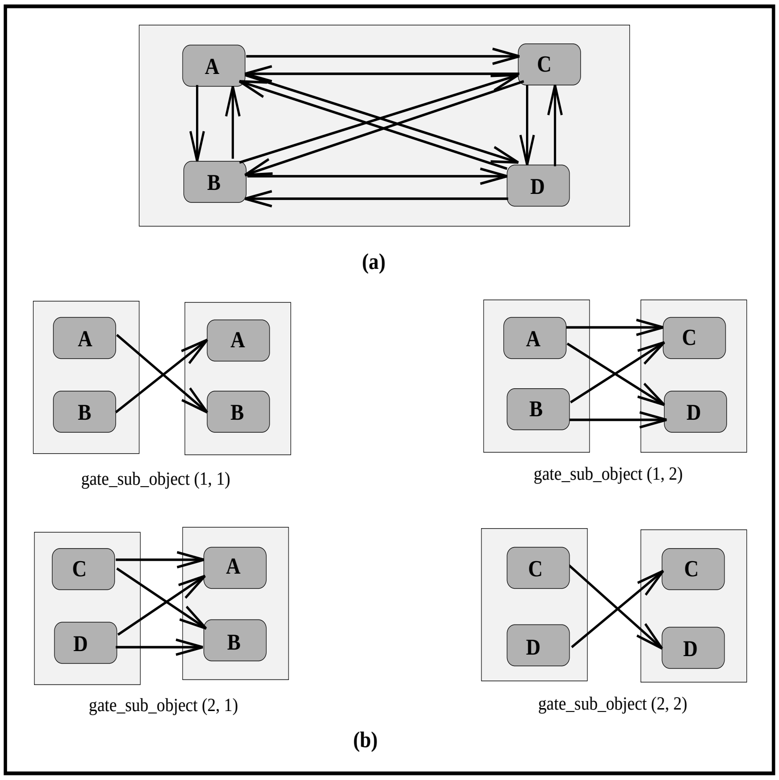

Figure 8: Comparisons performed by (a) the sequential algorithm and (b) our parallel algorithm for gate substitution. It shows that the number of comparisons is the same for both the algorithms.

then itself performs the substitution. It is through this centralized update, that coherence is maintained in data updates and it is guaranteed that no errors are made in the optimization process.

After the substitution is performed, the output functions of all the *successors* of the substituted gate will become invalid as well as the CSPFs of the *predecessors* of the substitute gate. This is shown in Figure 9(a). But if the output functions are reevaluated in the region where they are marked, then the CSPFs of all the gates in the transitive fanins of that region will get invalidated. Since in our approach, we do not have any synchronizing barrier to decide when to start reevaluation of the output functions and the CSPFs, the reevaluation can start any time in any order. So we mark the invalidation of output function and the CSPFs as shown in the Figure 9(b). Invalidation of the output function or the CSPF is done by incrementing the current version number attached to the corresponding gates and connections such that any BDD in the system with lower version number will be considered “invalid”.

