PARALLEL ALGORITHMS FOR STATE ASSIGNMENT OF FINITE STATE MACHINES

BY

GAGAN HASTEER

B.Tech., Indian Institute of Technology, New Delhi, 1994

THESIS

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

Urbana, Illinois

PARALLEL ALGORITHMS FOR
STATE ASSIGNMENT OF
FINITE STATE MACHINES

Gagan Hasteer, M.S.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1996
Prithviraj Banerjee, Advisor

With the rapid improvement in VLSI technology, circuit design is becoming extremely complex and is placing increasing demands on CAD tools. Parallel processing is becoming an attractive solution to reduce the inordinate amount of time spent in VLSI circuit design. The use of parallel platforms in the solution of significant computing problems remains largely restricted to a set of well-structured, numeric applications. This thesis addresses the design and implementation of parallel state assignment techniques for large Finite State Machines. We present two parallel algorithms which are portable across a wide range of MIMD architectures using the ProperCAD II platform which has been specifically designed for portable parallel CAD applications. Results of both algorithms have been evaluated on a SunSPARCServer1000 shared memory multiprocessor, CM-5 distributed memory multicomputers from Thinking Machines and a network of SUN workstations.

The first algorithm, ProperSTATE, is based on MUSTANG, a serial implementation from Berkeley. This algorithm is memory scalable and handles very large Finite State Machines which MUSTANG fails to handle. The speedups are superlinear and the quality degradation is minimal. The second algorithm, ProperJEDI, is based on a more recent serial implementation JEDI. It is a parallel simulated annealing based encoding scheme which takes much more time than ProperSTATE but yields better quality. The two algorithms provide the designer with a rich choice of optimizing the time-quality product in the design cycle.

PARALLEL ALGORITHMS FOR
STATE ASSIGNMENT OF
FINITE STATE MACHINES

Gagan Hasteer, M.S.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1996
Prithviraj Banerjee, Advisor

With the rapid improvement in VLSI technology, circuit design is becoming extremely complex and is placing increasing demands on CAD tools. Parallel processing is becoming an attractive solution to reduce the inordinate amount of time spent in VLSI circuit design. The use of parallel platforms in the solution of significant computing problems remains largely restricted to a set of well-structured, numeric applications. This thesis addresses the design and implementation of parallel state assignment techniques for large Finite State Machines. We present two parallel algorithms which are portable across a wide range of MIMD architectures using the ProperCAD II platform which has been specifically designed for portable parallel CAD applications. Results of both algorithms have been evaluated on a SunSPARCServer1000 shared memory multiprocessor, CM-5 distributed memory multicomputers from Thinking Machines and a network of SUN workstations.

The first algorithm, ProperSTATE, is based on MUSTANG, a serial implementation from Berkeley. This algorithm is memory scalable and handles very large Finite State Machines which MUSTANG fails to handle. The speedups are superlinear and the quality degradation is minimal. The second algorithm, ProperJEDI, is based on a more recent serial implementation JEDI. It is a parallel simulated annealing based encoding scheme which takes much more time than ProperSTATE but yields better quality. The two algorithms provide the designer with a rich choice of optimizing the time-quality product in the design cycle.

DEDICATION

To my parents

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Prithviraj Banerjee, for supporting and directing this work.

I would like to express my thanks to the members of the ProperCAD and Paradigm projects for both their technical help in issues of parallel computing and their camaraderie. I would especially like to thank John Chandy, my officemate, who helped me learn how to program on the ProperCAD platform for which there was no documentation.

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

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

With the rapid improvement in VLSI technology, circuit design is becoming extremely complex and is placing increasing demands on CAD tools. Parallel processing is fast becoming an attractive solution to reduce the inordinate amount of time spent in VLSI circuit design [1]. This has been recognized by several researchers in VLSI CAD as is evident in the recent literature for cell placement [2], circuit extraction [3], test generation [4], and logic synthesis [5].

In this research, we have developed two parallel algorithms for the State Assignment of Finite State Machines. The implementation is defined in terms of C++ classes, based on the ProperCAD II library. The parallel applications have been incrementally developed from an existing state-of-the-art serial application. In this chapter we first proceed to give an overview of the ProperCAD project in §1.1. Then we describe the State Assignment problem in §1.2. We subsequently describe some related work in State Assignment in §1.3. Next, we describe the motivation for parallel implementation of State Assignment algorithms in §1.4. In §1.5, we review previous work done in parallel simulated annealing which is a very successful but compute intensive technique for the state encoding phase of the State Assignment problem. We end the chapter with a summary of the contributions of this thesis.

1.1 An Overview of the ProperCAD Project

The major goal of the ProperCAD project [6] is to develop portable parallel algorithms for VLSI CAD applications that run on a range of parallel machines including shared memory multiprocessors such as the SGI Challenge and the SunSparcServer1000, distributed memory multicomputers such as the Intel Paragon, and the Thinking Machines' CM5, and a network of workstations. This has been accomplished in two phases. The first phase, ProperCAD I, involved the use of the C-based CHARM language and runtime system [7]. CAD application specific libraries were developed on top of CHARM to support the development of parallel CAD applications. Applications developed using ProperCAD I include circuit extraction [8], test generation [9], logic synthesis [10] and cell placement [11].

The second phase, ProperCAD II, involved the integration of the CAD libraries and the machine independent runtime support in a truly object–oriented manner [12, 13]. This environment consists of a C++ library which provides a fully object–oriented parallel interface based on the actor model [14] of concurrent object–oriented computing. Applications created using this newer

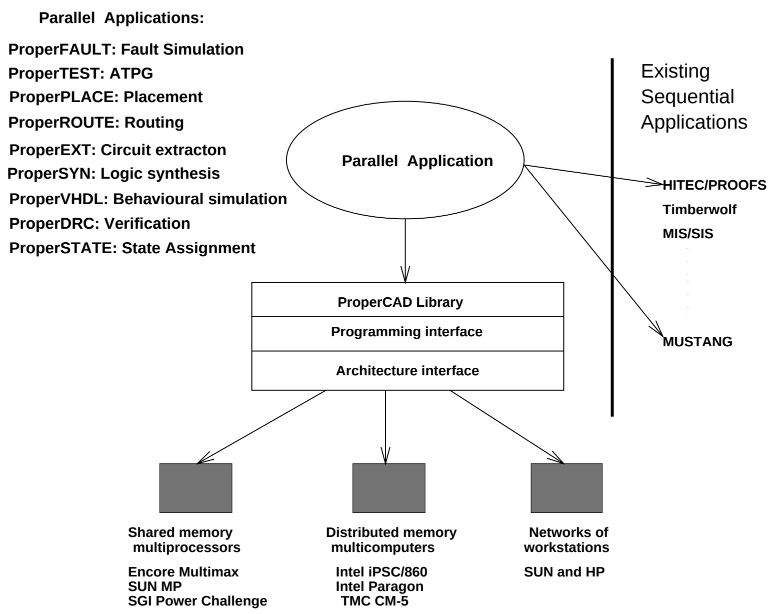

Figure 1.1 An overview of the ProperCAD project

environment include test generation [15], logic synthesis [5], fault simulation [16], placement [17] and state assignment which will be described further in this paper. (See Figure 1.1)

Parallel execution in the ProperCAD II environment is based on a fundamental object called an actor. An actor object communicates with other actors by sending messages, and all actor actions are in response to these messages. Specific actor methods are invoked to process each type of message, and actors are not allowed to block or explicitly make receive requests from other processors. Also concurrent abstractions known as aggregates [18] are available in ProperCAD II to support a multi-access interface to groups of actors.

1.2 Preliminaries

In this section we review some basic definitions and then describe the motivation for the efficient State Assignment of Finite State Machines.

1.2.1 Definitions

A *variable* is a symbol representing a single coordinate in the Boolean space (e.g. a). A *literal* is a variable or its negation (e.g. a or $\bar{a}$). A *cube* is a set of literals and represents the Boolean function corresponding to their product (e.g. $\{a, \bar{b}\}$ is a cube representing the product $a\bar{b}$). An expression is a set of cubes and represents the Boolean function corresponding to their disjunction (e.g. $\{\{a, \bar{b}\}, \{b, \bar{a}\}\}$ is an expression representing the function $a\bar{b} + b\bar{a}$).

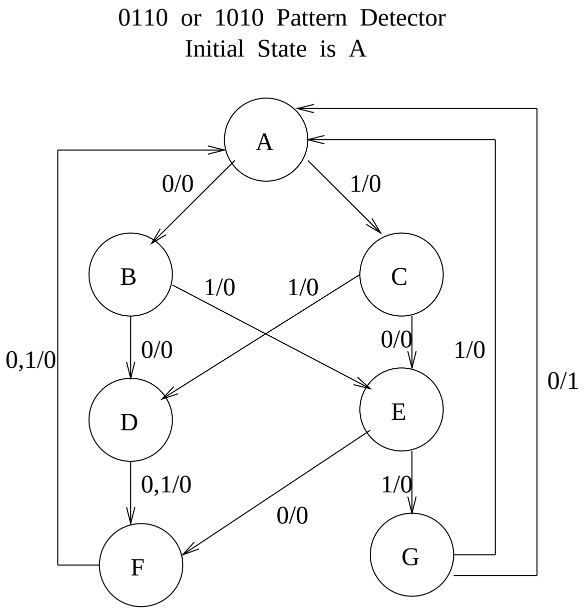

Figure 1.2 STG for an FSM (pattern detector)

The **Hamming distance** $H(c_i, c_j)$ between two codes c_i and c_j equals the number of bits in which they differ. It can also be interpreted as the number of empty literals in the intersection of the corresponding cubes that the codes represent. Conversely, the **Proximity** $P(c_i, c_j)$ between two codes c_i and c_j equals the number of identical bits (the number of non-empty literals in the intersection of the corresponding cubes).

The most commonly used representation of an FSM is a *State Transition Graph* (STG). An STG G(V,E,I(E),O(E)) is a set of vertices V connected by directed edges belonging to E. Each edge is labelled by two fields : an input field I(E) and an output field O(E). The set of vertices are in 1-1 correspondenc with the states of the FSM. A directed edge (v_i, v_j) joins v_i to v_j if there is a transition from the state corresponding to v_i to the state corresponding to v_j in the FSM. I(v_i, v_j) is the set of primary inputs that cause the transition from v_i to v_j and O(v_i, v_j) is the set of primary outputs produced on that transition.

1.2.2 The State Assignment Problem

Figure 1.2 shows the state transition graph (STG) for a Finite State Machine which outputs a 1 on detecting either of the two patterns 0110 or 1010. 3 bits are needed to encode the states of this FSM. There is a 1 bit output. This translates to 4 Boolean functions. Let x represent the input, y_1, y_2 and y_3 the 3 bits which encode the states of the FSM, and z the output. Consider the following state assignment: A(000), B(001), C(010), D(011), E(100), F(101), G(111). The resulting Boolean equations for the next state functions are

$$z = \bar{x}y_1y_2y_3 \tag{1.1}$$

$$y_1 = x\bar{y_1}y_3 + \bar{x}\bar{y_1}y_2 + y_1\bar{y_2}\bar{y_3} \tag{1.2}$$

$$y_2 = \bar{x}\bar{y_1}\bar{y_2}y_3 + x\bar{y_2}\bar{y_3} + x\bar{y_1}\bar{y_3} \tag{1.3}$$

$$y_3 = \bar{x}\bar{y_2}\bar{y_3} + x\bar{y_1}\bar{y_2} + \bar{x}\bar{y_1}y_3 \tag{1.4}$$

The combinational circuit required to realize these functions requires 13 gates assuming complemented inputs are available.