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

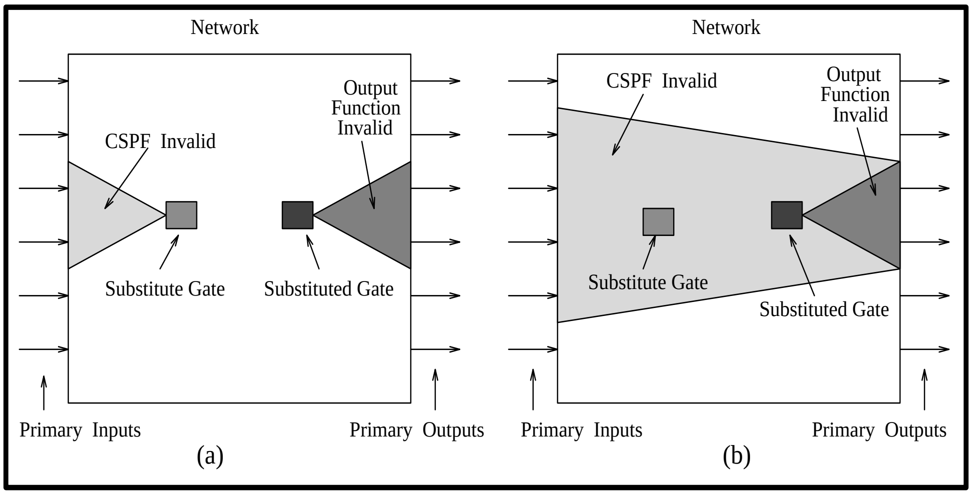

Figure 9: Invalidation of Output Function and CSPF due to gate substitution

3.6 Parallel Gate Merging

Similar to gate substitution, gate merging is an iterative improvement procedure. In this transformation, we search for a pair of gates, (g_1, g_2) and try to synthesize a third gate g_3 which can replace both the gates. Two gates, g_1 and g_2, are mergeable if the intersection of the CSPFs of the two gates is non-empty. Then we try to form a third gate with four simple gate types: AND, OR, NAND and NOR. For each gate type, we try to connect the inputs of the gate from the minimal set of outputs of the existing gates in the network. If we succeed in forming such a gate, we can replace both the gates, g_1 and g_2, by the new gate. However, in order to maintain the loop-free condition of the network, the inputs of the new gate, g_3, must not come from the successors of g_1 and g_2. Hence, to maximize the chance of gate merging, we prefer the gates g_1 and g_2 to be close to the primary outputs.

Let us denote p be the number of partitions. For each partition pair, (i, j), an object is created of type *gate_merge_object*. The *gate_merge_object* denoted by (i, j) is responsible for testing for any gate merging between any gate in the partition i and any gate in the partition j. In addition to that, a number (equal to p) of *gate_merge_objects* will be created, one for each partition to check for any possible gate merging in the same partition. So, in total $\binom{p}{2} + p$ objects are created. The procedure is depicted in Figure 10.

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

A *gate_merge_object* behaves almost the same manner as a *gate_sub_object* if it does not find any CSPF. It requests the *work manager* in the current processor, which sends a message to the *work manager* to the corresponding processor if needed. Whenever that CSPF is available, the *work manager* will inform this object. If this object is finished with searching for gate merging pairs with available CSPFs, it requests the

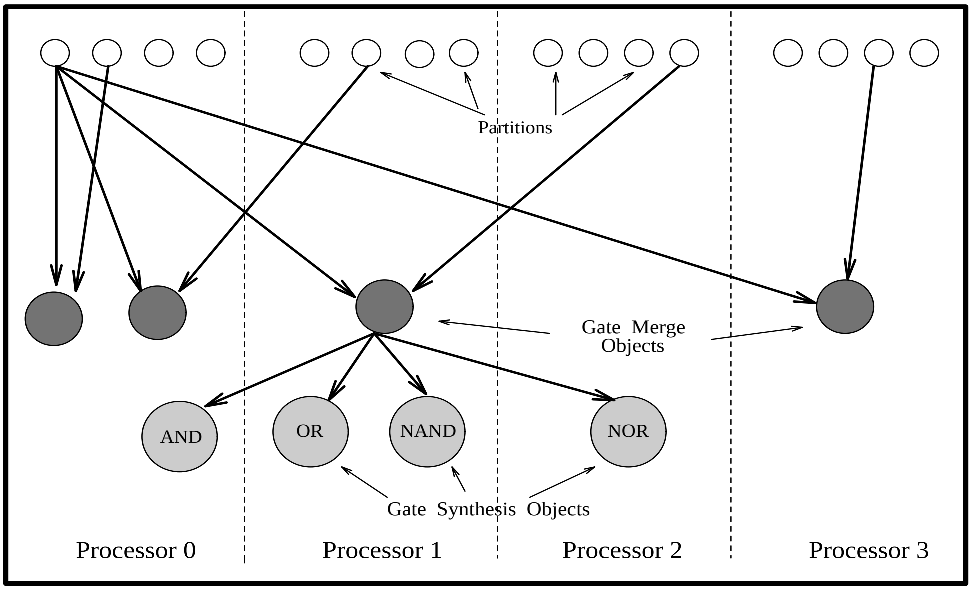

Figure 10: Creation of gate merge objects and creation of objects for gate synthesis

corresponding *partition_objects* to inform this object whenever the CSPF in those partitions change. We are omitting the details to avoid the repetition of the description of the almost similar procedure. The algorithm is described in Figure 11.