On the other hand the assignment A(000), B(010), C(011), D(110), E(111), F(100), G(101) yields the following next state functions

$$z = \bar{x}y_1\bar{y_2}y_3 \tag{1.5}$$

$$y_1 = \bar{x}y_1\bar{y_2} + x\bar{y_1}\bar{y_2} + xy_1y_2y_3 \tag{1.6}$$

$$y_2 = \bar{y_3} \tag{1.7}$$

$$y_3 = y_2 \tag{1.8}$$

The combinational circuit required to realize these functions requires only 5 gates.

Thus the state assignment problem involves assigning binary coded values to the states of an FSM in such a way that the area of the combinational circuit, required to realize the FSM, is minimized.

1.3 Previous Work

A lot of work has been done for the state assignment problem. MUSTANG [19], NOVA [20], and JEDI [21] are some of the recent algorithms. The algorithms targeting two-level PLA based implementations such as NOVA optimize for the number of product terms called cubes. However, the *number of literals in a factored form* for the logic is the standard estimate used by algorithms designed for multi-level implementations such as MUSTANG and JEDI.

Multilevel logic optimization systems such as SIS [22] primarily use algebraic techniques to factor and decompose the Boolean equations by identifying common sub-expressions. Hence the heuristics in both MUSTANG and JEDI try to maximize the number and size of common sub-expressions. MUSTANG finds pairs or clusters of states which, if kept minimally distant in the Boolean space representing the encoding, result in a large number of common sub-expressions. This minimizes the number of literals in the resulting combinational logic network after multilevel logic optimization.

In terms of quality, JEDI is the best known algorithm for State Assignment targeting multi-level implementations [21]. In MUSTANG, clusters of states with similar behavior are encoded closely by each processor based on some local weight information. This is a greedy heuristic and prohibits a state from being moved around once it has been encoded. The results from various processors are combined in a tree-based merge.

In contrast, in JEDI the states are moved around in the encoding space using a probabilistic hill climbing heuristic (simulated annealing). Since a majority of the time taken by JEDI is in this simulated annealing based encoding, most of the effort in implementing ProperJEDI was spent in efficiently parallelizing the encoding phase.

1.4 Why Parallel State Assignment ?

Finite State Machines(FSMs) form an important class of VLSI circuits and are especially useful for designing the control unit of various microprocessors. In general, a huge sequential circuit can be considered to be a big FSM. In such a scenario, even a circuit with modest number of latches (say 20) has more than a million states. Researchers have tried to use techniques like FSM decomposition and isolation of smaller interacting FSMs to make the time and memory requirements manageable [23]. These smaller sub-circuits are locally optimized by conventional techniques like state minimization and optimal state assignment [24].

The penalty for lack of global optimization techniques severely affects the quality of the results. In other words, the area requirement is significantly more for a piecewise locally optimized sequential machine than for a globally optimized one. To this effect our attempts to parallelize the recent state assignment algorithms assume significance. It will be amply demonstrated that a significant reduction in time and memory requirements can be achieved for the state assignment of huge FSMs, with negligible compromise in quality.

To our knowledge no work has been reported on parallel state assignment algorithms.

1.5 Parallel Annealing Strategies

A significant amount of work done in parallel simulated annealing has been for cell placement applications [25]. Using the taxonomy defined in [26], there are three major classes of parallel simulated annealing algorithms - *serial-like*, *asynchronous*, and *altered generation*.

Serial-like algorithms preserve the convergence characteristics of the sequential algorithm through the use of single move acceleration or serializable subsets. Kravitz and Rutenbar have investigated both approaches and found there is limited parallelism and it is more appropriate to shared memory architectures [27]. Speculative computation also falls into this class but has also been shown to be ineffective for parallel placement [28].

The second class of parallel simulated annealing techniques, altered generation, is distinguished from serial-like algorithms in that it does not follow the exact search space laid out by the sequential algorithm. This is usually accomplished with a processor or group of processors either exploring a restricted state space or using a restricted search on the entire state space. To ensure proper global convergence, the global state is kept up to date through periodic solution exchanges or with a shared memory architecture. This approach has been followed for cell placement with some success [29].

The final class of parallel simulated annealing algorithms are the asynchronous or "parallel moves" algorithms where each processor generates and evaluates moves independently. This differs from altered generation methods in that the state space is not restricted. In other words, each processor contains information on the entire circuit regardless of whether the global layout information is accurate in the local processor. Obviously, the cost function calculations may be incorrect due to the moves made by the other processors. There are various methods to address the effect of error, but all involve some form of periodic updates. The number of updates is directly related to the average acceptance rate of the particular annealing scheduling chosen. Several researchers have pursued this approach for cell placement [30, 2, 31, 32, 33, 34, 17].

Banerjee, Jones and Sargent [30] implemented a parallel placement algorithm using the parallel move approach on an Intel hypercube multiprocessor and proposed several partitioning strategies for the problem specific to the hypercube topology. Rose et al. [2] proposed a different parallel algorithm based on parallel moves on an experimental distributed memory multiprocessor.

Both Casotto and Sangiovanni-Vincentelli [31] and Wong and Fiebrich [32] have presented massively parallel algorithms for simulated annealing based placement on the Thinking Machines' CM-2. These methods fall in between a completely asynchronous approach and the altered generation methods. By completely distributing the circuit state, the necessity for global updates is removed, while still allowing for asynchronous parallel moves.

1.6 Summary of Contributions

The primary contributions of this thesis are:

1. Parallel algorithms for State Assignment of Finite State Machines, incrementally derived from state-of-the-art serial algorithms which achieve significant multiprocessor utilization without a sacrifice in resultant quality.

2. The ability of the algorithms to handle large circuits on multiple processors which the corresponding serial algorithms fail to handle

3. Demonstration of the utility of the ProperCAD II platform for developing portable and parallel CAD applications.

In Chapter 2, we give an overview of the MUSTANG algorithm. We go on to describe the implementation and results of the ProperSTATE parallel algorithm. Chapter 3 presents the motivation and implementation for a simulated annealing based state assignment algorithm, ProperJEDI. Chapter 4 compares and contrasts the two parallel algorithms. It presents observations on the most significant success of the implementations, and proposes features that would extend the usability of the implementations.

CHAPTER 2

ProperSTATE : A PARALLEL GREEDY ALGORITHM FOR STATE ASSIGNMENT OF FINITE STATE MACHINES

2.1 Introduction

In this chapter, we present a parallel algorithm, named ProperSTATE, for the state assignment problem of finite state machines (FSM). The parallel algorithm is based on one of the recent sequential algorithms, MUSTANG. The motivation for scalable parallelization of MUSTANG is twofold. Recent algorithms for state assignment of FSMs are particularly expensive because of the inherent compute intensive nature. Further, MUSTANG uses four different heuristics and is run four times to generate state assignments, the best of which is chosen. Hence the time requirements increase by a factor of four. Table 2.3 shows the large time requirements for just one of these heuristics for MUSTANG. The complexity and size of VLSI circuits is increasing exponentially with time. It is becoming more and more infeasible to adequately address VLSI CAD problems on a single processor due to inordinate memory requirements. In the past, a popular approach has been to partition the problem across processors, invariably suffering a loss of quality. Our algorithm has three significant contributions: (1) It is an asynchronous parallel algorithm portable across different MIMD machines. (2) The uniprocessor version of ProperSTATE is much more efficient than MUSTANG and results in an order of magnitude reduction in time requirements. (2) Time and memory requirements reduce by a factor of P (the number of processors), maintaining the quality comparable to MUSTANG.

MUSTANG heuristically minimizes an estimate of the area used by a multilevel implementation of the combinational logic. Since this area is proportional to the literal count, the final literal count after certain logic optimizations is used as a measure of quality. We ran experiments on 40 MCNC and 10 synthesized FSMs. Results obtained from 17 of them are reported for a network of Sun workstations (distributed computing), the Thinking Machines' CM-5 distributed multicomputer, and the SunSparcServer 1000 (shared memory multiprocessor).

The rest of this chapter is organized as follows: §2.2 describes the ProperSTATE algorithm in detail. Time and memory bounds for the parallel program are analyzed in §2.3. Quality improvement is discussed in §2.4. §2.5 discusses the performance of the ProperSTATE algorithm on several parallel machines. We conclude the chapter with a discussion on future work in §2.6.

2.2 ProperSTATE

We have built ProperSTATE around an existing uniprocessor implementation of a state assignment algorithm, MUSTANG. The input to MUSTANG is an FSM in the form of a state transition graph, shown in Chapter 1 Figure 1.2. MUSTANG solves the state assignment problem in two phases. In Phase 1 it evaluates a gain estimate of encoding two states at a minimal distance for all pairs of states. In Phase 2 it tries to maximize the total gain while mapping the states on the encoding hypercube. In ProperSTATE, we exploit parallelism by evaluating the gains as well as encoding the state in parallel. In addition, ProperSTATE has a third phase in which the processors exchange and enhance their encodings. In the following subsections we discuss the parallel implementation of each of the phases in detail.

2.2.1 Gain Function Computation

For computing an estimate of the gain obtained by encoding two states closely, MUSTANG provides an option between four heuristics- a fanout oriented algorithm and a fanin oriented algorithm, both targetted for two-level and multi-level implementations. All these algorithms generate a complete undirected weighted graph $G_M(V, E_M, W(E_M))$ where V is the set of states of the FSM, and $W(e_{ij})$ is the estimate of the gain that can be obtained by encoding states $v_i, v_j \in V$ close to each other.

The fanout oriented algorithm works on the output and the fanout of each state. Pairs of present states which assert similar outputs and produce similar sets of next states are given high edge weights. This has an effect of maximizing the size of common cubes in the Boolean functions corresponding to the output and the next state lines. The fanin oriented algorithm, on the other hand, works on the input and the fanin of each state. Pairs of next states which are produced by similar inputs and similar sets of present states are given high edge weights. This has an effect of maximizing the number of common cubes in the boolean functions corresponding to the next state lines. Without loss of generality we consider the fanin oriented algorithm.

2.2.1.1 Sequential algorithm

Let the number of inputs be N_i, the number of outputs N_o and the number of states N_s. The fanin algorithm proceeds as follows:

1. For each node (state) in the state transition graph, look at all the edges fanning in. For every node, construct $2N_i$ input factors, two for each input (corresponding to the input being a 1 or a 0). Each input factor is a value which indicates the number of times that input asserts the state being considered. If two states are asserted by similar inputs then their input factors for those inputs will be high.

```
GAIN_COMPUTATION()
if (myprocnum == 0)
  then Read the state transition graph.
       Compute P partitions of all nodes (states), one for each processor.
       for (i = 0; i < P; i++)
       do Send the entire graph and the ith partition information
            to processor i.
for each processor
do Receive the state transition graph and the local partition information.
   Initialize local data structures and the local partition node list.
   for all nodes in local partition
   do for all the edges fanning into this node
       do Update the Ni input factors for this node.
          Update the Ns present state factors for this node.
   Broadcast input and present state factors to all other processors.
   while information keeps coming in
         do for all nodes i in local partition
             do for all nodes j in the partition of the sender processor
                 do Compute weight W(i,j) between i and j using there
                      output and next state factors.
             delete the incoming information.
   Sort the local weight list, W.
```

Figure 2.1 An outline of Phase 1 of ProperSTATE using fanin oriented algorithm

2. For each node (state) in the state transition graph, look at all the edges fanning in. For every node, construct N_s present state factors, one for each state. Each present state factor is a value which indicates the number of times that present state leads to the state being considered.

3. For each pair of states in the FSM, generate a gain estimate of encoding them closely. This is done by multiplying their corresponding input and present state factors and summing them up. If two next states are asserted by similar inputs and produced by similar present states then they could potentially share a large number of common cubes. Their factors when multiplied give a large gain value.

2.2.1.2 Parallel Implementation

The state transition graph is replicated on all processors. ProperSTATE divides the set of states of the FSM among its processors for computation of the gain function. The work done in Steps 1 and 2 is proportional to the number of edges, whereas that in Step 3 is proportional to the number of nodes. Since Step 3 is computationally more intensive, a load balance is achieved by cyclically partitioning the nodes. All edges fanning out of a node are given to the owner processor of that node. This enables each processor to compute the $2N_i$ output factors and N_s next state factors for all its owned nodes independent of other processors (Steps 1 and 2). In practice, the cyclic partition of nodes leads to a more even distribution of edges than a blocked distribution. For computing Step 3, each processor needs the output and next state factors of all the nodes. This requires a broadcast of the computed factors to every processor. Thus each processor computes N_s edge weights (gain estimates) for each of its local nodes. Each weight in G_M is computed by the two processors owning the corresponding states of the STG. G_M which requires $O(N_s^2)$ space for its edge weights is thus divided among the processors and requires N_s^2/P space per processor.

A look at Figure 2.1 shows that there is no synchronization in Phase 1. The asynchronous nature of the algorithm enables overlap of computation and communication. Also to avoid storage of all output and next state factors received from all the processors, which require $O(N_s^2)$ storage, they are discarded immediately after use. At no time is the information from two different processors stored simultaneously. This is essential for achieving memory scalability.[1] An analysis of memory requirements is given in §2.3.

2.2.2 Encoding Phase

In Phase 1 each processor generates N_s edge weights for every node it owns. These weights are then sorted together in decreasing order to generate the local weight list. At this stage the processor is ready to go ahead with the encoding of the states. This is essentially equivalent to mapping the set of state on a Boolean hypercube. The dimension of the hypercube is equal to N_b, the number of bits for encoding the states of the FSM. First we will discuss briefly the encoding algorithm used by MUSTANG and then describe its parallel implementation in detail.

2.2.2.1 Sequential algorithm

MUSTANG tries to heuristically minimize the function

$$\sum_{i=1}^{N_s} \sum_{j=i+1}^{N_s} we(s_i, s_j) * dist(enc(s_i), enc(s_j))$$

. The function $we()$ returns the edge weight between vertices s_i and s_j in G_M. $dist()$ returns the Hamming distance between the two binary codes assigned to s_i and s_j. If a pair of states has a high edge weight then the algorithm attempts to encode them as close as possible to minimize this

[1] All $O(N_s^2)$ data structures required by the intermediate computations in the sequential algorithm have been handled using a partitioned approach. Entire data structures are not required simultaneously for any computation. Hence, each processor works on the part of the data structure it requires and then discards it.

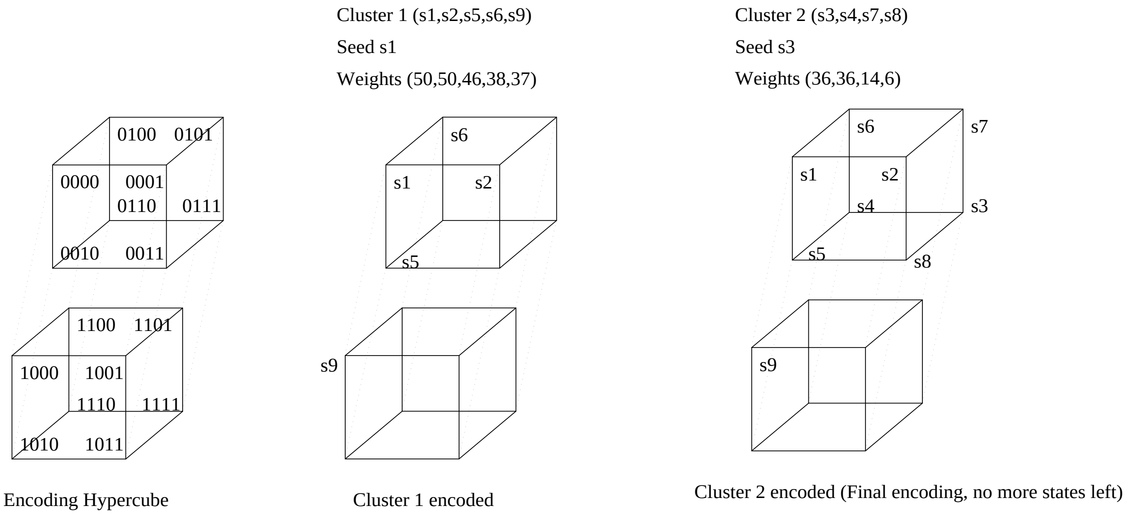

Figure 2.2 An example showing the Encoding Phase of Mustang

sum. It looks for clusters or pairs of states to encode together. Clusters of nodes with cardinality no greater than $N_b + 1$ and consisting of edges of maximum total weight are identified. One of them is picked as a *seed* and embedded in the encoding hypercube. The rest of the states in the cluster are embedded at a Hamming distance of 1 from the seed.

Figure 2.2 shows the encoding phase of MUSTANG on a 9 state FSM. (s1,s2) has the highest weight in the sorted weight list. s1 is picked as the seed and the states with maximum weights with s1 are chosen to form a cluster of size $5(N_b = 4)$. The seed is embedded first and remaining nodes are embedded on distance-1 vertices. After embedding this cluster the weight list is scanned to find the next potential seed. (s3,s4) has the next highest edge weight and both are still unencoded. Hence, s3 is picked as the seed and a cluster is formed in a similar manner. This cluster is of size 4 since all other states have been encoded. The encoding space is searched for the first vertex which has 3 distance-1 neighbors unassigned. 0111 satisfies the constraint and is chosen to embed the seed s3. Again the remaining vertices are embedded on the distance-1 neighbors. This is the final encoding as no other state remains unencoded.

2.2.2.2 Parallel Implementation

The encoding hypercube is partitioned among the different processors. Each processor depending on its identity, works on a different partition. The sorted list of gain values is scanned to identify clusters. The seed of these clusters is embedded in the processor's own partition. The remaining states in the cluster are embedded as explained. If there are 8 processors and a 7-dimension hypercube (7 bit encoding for states) then each processor works on a 4-dimension hypercube ($2^7/8 = 2^4$). But if each processor is restricted to a smaller encoding space then it cannot encode large clusters.

In this case, each processor would be encoding clusters of maximum size 5 whereas the sequential algorithm encodes clusters of size 8. This would lead to a significant degradation in quality. Hence, each processor is allowed to claim vertices from another processor's partition. These claims are resolved in the merge phase along with other contentions as explained in §2.2.3. Due to partitioning of the encoding space, each node of the hypercube loses $log_2 P$ distance 1 neighbors. After embedding maximum possible states at a distance of 1 from the seed on the local partition if some states of the cluster still remain unencoded then the processor claims non-local distance 1 neighbors.

A state may be encoded by two or more processors. This happens because all processors work on their local gain lists which contain the gain estimates of its local nodes with all the nodes of the state transition graph. A node may end up being included in different clusters on different processors. This is another correction to be performed in the merge phase.

2.2.3 Merge Phase

After each processor embeds its clusters in its local partition of the hypercube they merge their information pairwise using a standard tree based merge approach. The following tasks are done at each merge step:

- Exchange the best encoding information
- Resolve claims on the same encoding space
- Correct codes for state with multiple assignments

The two processors send their encoding information to each other. In general there would be clusters of states, which a processor could not code in the encoding phase due to lack of information about their gain estimates. So each processor selectively enhances its encoding using the incoming information. The resolution of claims is complicated by the fact that one of the contenders could be a seed of a cluster. The gain estimate realized in an encoding is preserved and communicated along with the coding information. In case of a conflict the choice with better gain is picked. If a seed loses a claim then its cluster is also removed from the encoding hypercube. A similar technique is applied for picking the better choice for states with multiple assignments. Due to the dynamically changing encodings and states being removed, the associated gain values realized have to be consistently updated. A detailed description of the utility and update of the gain values is give in §3.3.

The heuristic used for correction of codes actually results in the two processors resolving the conflicts differently. The exact detail is slightly involved. As an example if two nodes encoded with the same gain claim the same encoding, then the processor resolves the conflict in favor of the one which is local to the processor at the time of the merge. Hence, all the processors are allowed to generate complete encodings for the FSM through the $log_2 P$ merge phases. Some states may

```
ENCODING_PHASE()
for each processor
do max_cluster_size = Nb + 1 /* no. of bits required to represent a code + 1 */
    repeat
            Look for the first node pair in the local weight list
            with both nodes unassigned.
            Choose the local node of the pair as the seed for
            creating a cluster.
            Go through the weight list and form a cluster of size
            equal to (max_cluster_size) by picking up nodes with
            maximum weights with the seed.
            embedded = FALSE
            while (cluster_size > N_b/2 + log_2 P) and (embedded == FALSE)
                do Search the local encoding space for a vertex which has
                   (cluster_size - 1) distance - 1 neighbors unassigned.
                   /* Non - local neighbors are assumed to be unassigned */
                   if such a vertex is found
                      then Embed the seed on this vertex.
                           Embed the rest of the cluster on the
                           distance_1 neighbors.
                           Store the gain estimate between the seed and
                           each node of the cluster with the corresponding
                           encoding vertices of the nodes.
                      else Reduce cluster_size and max_cluster_size by 1.
                           Remove the last node from the cluster.
                           /* Last node had the least gain with the seed */
      until clusters of size greater than (N_b/2 + log_2 P) are being encoded.
```

Figure 2.3 An outline of Phase 2 of ProperSTATE using fanin oriented algorithm

```
MERGE_PHASE()
for  each processor
do for i = 0 to i < log2P in steps of 1
    do Send the local encoding information and the associated gain
        estimates to the processor whose number differs in the ith bit.
        Look at the seeds of the incoming encoding.
        Try to enhance the local encoding by encoding the seeds which
        result in a net gain. /* An already encoded node or
        a cluster may be undone in this process */
        Try to encode the incoming clusters corresponding to the
        seeds encoded.
        /* Each node in the cluster is picked one by one and
        embedded if there is a net gain. Some already encoded
        nodes may be undone */
        Discard the incoming information.
    Go through the local weight list to find all unassigned nodes
    and record the nodes which have maximum weight (gain) with
    each of these nodes.
    Encode each unassigned node closest to its recorded node.
```

Figure 2.4 An outline of Phase 3 of ProperSTATE using fanin oriented algorithm

remain unencoded after the merge phases. They are coded by picking up the maximum possible gain in the local gain list of that processor for that state and trying to realize that gain. The final encodings obtained by each processor are in general, different from each other. So P outputs are generated. Total gain achieved by each final output is computed by summing up the individual gains associated with each encoded state in that output. The output with the maximum gain value is picked as the final solution.