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

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

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

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

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

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

Figure 11: Algorithm for Parallel Gate Merging

manager, it terminates itself.

When a *gate_synthesis_object* receives an output function, it again checks if both the gates which are to merged exist and they have the same CSPF version number. It then checks if the output function can be connected to the synthesized gate. If it is possible to connect to the synthesized gate, it checks for all the gates in the *partial_input_list* to ensure that the version number of the output functions has not changed. If there is no change, it just adds this new connection and updates the partial output function and *partial_input_list*. If any change has happened, it has to discard the partial output function and perform the computations again. The algorithm is described in Figure 12.

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

```
Algorithm: Parallel Gate Synthesis (g1, g2, type)
/* This object tries to synthesize a gate of type 'type' and whose output function */
/* is a member of the intersection of CSPFs of g1 and g2 */

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

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

Figure 12: Algorithm for Parallel Gate Synthesis of a given type

3.7 Parallel Pruning

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

3.8 Parallel Generalized Gate Substitution

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

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

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

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

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

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

When the *master* processor receives a request for permission for generalized gate substitution, it checks if the gates and connections exist in the network. Then it checks for the version number of the BDDs. If they have not changed, the substitution is permitted. If any version number has changed, the *gen_gate_sub_object* is asked to check for substitution again. The algorithm is given in Figure 13.

3.9 Parallel Gate Input Reduction

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

To synthesize a new gate which can replace an existing gate γ in the network, the output function of the new gate must be a subset of the CSPF of γ. The input connections to the new gate are from the existing gates in the network. To check if a gate can be connected to the new gate which has a given CSPF, the *effectively connectable* condition is used as given in [4].

For each gate in the network, four *gate_inp_red_objects* are created, one for each of the gate type: NAND, NOR, AND and OR. Each *gate_inp_red_object* tries to synthesize a new gate of type given to it during its

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

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

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

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

Figure 13: Algorithm for Parallel Generalized Gate Substitution

creation. The number of input connections to the new gate must be less than that to the existing gate. Other than that, the parallel algorithm is almost similar to that of gate synthesis algorithm described in context with gate merging. Hence, the details of this algorithm are omitted for the sake of conciseness.

3.10 Overview of the ProperSYN Algorithm

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