Figure 2.5 shows the encoding and merge phases on the FSM of Figure 2.2 for a 4 processor run. For simplicity the first cluster to be encoded by each processor is made to include all local states. Since each cluster requires a local unencoded state as a seed, no more clusters are generated. The first node in the cluster is the seed and the weights refer to the gain estimates of the corresponding nodes with the seed. Each processor embeds the seed first (e.g. P0 encodes s1 as 0000, P1 encodes s2 as 0111). The remaining nodes are embedded on local distance-1 vertices. When these vertices are exhausted, non-local distance-1 neighbors are claimed (e.g. P0 encodes s2 as 0001, s5 as 0010 on local vertices but s6 as 0100, s9 as 1000 on non-local vertices).

There are $log_2 4 = 2$ merge phases. In the first merge P0 exchanges information with P1 (and P2 with P3). On P0 the encoding of P1's seed s2 is rejected since it is already encoded with the same gain estimate. Hence, none of the encodings proposed by P1 is accepted by P0. P0's claim for s6 is settled without any conflict. The merge step on P1 proceeds similarly. The local cluster

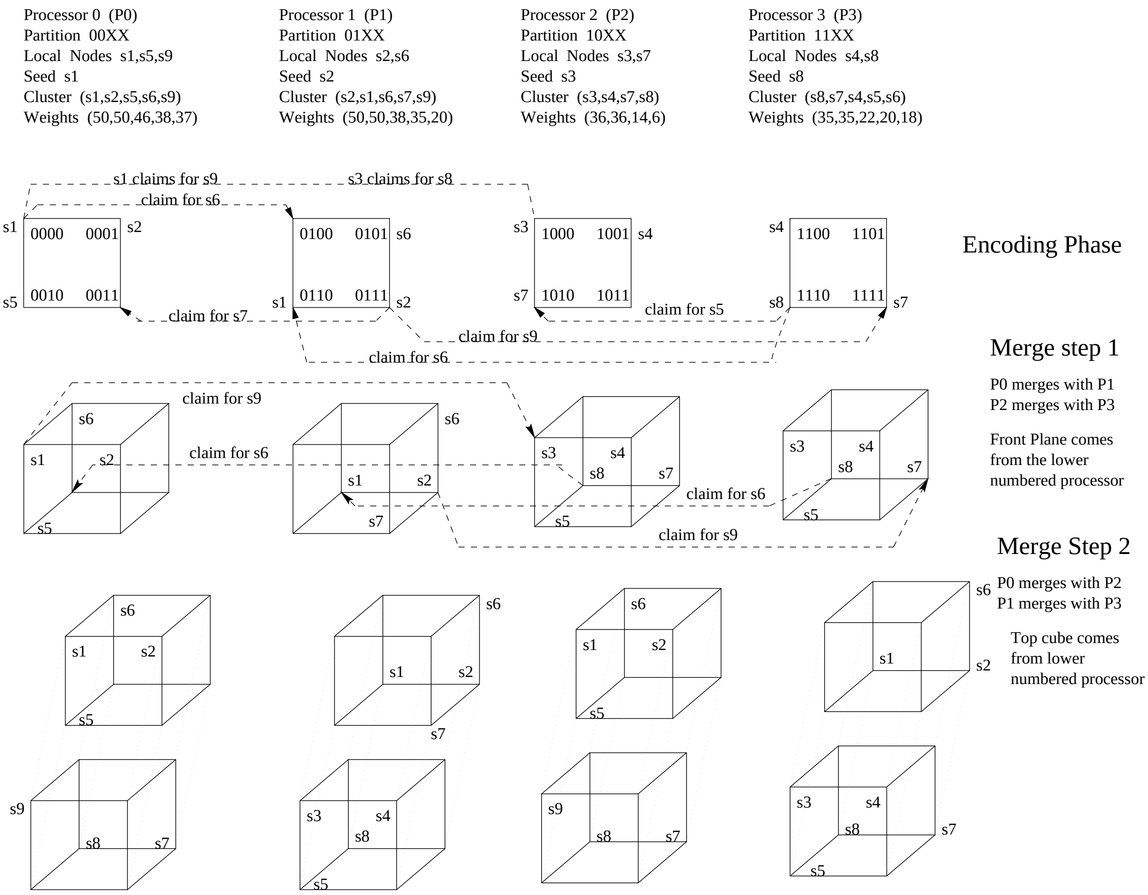

Figure 2.5 An example of Encoding and Merge Phases of ProperSTATE on 4 processors

of P2 is weak as seen from the gain estimates. P3 proposes an alternate encoding for s7 with a higher gain. This is accepted by P2. Hence, vertex 1010 is released and later on taken by s5. P3, however, loses its claim for s4. Same conflicts are resolved on P3 similarly; however, there is a difference. The proposal for encoding s7 by P2 faces two conflicts on P3, one with s5 which claims the same vertex and the other with already encoded local s7. The unsettled claim for s6 made by P3 is recorded by P2 while accepting s8. P2 also drops its claim for s8 on P0 after encoding s8.

In the second merge step, P0 exchanges information with P2 and wins its claim for s9. Since the losing node is the seed, its cluster (which has been reduced to s3,s4) is also undone. Both processors end up in an identical encoding after this merge. An initial sub-optimal decision for encoding a weak cluster by P2 has been rectified in this process. The merge of P1 and P3 has two conflicts. P1 claims vertex 1111 for s9 and loses the claim because of lower gain on both the processors. s7 is encoded by both the processors with the same gain estimate. Thus both processors stick to their own encoding for s7.

After the merge each processor encodes any remaining unencoded state and outputs a complete solution with the estimated total gain.

2.3 Analysis

In this section we bound the maximum number of encoding conflicts that can occur in the algorithm and its effect on the quality of the solution. In the end we derive the time and space bound for ProperSTATE and show that both time and memory requirements reduce by a factor of P for a P-processor run asymptotically as the number of states $|V| \rightarrow \infty$.

Let $|V|$ be the number of nodes or states in the state transition graph and E the number of edges. Let n_o be the number of outputs and n_i the number of inputs. P is the number of processors. For simplicity the communication costs are assumed to be independent of the network load. A distributed memory MIMD machine is assumed with a communication cost model of $\alpha + \beta * (MessageSize)$ where α is the fixed cost for initiating a message and β is the cost incurred per byte of the message [35].

2.3.1 Encoding Conflicts

The size of the local encoding space on a processor is $\frac{|V|}{P}$. Each encoded cluster is at least $\frac{log_2|V|}{2} + log_2 P$ in size (Figure 2.3 line 9). At most $log_2 P$ distance-1 neighbors of a code can be non-local (This is true because of the way the encoding space is partitioned among processors. See §2.3.2 for more detail). So the seed of each cluster can claim atmost $log_2 P$ non-local vertices. Hence at least $\frac{log_2|V|}{2}$ vertices are local. Hence the maximum number of clusters encoded by a processor , $NC_1 \leq \frac{2|V|}{Plog_2|V|}$ (Ratio of local encoding space to the minimum local size of each cluster).

Multiplying this with the number of processors P, we get the maximum number of clusters that can be encoded in the system $NC_{total} = NC_1 * P \leq \frac{2|V|}{log_2|V|}$. Each cluster claims at most $log_2 P$ non-local vertices. So there can be at most $log_2 P$ conflicts for every encoded cluster. Hence the maximum number of conflicts in the system $= NC_{total} * log_2 P \leq \frac{2|V|log_2 P}{log_2|V|}$.

Further, let k be the number of weak clusters that are removed from the system. The number of conflicts in the system gets reduced by $klog_2 P$. The number of conflicts remaining in the system $= \frac{2|V|log_2 P}{log_2|V|} - klog_2 P = (\frac{2|V|}{log_2|V|} - k)log_2 P$. Assuming that these conflicts affect each of the remaining $\frac{2|V|}{log_2|V|} - k$ clusters equally, the average degradation per cluster is given by $\frac{(\frac{2|V|}{log_2|V|} - k)log_2 P}{\frac{2|V|}{log_2|V|} - k} = log_2 P$.

Hence, the average cluster size reduces by $log_2 P$ in the worst case due to the encoding conflicts in the system.

2.3.2 Time Bound for Conflict Resolution

The parallelism overhead of ProperSTATE can be attributed to two main factors:

1. Communication.

2. Conflict resolution during the merge phase.

While the communication times are analyzed in the next section, let us take a closer look at the time requirements for resolving the conflicts.

Let Nb denote the number of bits used for representing a code (log_2(size of the encoding hypercube)). Let us assume a code to be represented by the decimal number that its bit pattern represents. The encoding space is partitioned among the processors in such a way that processor P_i owns codes $(i-1) * \frac{2^{Nb}}{P}$ to $i * \frac{2^{Nb}}{P} - 1$. In other words the $log_2 P$ most significant bits of a code identify its owner processor uniquely. So if two codes $C_m \in P_a$ and $C_n \in P_b$ (with binary representations m and n) are distance-1 neighbors then either $P_a = P_b$ or a and b differ in the same bit as m and n counting from the most significant bits. Thus two processors differing only in bit i, merge information with each other at step i.

Consider two processors P_i and P_j exchanging information in the first merge step. Before the first merge step each processor can have a maximum of $NC_1 = \frac{2|V|}{Plog_2|V|}$ clusters as shown in §2.3.1. The seed for each cluster on $P_i(P_j)$ claims at most 1 distance 1-neighbor on $P_j(P_i)$. So a maximum of $2NC_1$ conflicts have to be resolved in the first merge step between any 2 processors. After every merge step the size of the local encoding doubles because of combining with an equal sized encoding space from the merging processor. The number of encoded clusters also doubles correspondingly. By a similar argument as above, the maximum number of conflicts resolved in step i of the merge phase is $2^{(i-1)} * 2NC_1$ between each pair of processors. Notice that all the conflicts between different pairs of processors are resolved in parallel at each merge step. So a processor has to resolve $(2NC_1 + 2 * 2NC_1 + ... + 2^{(log_2 P - 1)} * 2NC_1)$ since there are $log_2 P$ steps in the merge phase. Summing up the geometric series we get a bound of $2(P-1)NC_1 \leq \frac{4|V|}{log_2|V|}$. If T_{cr} is the time taken to resolve each conflict then the time overhead for corrections is bounded by $\frac{4|V|}{log_2|V|}T_{cr}$. This is a strong result as it shows independence of the computation part of the parallelism overhead from the number of processors in the system.

T_{cr} for our algorithm is $O(1)$. A more complicated conflict resolution strategy would give better quality but increase the time overhead.

2.3.3 Runtime Analysis

Weight Generation Phase (Phase 1 in Figure 2.1):
Reading and initializing the STG on Processor 9 takes $O(|V|+|E|(n_o+n_i))$ (line 2 in Figure 2.1). This is followed by the partitioning of the STG. States are cyclically assigned to P partitions taking $O(|V|)$ time (line 3). The STG and the relevant partition information is sent to each processor taking $(P-1)[\alpha+\beta(\frac{|V|}{P}+|E|(n_o+n_i))]$ (lines 4-6). All processors set up their data structures after receiving the STG. This includes initializing the STG and the local state partition list, which takes $O(|V|+|E|(n_o+n_i))$ (line 9). Each processor owns $\frac{|V|}{P}$ states. For each state all edges fanning into it are looked at. The input and present state factors are updated by looking at n_i inputs and 1 present state corresponding to that edge. If the maximum in degree of a vertex is k then this step takes $O(\frac{|V|}{P}kn_i)$ (lines 12-17). The $2n_i$ input factors and the $|V|$ present state factors for each local node are broadcast to all processors. Hence, line 18 takes $O((P-1)[\alpha+\beta\frac{|V|}{P}(2n_i+|V|)])$ time. In lines 19-27, for each local node, $|V|$ weights are computed. Each weight is obtained by multiplying the corresponding input and present state factors taking $\frac{|V|}{P}|V|(|V|+2n_i) = O(\frac{|V|^3}{P}+2\frac{|V|^2}{P}n_i)$. Sorting the weight list of length $\frac{|V|^2}{P}$ takes $\frac{|V|^2}{P}log_2\frac{|V|^2}{P}$ on line 28 (assuming a standard $O(nlog_2n)$ sorting technique). Thus for a large $|V|$ time required by phase 1 $\approx O(\frac{|V|^3}{P})$.

Encoding Phase (Phase 2 in Figure 2.3):
Choosing the seed and creating the cluster involves going through the weight list once, taking $O(\frac{|V|^2}{P})$ (lines 3-7 in Figure 2.3). Size of local encoding space on each processor is $O(\frac{|V|}{P})$. Searching this space to find out a vertex with $N_b = ceil(log_2|V|)$ distance-1 unassigned neighbors takes $O(\frac{|V|}{P}log_2|V|)$ (lines 10-11). Embedding the cluster on these vertices (lines 13-18) takes $O(log_2|V|)$.

Let us calculate an upper limit for the number of executions of the while loop (line 9). The maximum possible successful cluster encoding attempts is $O(\frac{2|V|}{Plog_2|V|}))$ as shown in §2.3.1. The maximum possible *unsuccessful* encoding attempts is $N_b = O(log_2|V|)$ since maximum cluster size is decreased by 1(line 19) at each unsuccessful encoding attempt. Hence, the total possible cluster encoding attempts is $O(\frac{2|V|}{Plog_2|V|}+log_2|V|))$. The time for encoding all the clusters is $\frac{|V|}{P}log_2(|V|) *$ $(\frac{2|V|}{Plog_2|V|}+log_2|V|) = O(\frac{|V|^2}{P^2}$ Hence the total time required for Phase 2 is $O(\frac{|V|^2}{P}+\frac{|V|}{P}+\frac{|V|^2}{P^2})$ which is $O(\frac{|V|^2}{P})$.

Merge Phase (Phase 3 in Figure 2.4)

For each encoded node the following information is sent: the node number, its associated seed number, corresponding vertex on which it is embedded in the encoding hypercube and the associated gain estimate. Thus communication at each merge step (line 2 in Figure 2.4) requires $O(\alpha + 4\beta|V|)$. Enhancing the local encoding requires going through the incoming information once and embedding them after conflict resolution.

Since conflict resolution is O(1), as explained in §2.3.2, the time required at each merge step is $O(4|V|)$. Since there are $log_2 P$ merge steps, the merge phase takes $O(|V| log_2 P)$ time, for large $|V|$.

At the end of the merge phase any remaining nodes are encoded. This involves going through the weight list once which is $O(\frac{|V|^2}{P})$ (line 12). There may be $O(|V|)$ unencoded nodes. The entire embedding of the hypercube of size $O(|V|)$ is searched to find a suitable vertex for each such node. Thus the total time taken is $O(|V|^2)$ (line 13).

2.3.4 Memory Requirement

Weight Generation Phase (Phase 1 in Figure 2.1):

Reading and initializing the STG on Processor 0 requires $O(|V| + |E|(n_o + n_i))$ memory (line 2 in Figure 2.1). Partitioning the STG takes $O(|E|)$ space for the P partition lists (line 3). The message size for sending the STG and the relevant partition information to each processor takes $O(\frac{|V|}{P} + |E|(n_o + n_i))$ space (lines 4-6). All processors set up their data structures after receiving the STG. This takes $O(|V| + |E|(n_o + n_i))$ space on those processors (lines 8-9). The $2n_i$ input factors and $|V|$ present state factors for each local node are computed taking $O(\frac{|V|}{P}(2n_i + |V|))$ space (lines 12-16). These factors are broadcast to all processors. Hence, each processor receives (P-1) messages of size $O(\frac{|V|}{P}(2n_i + |V|))$ (line 18). Each message is processed and then discarded. Hence, the space overhead remains $O(\frac{|V|}{P}(2n_i + |V|))$ and does not blow up to $O(|V|(2n_i + |V|))$. The local weight list requires $O(\frac{|V|^2}{P})$ space (lines 20-24). Hence, for large $|V|$ the space required by phase 1 $\approx O(\frac{|V|^2}{P})$.

Encoding Phase (Phase 2 in Figure 2.3):

The entire encoding space is created on each processor since each processor computes the state assignment for all nodes. This requires $O(2^{ceil(log_2|V|)}) \approx O(V)$ space.

Merge Phase (Phase 3 in Figure 2.4):

For each encoded node the node number, its associated seed number, corresponding vertex on which it is embedded in the encoding hypercube and the associated gain estimate are exchanged by the merging processors. Hence, the communication at each merge step requires $O(4|V|)$ space.

These messages are discarded before the next merge. Hence, for large $|V|$ the merge phase takes $O(|V|)$ space.

2.4 Quality Control

The solution to the state assignment problem is basically an exploration of a search space (Boolean hypercube) to allocate encodings to the states of an FSM. Effectively we have utilized the parallelism to search more during the merge phase. Instead of keeping other processors idle after they communicate their information, they are programmed to explore other possible encodings. The sequential algorithm greedily picks up clusters. After encoding them it does not have a backtrack mechanism to undo any encoding. Hence, a suboptimal decision made during this phase cannot be changed. On the other hand, the $log_2 P$ merge phases of the parallel algorithm involve a correction step. This step implicitly provides a backtrack in favor of a better solution by undoing an encoding with lesser gain for a more prolific encoding. As a side effect of the correction step the clusters tend to reduce in size with each merge step. This has an effect of degrading the quality.

In order to pick up the best output of the P outputs produced, a gain estimate is kept with each state encoding. This estimate is consistently updated during the merge phase. Each processor calculates the total gain associated with its output by summing up the gains for each individual encoding. The output with the maximum gain is picked as the final solution. This solution may not be the best owing to the following reasons:

1. A fanin oriented algorithm maximizes the number of common cubes in the Boolean functions representing the combinational logic. Hence, the gain values estimate this number. However, implicitly the size of these common cubes could be maximized in a solution. This solution may turn out to be better than the one with highest gain. The same logic applies to the fanout oriented algorithm which maximizes the size of the common cubes.

2. MUSTANG optimizes for common cubes but ignores other types of optimizations such as common kernel extraction and Boolean factoring. Some solutions may yield better results after these optimizations, available in the optimization tool.

3. It is very complicated and computationally intensive to calculate the exact gains realized as the encoding builds up. The interaction between two or more clusters and states which are encoded independent of each other are not modeled in our gain estimates.

Table 2.1 shows the gain estimates for 2-processor runs on some MCNC benchmark FSMs. In practice, with increasing number of processors we get better best solutions but the probability of picking a sub-optimal solution also increases. The positive effect of the former offsets the loss due to the latter. Hence we see a net improvement in quality as we move to higher number of processors

Table 2.1 Effectiveness of Gain Estimates on 2 Processors for the Fanin Oriented Heuristic Gain vs. Literal count

FSM	Proc 0 # lits	Proc 0 Gain	Proc 1 # lits	Proc 1 Gain
scf	1196	642	1180	785
bbara_bbtas	1130	49005	1216	43741
s298	7760	22512	8162	20074
fetchXlog	9252	4605	9532	4534
dvramXmarkl	9892	8685	10127	8558

upto 8 processors. With an increasing number of processors, the total number of encoded clusters increases. When $P > log_2|V|+1$ the number of encoded clusters becomes $|V|$ in the worst case ($\frac{|V|}{P}$ clusters per processor). Since each cluster is of size $O(log_2|V|+1)$, the total number of encodings to be undone is $|V|log_2|V| - |V| = (|V|-1)log_2|V|$. This breaks up lots of clusters and reduces their size a lot, offsetting the advantage of the correction. Therefore we see a degradation in quality for 16 and 32 processor runs.

$P < log_2|V|+1$ is a good criterion for selecting the number of processors for a given problem size. Our experiments show that although this does not guarantee the best solution, it maximizes the chance for obtaining one. We selected a benchmark set with number of states in the range of 120-500. As expected the best results are obtained with 8 processors. We believe that for a large problem size where $|V| >> P$ the quality of the solution will improve with increasing P as long as $P < log_2|V|+1$.

2.5 Experimental Results

The ProperSTATE parallel algorithm described in the previous sections has been implemented as part of the ProperCAD project [13]. In this section, we will describe the results obtained by applying ProperSTATE on various circuits in the MCNC benchmark suite. The output obtained from ProperSTATE is passed through multilevel logic optimization scripts of SIS, a logic synthesis tool from Berkeley. The quality of the solution is measured as the number of literals in a factored form of the logic after optimization. Less literals imply less area for implementing the combinational logic and hence better quality. Literal count after optimization has been reported for 17 benchmarks. For each benchmark, all the four weight generation heuristics were run and the best quality was picked. The quality of the solution improves with increasing number of processors upto

[2] The time taken by MUSTANG (and ProperSTATE) is almost the same for the four different weight generation heuristics. These tables show the sum of the individual times for each of them.

Table 2.2 Quality Comparison between various runs of ProperSTATE and MUSTANG(Dashed entries correspond to small problem sizes relative to the number of processors)
Quality in Literal counts

FSM BENCHMARK	MUSTANG	ProperSTATE					
		1 PE	2 PE	4 PE	8 PE	16 PE	32 PE
dk16(27)	286	266	294	279	295	-	-
dvram(35)	234	198	211	192	209	-	-
fetch(26)	152	139	144	146	155	-	-
keyb (19)	196	210	249	250	262	-	-
log (17)	131	111	110	112	123	-	-
planet(48)	628	609	632	620	645	-	-
ram_test(72)	821	579	595	704	732	-	-
s1488(48)	669	711	703	739	740	-	-
s1494(48)	694	699	672	731	718	-	-
s510 (47)	522	428	390	387	434	-	-
sand (32)	709	697	682	772	762	-	-
sync (52)	502	340	342	427	446	-	-
bbara_bbtas(128)	1110	1064	1039	1049	1110	1164	1304
scf(121)	1046	918	1002	1016	1144	1221	1180
s298 (218)	8219	8031	8016	8739	7642	8324	8576
fetchXlog(442)	9240	9139	9043	9265	9450	9960	9514
dvramXmarkl(423)	10248	10147	9980	9973	9814	9877	10721
Scaled count	100	91.9	93.8	97.1	100.1	100.3	100.3

Table 2.3 ProperSTATE on a CM-5 (distributed memory multicomputer)[2]
Time in seconds(Speedup)

FSM BNCHMARK	MUS-TANG	ProperSTATE				
		1 PE	4 PE	8 PE	16 PE	32 PE
bbara_bbtas	324.0	12.48(1.0)	3.76(3.32)	2.16(5.8)	1.52(8.21)	1.24(10.06)
scf	149.6	28.4(1.0)	7.76(3.66)	3.76(7.55)	2.2(12.9)	1.68(16.9)
s298	3416.0	79.28(1.0)	20.2(3.92)	11.24(7.05)	6.72(11.8)	4.56(17.38)
fetchXlog	54032.8	588.6(1.0)	140.64(4.18)	71.56(8.22)	38.76(15.18)	23.7(24.86)
dvramXmark1	51886.0	1983.7(1.0)	481.28(4.12)	169.4(11.7)	64.64(30.68)	32.48(61.1)
fetchXkeyb	timeout	5747.8(1.0)	1381.8(4.16)	588.8(9.76)	291.84(19.7)	163.4(35.2)
planetXfetch	memout	memout	8148.15	2398.16	1162.32	554.48

Table 2.4 ProperSTATE on a SparcServer1000 (8 processor shared memory multiprocessor) Time in seconds(Speedups)

FSM BENCHMARK	MUSTANG	ProperSTATE			
		1 PE	2 PE	4 PE	8 PE
bbara_bbtas	83.6	2.0(1.0)	1.6(1.25)	0.8(2.5)	0.4(5.0)
scf	67.6	8.8(1.0)	6.0(1.5)	2.8(3.14)	1.2(7.33)
s298	842.4	14.0(1.0)	11.6(1.2)	5.6(2.5)	3.6(3.9)
fetchXlog	11861.6	99.2(1.0)	70.4(1.4)	39.2(2.53)	21.6(4.6)
dvramXmark1	11434.4	574.0(1.0)	335.6(1.71)	155.6(3.69)	55.6(10.32)
fetchXkeyb	20911.2	340.8(1.0)	202.8(1.68)	108.4(3.14)	44.8(7.6)
planetXfetch	Timeout	8639.2(1.0)	5954.4(1.74)	2596.0(3.33)	1010.4(8.55)

8 processors as seen in Table 2.2. The literal count for each heuristic is reported separately for MUSTANG and one, two and four processor runs of ProperSTATE. Interested reader should look at Tables 2.7, 2.8, 2.9 and 2.10 reported at the end of the chapter. Note that for mulltiprocessor runs each processor produces a separate output along with a gain estimate. The output with the best gain estimate is picked for that particular run. In general, some other output may give the least literal count after optimization through SIS. The correlation of gain estimate to the final quality has been studied for a few benchmarks in Table 2.1.