We should mention that the ProperSYN system is not the same algorithm as the Transduction system. In the transduction system, the user specifies the *order* in which different transformations are to be applied at the beginning and the transformations are applied strictly in that order, one after the other. A typical run of the Transduction method is shown in Figure 14(a). The order in which different transformations are applied is determined by the cost-performance ratio of each transformation. For example, the *gate substitution* has the lowest cost-performance ratio, i.e., checking for gate substitution is the cheapest operation. Hence, the *gate substitution* transformation is applied first on the network to reduce the size of the network. Then one applies the more expensive transformations like *gate merging* to further reduce the size of the network. On the other hand, if one chooses to apply the *gate merging* transformation first on the network, it will take significantly more time to reduce the network.

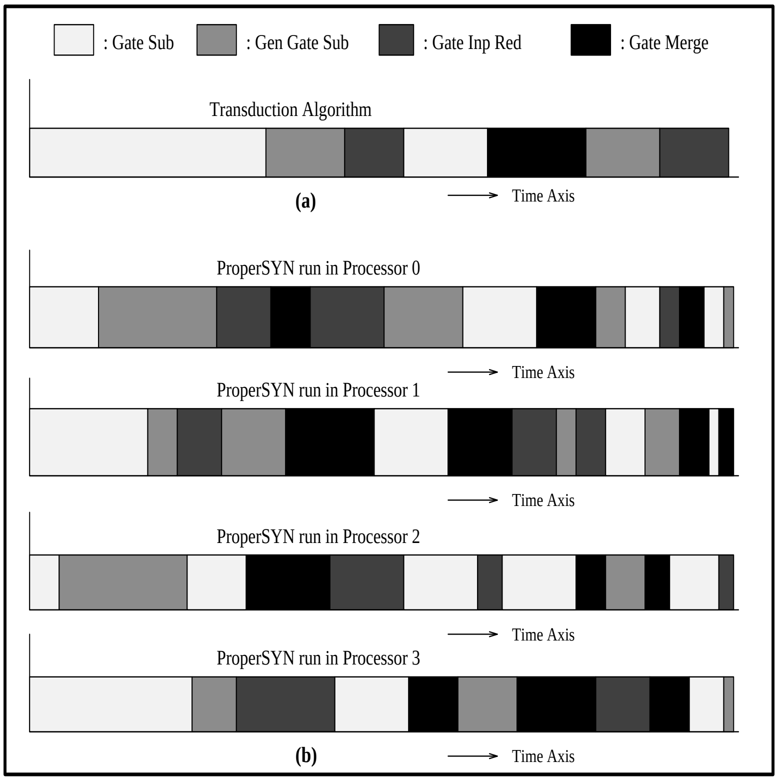

Figure 14: (a) A Typical run of the Transduction method, (b) A typical run of the ProperSYN with 4 processors

In case of the ProperSYN, although we use the same transformations as the Transduction method, applications of the different transformations are intermixed. All the transformations are simultaneously started to avoid any synchronizing barrier. However, the gains of a particular ordering is preserved by the choice of priorities. We experimented a lot with the actual priorities until we found good results. We assign different priorities to different transformations according to their cost to performance ratio to guide the synthesis process. The list of transformations in the decreasing order of priorities : *gate substitution, generalized gate substitution, gate input reduction* and *gate merging*. It is the same order in which the transformations are applied in the Transduction system. There is no *synchronizing barrier* at the end of each transformation. Each transformation is applied repeatedly until there in no more change in the network. A typical run of ProperSYN on 4 processors is shown in Figure 14(b). This figure emphasizes the fact that different processors may be trying different transformations at any particular time.

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

Table 1: Run Times and Speedups in Encore Multimax (Shared Memory MIMD)

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

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

4 Results

The parallel algorithms developed in the last section have been implemented on the ProperCAD environment and is called ProperSYN. The system contains approximately 12000 lines of C code.

4.1 Run Times and Speedups on Different Parallel Machines

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

Table 1 presents the result obtained in an Encore Multimax machine. It is a *shared memory MIMD* machine with 8 processors. Table 2 presents the results obtained in a Sequent Symmetry machine. It is another *shared memory MIMD* machine. Table 3 presents the results obtained in an Intel/860 machine. It is a *distributed memory MIMD* machine with 8 processors connected by a hypercube configuration. Table 4 presents the results in a network of workstations environment. This behaves as a *distributed processing* system. There is a *host* workstation, which distributes work to other workstations. The workstations used in this experiments are SUN4 workstations with SPARC 1 processor in them. In all the tables, we present run times for various

Table 2: Run Times and Speedups in Sequent Symmetry (Shared Memory MIMD)

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

Table 3: Run Times and Speedups in Intel i860 Hypercube (Distributed Memory MIMD)

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

Table 4: Run Times and Speedups in Network of Workstations (Distributed Processing)

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

MCNC benchmark circuits on different number of processors and speedup with respect to 1 processor runtime. One can observe that sometimes our algorithm produces "super-linear" speedups. This is due to the fact that synthesis is search problem and in parallel implementation of a search problem, *speedup anomaly* is possible. If one transformation preceeds another transformation, the search space may be reduced considerably, which may result in a "super-linear" speedup.

4.2 Comparison of Quality

In this section, we will compare the quality obtained by ProperSYN with that obtained by the Transduction method [4] version 1.2 and MIS 2.2 [3]. Since the ProperSYN requires the network to be consists of simple gates like AND, OR, NAND, NOR and INVERTERS, we first perform a *technology mapping* of the given network using a simple gate library which contains only simple gates with at most 5 fanins. The purpose of this technology mapping is *only* to create the initial network, not for any other purpose. The transduction method generates the initial network using SYLON_XTRANS_INI [4], but we have not implemented that. Then ProperSYN is applied to that network and at the end it is again technology mapped. In the case of MIS 2.2, we used the boolean script to perform synthesis of the network [22]. Then we performed *technology mapping* by using the same simple gate library. The transduction method, XTRANS 1.2, was run with the script provided and then technology mapped. We present the results of different benchmark circuits in terms of the number of gates and the number of literals, gate/lit, in the final network in Table 5. As it can be observed from the results, ProperSYN produced better results than MIS 2.2 for almost all the circuits. The results produced by ProperSYN and XTRANS 1.2 were comparable.

Table 6 shows the variation of the quality of the circuit produced with different number of processors. The result is shown for only one type of machine and we have chosen the Intel/860 machine for that purpose.

Table 5: Comparison of Quality of Network Produced by ProperSYN, the Transduction Method and MIS-II

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

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

The result indicates that there is not much variation in the quality of the circuits and sometimes better circuits are produced with larger number of processors. That can be due to different orders in which different transformations are tried. The variations were similar for other parallel machines. The variation of quality with different number of processors in different parallel machines is shown in Table 7. The variation in the result from machine to machine is due to the non-deterministic nature of our algorithm and has to again do with the fact that different processor speeds on different machines can produce different ordering of successful transformations.

Due to the non-deterministic nature of our algorithm, the quality and runtime varied slightly from run to run for more than one processor. We performed an experiment to determine the distribution of the variations of the quality and the runtime. For the benchmark circuit *bw*, we ran our algorithm in Encore Multimax for 20 times each for varying number of processors (2, 4 and 8). The distribution of the number of gates in the final circuit and the run time is shown in Figure 15.

5 Conclusions and Future Research Direction