The speedups are reported for a SparcServer1000 shared memory multiprocessor, a CM-5 distributed memory multicomputer and a network of Sun workstations. The memory required per processor for the data structures of the algorithm is reported in Table 2.6 for the CM-5. For large benchmarks the memory requirement reduces by a factor of 2 for 4 processors and 3 for 16 processors. Thus *planetXfetch* which was too large to fit on 2 processors, was successfully run on 4 processors on the CM-5 as seen in Table 2.3.

Due to a lack of large FSMs in the benchmark suite, they were synthesized through the cross products of the FSMs in the benchmark. A cross product generator [3] was written which takes as input two FSMs M_1 and M_2 and outputs another FSM $M_3 = M_1 X M_2$. SIS was unable to optimize very large FSM benchmarks like *planetXfetch* which were encoded by ProperSTATE. A partitioned parallel implementation of SIS is in progress currently in the ProperCAD group. Using this memory scalable implementation we hope to optimize these huge benchmarks.

[3] Given two FSMs M_l and M_2 their cross product $M_3(M_1 X M_2)$ is defined as a machine which has the ability to execute M_l and M_2 in parallel. If $M_1 = (S_1, X_1, Z_1)$ and $M_2 = (S_2, X_2, Z_2)$ then $M_3 = (S_3, X_3, Z_3)$ where
$S_3 = ((s_1, s_2)|s_1 \in S_1 \text{ and } s_2 \in S_2)$,
$X_3 = ((x_1, x_2)|x_1 \in X_1 \text{ and } s_2 \in X_2)$,
$Z_3 = ((Z_1, Z_2)|Z_1 \in Z_1 \text{ and } Z_2 \in Z_2)$.

Table 2.5 ProperSTATE on a Network of Sun Workstations
Time in seconds(Speedups)

FSM BENCHMARK	MUSTANG	ProperSTATE		
		1 PE	2 PE	4 PE
bbara_bbtas	146.8	3.2(1.0)	2.4(1.33)	1.6(2.0)
scf	122.0	10.8(1.0)	7.6(1.4)	5.6(1.9)
s298	1408.4	17.2(1.0)	12.4(1.4)	10.4(1.7)
fetchXlog	21221.6	119.2(1.0)	73.6(1.6)	55.6(2.1)
dvramXmark1	20262.4	624.8(1.0)	425.2(1.5)	206.8(3.0)
fetchXkeyb	29631.2	390.0(1.0)	274.4(1.4)	175.2(2.2)
planetXfetch	Timeout	9207.2(1.0)	6315.2(1.46)	3385.2(2.72)

The results are reported only for 7 benchmarks. For the rest of the benchmarks the time taken is less than 1 second and hence their data is not interesting from the point of view of timing. However, it is interesting to study their quality degradation due to the parallelism of the application.

As seen from Tables 2.3, 2.4 and 2.5 the uniprocessor implementation of ProperSTATE is much more efficient than MUSTANG. We attribute the superlinear speedups as seen in Table 2.3 to the cache effects. Since the memory requirements decrease with increasing number of processors, the cache hit ratio increases. The sorting time also speeds up by a factor greater than 1 (reduces from $|V|^2 log_2 |V|^2$ to $\frac{|V|^2}{P} log_2 \frac{|V|^2}{P}$). Further the searching of the boolean hypercube for cluster formation may result in superlinear effects.

As explained in §3.3 larger benchmarks give better quality with increasing processors. Our benchmark set gives best quality results with 8 processors.

2.6 Summary

In this chapter, we have demonstrated that it is possible to design and implement a portable parallel state assignment tool that is cleanly and efficiently interfaced with MUSTANG using the ProperCAD II environment. We have shown that ProperSTATE gives very good speedups on both, shared memory machines and message passing machines in this manner. It also improves the quality by implicitly correcting some sub-optimal decisions made earlier. The state assignment of large benchmarks which do not fit on single processors can be done using ProperSTATE because of its memory scalability.

The implementation is very modular and we expect that a new algorithm can replace the sequential MUSTANG code to give a better parallel algorithm if the sequential algorithm has similar phases (weight generation and encoding). Any improvements to the uniprocessor algorithm

Table 2.6 Memory requirement for ProperSTATE data structures on CM-5
(Maximum memory required is chosen for multiprocessor runs)
Memory in Kilobytes (Memory Scaledowns)

FSM	ProperSTATE					
BENCHMARK	1 PE	2 PE	4 PE	8 PE	16 PE	32 PE
bbara_bbtas	270(1.0)	174(1.55)	115(2.35)	108(2.5)	105(2.57)	117(2.31)
scf	268(1.0)	182(1.47)	140(1.91)	124(2.16)	123(2.18)	141(1.9)
s298	772(1.0)	493(1.57)	407(1.9)	291(2.65)	270(2.86)	241(3.2)
fetchXlog	3006(1.0)	1920(1.57)	1442(2.08)	1121(2.68)	1013(2.97)	1006(2.99)
dvramXmark1	2921(1.0)	1544(1.89)	1189(2.46)	1048(2.79)	972(3.0)	964(3.03)
planetXfetch	-	-	10480.0	8482.3	7400.6	6955.4

will also result in improvements to the parallel performance of the ProperSTATE algorithm with little effort.

Table 2.7 Quality of MUSTANG for different heuristics
Quality in literal counts

FSM	ProperSTATE			
BENCHMARK	Fanin	Fanout	Two-level-in	Two-level-out
dk16	308	297	286	309
dvram	247	257	234	244
fetch	152	162	152	168
keyb	196	239	271	239
log	134	134	131	150
planet	628	686	686	663
ram_test	843	847	821	938
s1488	763	735	737	669
s1494	739	694	731	704
s510	545	558	522	539
sand	709	778	754	769
sync	519	582	502	582
bbara_bbtas	1215	1266	1110	1266
scf	1118	1220	1046	1211
s298	8219	9039	8487	8856
fetchXlog	9315	9362	9240	9349
dvramXmark1	10501	10467	10248	10485

Table 2.8 ProperSTATE Quality of different heuristics for Uniprocessor runs
Quality in literal counts

FSM	ProperSTATE			
BENCHMARK	Fanin	Fanout	Two-level-in	Two-level-out
dk16	320	310	279	266
dvram	216	230	198	233
fetch	139	179	139	173
keyb	210	261	277	257
log	119	122	111	128
planet	647	694	617	609
ram_test	828	895	579	732
s1488	808	753	711	721
s1494	762	775	699	739
s510	467	565	428	476
sand	697	720	739	780
sync	439	565	340	420
bbara_bbtas	1266	1339	1064	1339
scf	1173	1239	918	1135
s298	8031	8965	8556	8953
fetchXlog	9315	9862	9139	9509
dvramXmark1	10501	10104	10071	10485

Table 2.9 ProperSTATE: Quality of different heuristics for Two-processor runs (For multiprocessor runs the output with the best gain estimate is picked for each heuristic)
Quality in literal counts

FSM	ProperSTATE			
BENCHMARK	Fanin	Fanout	Two-level-in	Two-level-out
dk16	294	303	298	302
dvram	227	270	211	246
fetch	147	166	144	177
keyb	249	276	287	258
log	119	141	110	125
planet	688	666	632	636
ram_test	648	763	595	719
s1488	762	703	754	763
s1494	723	672	750	719
s510	401	528	390	477
sand	799	682	767	805
sync	422	537	342	476
bbara_bbtas	1294	1208	1039	1175
scf	1212	1272	1002	1053
s298	8016	9060	8650	9281
fetchXlog	9125	9335	9043	9634
dvramXmark1	9980	10289	10305	10273

Table 2.10 ProperSTATE Quality of different heuristics for Four-processor runs (For multiprocessor runs the output with the best gain estimate is picked for each heuristic)
Quality in literal counts

FSM	ProperSTATE			
BENCHMARK	Fanin	Fanout	Two-level-in	Two-level-out
dk16	289	279	289	289
dvram	192	251	199	197
fetch	149	179	146	167
keyb	279	250	264	257
log	112	121	112	125
planet	675	659	620	639
ram_test	704	844	645	772
s1488	816	739	816	758
s1494	774	731	789	813
s510	387	561	396	528
sand	877	792	772	892
sync	495	581	427	538
bbara_bbtas	1066	1146	1049	1086
scf	1127	1219	1016	1082
s298	8741	8960	8739	9348
fetchXlog	9275	9644	9265	9775
dvramXmark1	9973	10508	10128	10523

CHAPTER 3

ProperJEDI : A SIMULATED ANNEALING BASED PARALLEL ALGORITHM FOR STATE ASSIGNMENT

3.1 Introduction

In this chapter we describe another parallel algorithm for state assignment of finite state machines (FSM). The parallel algorithm is based on the most recent sequential algorithm, JEDI [21]. JEDI uses a simulated annealing based encoding scheme which makes it computationally costly. It can take several hours for large circuits (see Table 3.2). Further, it uses four different options and is run four times to generate state assignments, the best of which is chosen. Hence the time requirements increase by a factor of four.

Our earlier parallel state assignment algorithm, ProperSTATE, was based on a greedy heuristic for encoding the states. A divide and conquer technique was used by partitioning the state space across processors and allowing each processor to encode on the basis of its local information. ProperSTATE is very parallel and gives results comparable to MUSTANG, the sequential algorithm it is based on [19]. But JEDI, the most recent state assignment algorithm, improves upon MUSTANG as well as ProperSTATE in quality. In contrast, the time requirements for JEDI are an order of magnitude more than ProperSTATE (even on 1 processor). Hence, ProperSTATE and JEDI offer a speed-quality trade off to a designer. Our parallel implementation of JEDI justifies itself with its remarkable speedups and good quality, thus achieving the best of both approaches.