In this paper, we have presented a portable parallel algorithm of logic synthesis, called ProperSYN. ProperSYN has been implemented as a part of the ProperCAD environment [13] based on the CHARM system [14, 15] and it runs on variety of parallel machines like Sequent Symmetry, Encore Multimax, Intel/860 Hypercube and network of workstations. The algorithm developed for ProperSYN uses an asynchronous message driven data-flow model of computation. No explicit synchronizing barrier is allowed in the algorithm, hence our algorithm

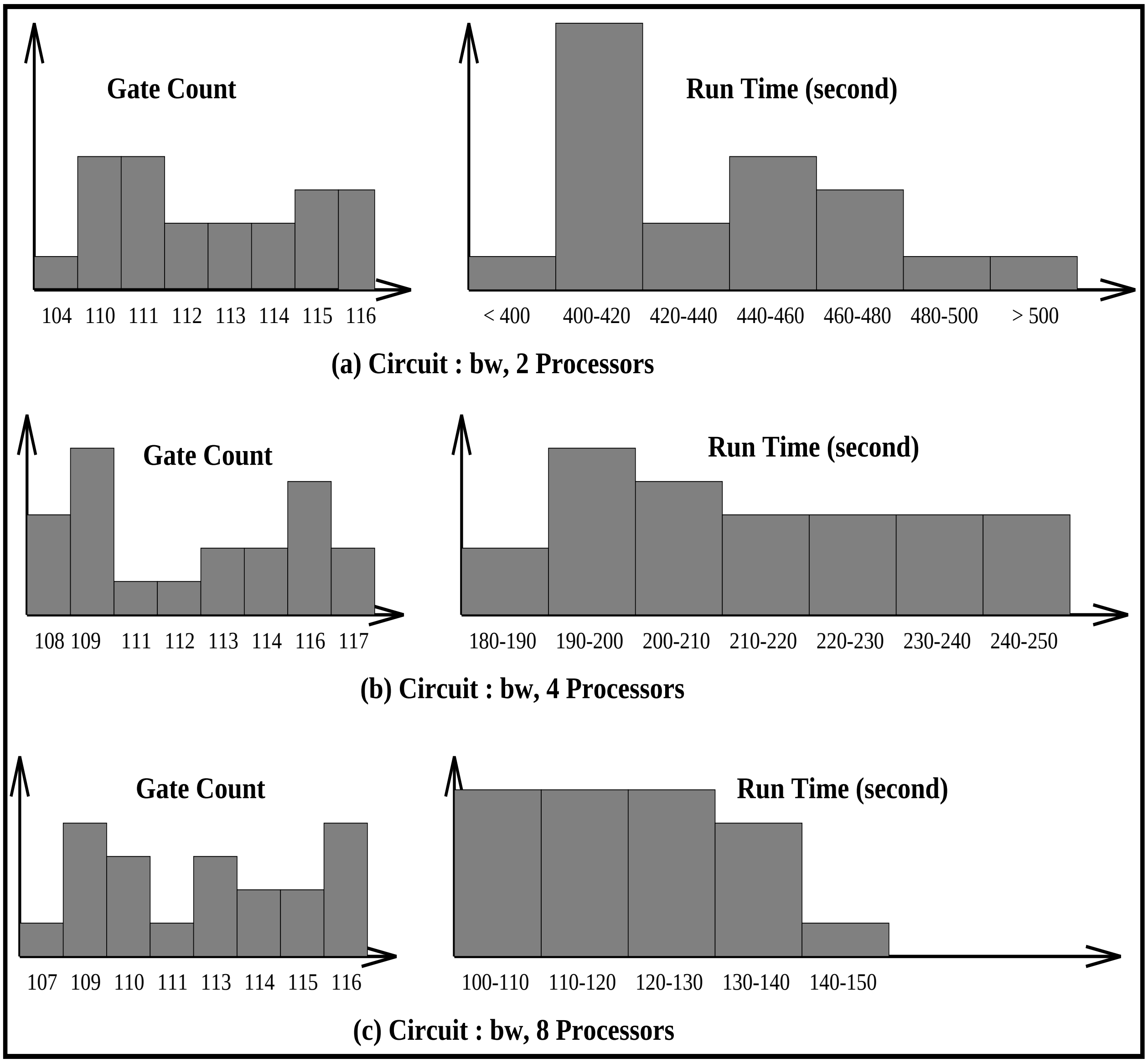

Figure 15: Distribution of the number of gates and the runtime for circuit *bw* in Encore Multimax for 20 runs with (a) 2 processors, (b) 4 processors and (c) 8 processors

Table 6: Comparison of Quality of Network with different number of processors in Intel/860

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

Table 7: Comparison of Variation of Quality of Network $5xp1$ for Intel/860, Encore Multimax and network of Workstations

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

is developed to scale well with larger number of processors. We have implemented the algorithm and have evaluated it on various *MCNC* synthesis benchmark examples. Some results are reported on different parallel machines. Results show excellent speedup in all the machines with almost no degradation in the quality of the synthesized network over the uniprocessor algorithm.

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

For really large circuits when the BDD size becomes too large to fit in the memory, we plan to use partitioning techniques based on [19] similar to the work reported by Lim et al [11, 12].

ProperSYN only has the transformations based on the Transduction method at this moment. Later, we want to add more transformations like algebraic factoring used in synthesis systems such as MIS to ProperSYN such that it becomes more powerful.

6 Acknowledgement

We are grateful to C. F. Lim and Professor S. Muroga for various discussions on the transduction method of logic synthesis. We are grateful to Professor L. V. Kale for the use of the CHARM system. We are also thankful to Argonne National Laboratory for giving us the access to their parallel machines.

References

[1] R. K. Brayton and et al., "ESPRESSO-II: A New Logic Minimizer for Programmable Logic Arrays," *CICC*, pp. 370–376, June 1984.

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

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

[4] X. Xiang, *Multilevel Logic Network Synthesis Systems, SYLON-XTRANS*. PhD thesis, Univ. of Illinois, 1990.

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

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

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

[8] G. Zipfel, "Parallel Algorithm for Algebraic Factorization with Application to Multi-Level Logic Synthesis," Master's thesis, Univ. of Illinois, 1991.

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

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

[11] C.-F. Lim, "Parallel Algorithm for Multi-Level Logic Synthesis Using the Transduction Method," Master's thesis, Univ. of Illinois, 1991.

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

[13] B. Ramkumar and P. Banerjee, "ProperCAD: A Portable Object-oriented Parallel Environment for VLSI CAD," *International Conference in Computer Design*, 1992, Also Submitted to *IEEE Transactions on CAD*.

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

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

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

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

[18] K. De, B. Ramkumar, and P. Banerjee, "ProperSYN: A Portable Parallel Algorithm for Logic Synthesis," *To appear in International Conference in Computer-Aided Design*, 1992.

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

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

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

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