This algorithm has also been implemented as part of the ProperCAD project. The major goal of the ProperCAD project [36] is to develop portable parallel algorithms for VLSI CAD applications that will run on a range of parallel machines including shared memory multiprocessors such as the Encore Multimax and SunSparcServer1000, distributed memory multicomputers such as the Intel iPSC/860, Intel Paragon, and Thinking Machines CM-5, and networks of workstations. ProperCAD II provides C++ library-based machine independent runtime support in an object–oriented manner [13].

The rest of this chapter is organized as follows: §3.2 describes the ProperJEDI algorithm in detail. Quality improvement is discussed in §3.3. §3.4 discusses the performance of the ProperJEDI algorithm on several parallel machines. §3.5 describes briefly a data partitioned version of ProperJEDI. We conclude the chapter with a discussion on future work in §3.6.

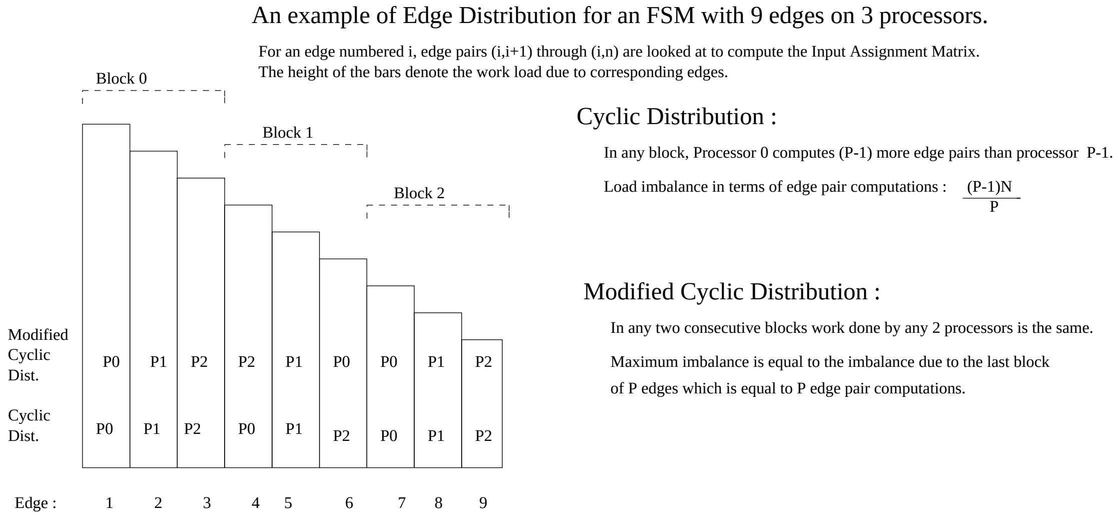

Figure 3.1 Modified Cyclic Distribution of Edges for Gain Function computation

3.2 ProperJEDI

Most of the state assignment algorithms operate in two phases. In phase 1 they generate an estimate of the affinity of states to each other. Two states which behave similarly in the FSM are given a high weight (priority) for encoding closely. Phase 2 encodes the states, using a suitable heuristic to maximize the sum of the estimates generated in phase 1. We will call the two phases the Gain Function Computation phase and the Encoding phase respectively.

In JEDI, there are four heuristics for generating weights between pairs of states: input dominant, output dominant, coupled and variation. All of them have been parallelized in ProperJEDI. The input dominant algorithm works on the source states of the transitions of the FSMs (the states from which the transitions are triggered). . Pairs of present states which assert similar outputs and produce similar sets of next states are given high edge weights. This has an effect of maximizing the size of common cubes in the Boolean functions corresponding to the output and the next state lines. The output dominant algorithm, on the other hand, works on the output states of the transitions. Pairs of next states which are produced by similar inputs and similar sets of present states are given high edge weights. This has an effect of maximizing the number of common cubes in the boolean functions corresponding to the next state lines. A coupled approach uses a hybrid of these two heuristics. Without loss of generality the description of our algorithm considers only the input dominant algorithm.

```
GAIN_COMPUTATION()
for each processor
do Read the state transition graph (STG).
    Update the data structures after reading each product term.
    /* Each edge in the STG corresponds to a product term */
    Compute the distance matrix H, where H(c1, c2) = Hamming distance
    between codes c1 and c2.
    /* Build the local part of the Input Assignment Matrix M_I as follows */
    for i = my_proc_num to i = number_of_edges in steps of P
    do /* A modified cyclic approach is used in the actual implementation */
        Look at the edge numbered i.
        for j = i + 1 to j = number_of_edges in steps of 1
        do Look at the states from which edges i and j fan out.
            Look at all the outputs produced on the transitions corresponding
            to these edges.
            Increment the Assignment matrix entry corresponding to
            these 2 states by the number of identical outputs.
    /* Add the local components of the Assignment Matrix
    by a tree merge */
    for i = 0 to i = log_2 P - 1 in steps of 1
    do Send the local gain matrix to the processor whose number
        differs in the ith bit.
        Receive the gain matrix from the processor whose number
        differs in the ith bit.
        Add the received matrix to the local gain matrix.
```

Figure 3.2 Gain Function Computation Phase of ProperJEDI (Input Dominant Algorithm)

3.2.1 Gain Function Computation

The input to ProperJEDI is an FSM in the form of a State Transition Graph (STG) which is replicated on all processors.

Let the number of states in the FSM be N_s and the number of transitions be N_e(number of nodes and edges in the STG respectively). An Input Assignment Matrix M_I is constructed in the gain function computation phase. It is an $N_s \times N_s$ symmetric matrix where each entry m_{xy} corresponds to the gain that will be realized if states s_x, s_y are encoded close together. Mathematically :

$$m_{xy} = \sum_{i=1}^{|O_x|} \sum_{j=1}^{|O_y|} P(o_{xi}, o_{yj})$$

where O_x and O_y are the number of transitions out of states s_x and s_y in the STG. o_{xi} is the set of binary outputs produced by the ith transition out of state s_x. Hence for calculating the input

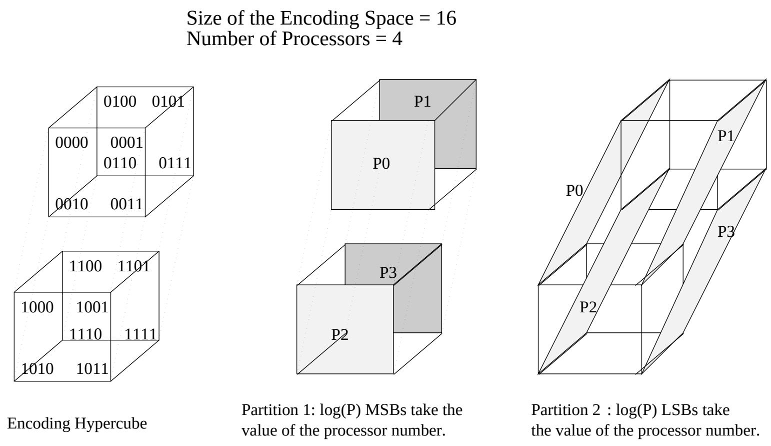

Figure 3.3 An example of Dynamic Repartitioning(This is also the minimum number of partitions for attempting any exchange in the global state space for this example. See §3.3.)

assignment matrix M_I, each pair of transitions in the STG is looked at and the corresponding entry in M_I is updated.

In the parallel algorithm, the edges are divided among processors in a *modified cyclic fashion* as shown in Figure 3.1. If there are n edges in the STG then for an edge numbered i, edge pairs $(i, i+1)$ through (i, N_e) have to be considered for building M_I. This gives rise to a triangular work load which justifies a cyclic distribution of edges. Our modified cyclic approach reduces the load imbalance further, from $O(N_e)$ to $O(P)$ where P is the number of processors.

Each processor builds a local input assignment matrix based on the edge pairs assigned to it. These local matrices are summed up using a tree based merge technique ($log_2 P$ communication steps per processor) to generate M_I on each processor. See Figure 3.2 for a detailed description of the parallel algorithm for this phase.

3.2.2 Encoding Phase

The encoding phase aims at assigning binary codes to states such that the following weighted sum is minimized:

$$\sum_{i=1}^{k} \sum_{j=i+1}^{k} m_{ij} H(b_i, b_j)$$

where b_i and b_j are binary codes for states s_i and s_j and m_{ij} is the gain estimate between them. Intuitively, if two states have a high gain estimate m_{ij} then the distance between their binary encodings $H(b_i, b_j)$ should be small. JEDI achieves this by a probabilistic hill climbing heuristic. States are randomly assigned to the vertices of the encoding hypercube. Two vertices are picked at random and their encodings are switched if it results in a reduction in the above cost. A local minima is avoided by probabilistically making a move even if it increases the cost estimate. This process is repeated until a sufficiently low cost is reached and making more moves appears futile.

The parallel algorithm for the encoding phase has to deal with two main issues. First it cannot allow any processor to exchange any two vertices. This would cause errors in the state space when many processors simultaneously try to exchange a particular vertex with other vertices. To avoid this, the encoding space is partitioned in a disjoint manner among processors. Each processor attempts a series of moves in its partition. But, if moves are restricted to static partitions, potentially useful moves across partitions would not be attempted. This would result in a degradation in quality. Our algorithm dynamically reconfigures the partitions in such a way that all possible exchange moves are explored. Further, it achieves this in minimum possible reconfigurations (see Figure 3.3). An analysis of quality control using repartitioning is given in §3.3.

The second issue is to avoid accumulation of errors, since a processor does not know what changes have been made outside its local space. A periodic global update of the state space at the time of repartitioning achieves this objective. Thus there are alternating stages of state update and annealing in ProperJEDI. But it may make unproductive moves in the middle of an annealing stage where it assumes that everything outside its local space is fixed (see Figure 3.6). Although such moves cannot be avoided, they can be minimized. Error control mechanism is discussed in detail in §3.3.

Figure 3.4 gives the encoding algorithm of ProperJEDI in detail.

3.3 Quality and Error Control

3.3.1 Quality Control

If each processor is restricted to work in its local space, it may result in a loss of quality. There has to be a mechanism to ensure that any two arbitrary states will be potential candidates for an exchange at some point of time. Our dynamic repartitioning scheme ensures this. At each global update step each processor recalculates the subset of the encoding space in which it has to work for the next annealing phase. This is achieved by fixing a subset of log_2P bits of the encoding space. These bits are ensured to be different for each processor (More precisely they represent the processor number in binary). At each repartitioning a different set of bits is chosen.

Let $b = \lceil log_2 N_s \rceil$ be the number of bits used to represent a code. The number of constrained bits $= log_2P$. We will call a bit position free , if it is not constrained. To explore all possible exchanges each bit position has to be free during some annealing phase. The number of free bits in each phase is $b - log_2P$. Therefore, the minimum number of partitions required is $\lceil \frac{b}{b-log_2P} \rceil$. This is also the minimum number of partitions required to carry out any arbitrary exchange in the global encoding space.

```
ENCODING_PHASE()
for  each processor
do Assign states randomly to the vertices (codes) of the encoding hypercube.
   Initialize temperature and other parameters for simulated annealing.
   Initialize the shift parameter which decides the local partition of
   the encoding space.
   repeat
           Calculate the number of moves k to be attempted at
           current temperature T.
           while number_of_attempted_moves  <  k
                 do Pick two vertices randomly from the local
                    encoding space.
                    Calculate the additional cost ΔG saved
                    by exchanging the two encodings.
                    If ΔG ≥ 0 accept the move.
                    else accept the move with probability e^(ΔG/T).
                    If the move is accepted update the cost estimate and
                    the local data structures.
                    Increment number_of_attempted_moves.
           Broadcast the information about the accepted moves to
           all processors.
           for i = 0 to i = P − 1 in steps of 1
           do Receive update information./ * It could come in any order * /
              Update the local data structures to reflect the accepted moves.
           / * At this stage the state space of all processors becomes identical * /
           Calculate the new shift and update the temperature T.
           Recalculate the new cost estimate.
     until stopping criterion is satisfied.
```

Figure 3.4 Encoding Phase of ProperJEDI

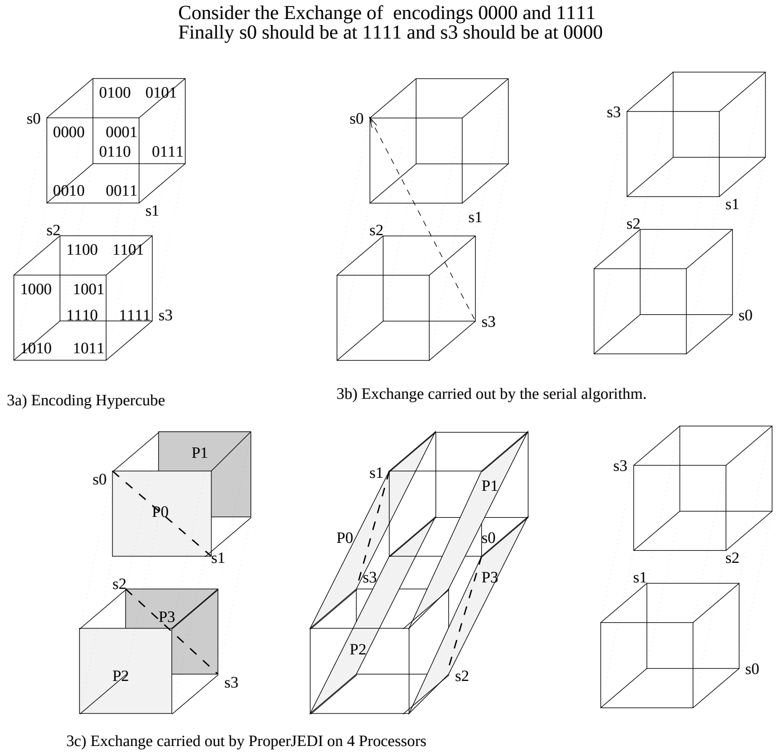

Figure 3.5 An example of an Indirect Exchanges through Dynamic Repartitioning

As a simple example consider a 4 bit encoding space (16 codes). Consider a 4 processor parallel annealing algorithm being carried out on this space. The minimum number of phases to enable any possible exchange in this case would be 2. During the first phase the two least significant bits are fixed (00 for processor 0, 01 for processor 1 and so on). The local space for processor 0 contains 0000,0001, 0010,0011. Similarly the local space for processor 3 contains 1100,1101, 1110,1111. A direct exchange between codes 0000 and 1111 can never be carried out since they would always lie on different processors.

This is achieved by dynamic repartitioning as follows. Let processor 0 exchange 0000 and 0011 and processor 3 exchange 1100 and 1111. In the next annealing phase the two least significant bits will be constrained. Hence, after the repartitioning stage, processor 0 owns codes 0000,0100,1000,1100 and processor 3 owns 0011,0111, 1011,1111. Now a local exchange of 0000 and 1100 on processor 0 and 0011 and 1111 on processor 3 has a net effect of exchanging 0000 and 1111 in the global space. Figure 3.5 illustrates the entire process.

3.3.2 Error Control

All processors are allowed to anneal their local partitions in parallel. This introduces some error in the cost function estimates. Figure 3.6 illustrates how this error can initiate unproductive moves.

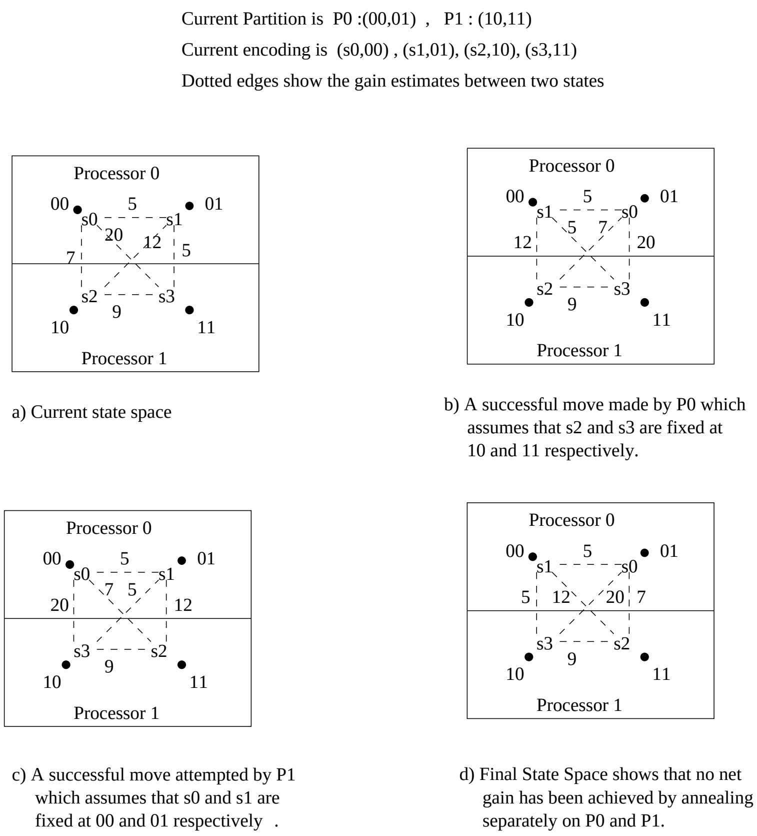

Figure 3.6 An example of unproductive move due to error in the cost estimate

Hence, an important criterion for the parallel algorithm is to control this error which accumulates with each move. The following analysis shows how our algorithm achieves this.

The annealing process consists of alternating stages of annealing and state update. A fixed number of moves is attempted after which the global state is updated to the latest information. At this stage the temperature is also updated to its new value. Assume for simplicity that the acceptance rate, α is constant for the entire simulated annealing process. Let the number of attempted moves for every annealing stage be k. The number of successful moves per stage $= k\alpha$ on an average. If the number of processors is P, the number of invalid state entries in a processor at the end of an annealing stage is $2(P-1)k\alpha$ (each successful move on some other processor affects two states in the worst case). The expected percentage error in the cost function is the ratio of the number of wrong state entries to the total number of states. Hence, the worst case percentage error is $\frac{2(P-1)k\alpha}{N_s}$. The task of our parallelization process was to keep this fraction bounded. If the problem size (N_s) doubles, the number of attempted moves also doubles. On the other hand, if the number of processor doubles, the parallel algorithm attempts half the moves on each processor. Further, with decreasing temperature, the acceptance rate α decreases. Our algorithm heuristically increases the number of attempted moves with a decrease in temperature so that $k\alpha$ remains constant across different annealing stages. Hence a constant bound on $\frac{2(P-1)k\alpha}{N_s}$ is maintained through out the algorithm. This bound is determined by the initial value of parameter k.

3.4 Experimental Results

The ProperJEDI parallel algorithm described in the previous sections has been implemented as part of the ProperCAD project [13]. In this section, we will describe the results obtained by applying ProperJEDI on various circuits in the MCNC benchmark suite. The output obtained from ProperJEDI is passed through multilevel logic optimization scripts of SIS, a logic synthesis tool from Berkeley. The quality of the solution is measured as the number of literals in a factored form of the logic after optimization. Less literals imply less area for implementing the combinational logic and hence better quality.

The speedups are reported for SparcServer1000 shared memory multiprocessor, CM-5 distributed memory multicomputer and a network of Sun workstations. A cross product generator (described in previous chapter) was used to generate some large benchmark FSMs.

As seen from Tables 3.1, 3.2 and 3.3 the speedups are good for both shared memory and distributed memory machines and increase with increasing problem size. They range from 50-80

[1] The actual time taken by JEDI (and ProperJEDI) is the sum of the individual times taken by the four weight generation heuristics.n

[2] The scaled count for 32 and 64 processor results is calculated using only the corresponding entries of JEDI.

Table 3.1 ProperJEDI on a CM-5 (Dashed entries correspond to small problem sizes)[1]
Time in seconds

FSM BNCHMARK	JEDI	ProperJEDI			
		1 PE	4 PE	16 PE	64 PE
dk16	84.1	24.01(1.0)	10.87(2.21)	7.13(3.37)	-
dvram	172.7	25.51(1.0)	12.32(2.07)	8.27(3.08)	-
fetch	69.3	14.27(1.0)	6.88(2.16)	5.81(2.46)	-
keyb	36.4	17.06(1.0)	7.50(2.27)	6.52(2.62)	-
log	24.7	9.37 (1.0)	4.29(2.18)	5.25(1.78)	-
planet	463.9	93.73(1.0)	60.40(1.55)	14.53(6.45)	-
ram_test	1830.1	254.52(1.0)	108.75(2.34)	32.19(7.91)	-
s1488	602.0	161.20(1.0)	107.31(1.5)	26.3(6.13)	-
s1494	589.1	171.81(1.0)	106.67(1.61)	23.95(7.17)	-
s510	435.5	68.33(1.0)	25.36(2.7)	12.16(5.62)	-
sand	167.2	55.61(1.0)	23.7(2.35)	10.02(5.56)	-
sync	613.6	78.94(1.0)	39.1(2.02)	13.2(5.99)	-
bbara_bbtas	17368.3	14603.4(1.0)	3432.8(4.25)	837.1(17.44)	221.8(65.8)
scf	9589.0	1717.86(1.0)	760.35(2.26)	179.7(9.56)	54.71(31.4)
s298	Timeout	Timeout	Timeout	4780.24	1289.9
fetchXlog	Timeout	Timeout	Timeout	Timeout	9706.17
dvramXmark1	Timeout	Timeout	Timeout	Timeout	8235.5

Table 3.2 ProperJEDI on a SparcServer1000 (8 processor shared memory multiprocessor)
Time in seconds(Speedups)

FSM BENCHMARK	JEDI	ProperJEDI			
		1 PE	2 PE	4 PE	8 PE
dk16	26.42	10.48(1.0)	6.68(1.57)	11.94(0.88)	13.27(0.79)
dvram	54.39	12.75(1.0)	7.72(1.65)	20.02(0.64)	26.90(0.47)
fetch	21.35	8.53(1.0)	5.26(1.62)	15.42(0.55)	18.12 (0.47)
keyb	10.88	8.60(1.0)	5.48(1.57)	7.89(1.09)	28.37(0.3)
log	5.98	4.94(1.0)	3.13(1.58)	6.03(0.82)	11.80(0.17)
planet	149.68	36.80(1.0)	25.63(1.44)	44.52(0.83)	33.52(1.1)
ram_test	601.62	90.22(1.0)	59.24(1.52)	40.54(2.22)	33.91(2.66)
s1488	194.71	61.56(1.0)	47.88(1.28)	56.26(1.1)	39.21(1.57)
s1494	190.89	61.34(1.0)	40.93(1.5)	44.31(1.38)	28.35(2.16)
s510	136.95	27.63(1.0)	18.16(1.52)	22.35(1.24)	26.22(1.05)
sand	53.34	25.18(1.0)	18.13(1.39)	23.32(1.08)	23.46(1.07)
sync	201.61	34.62(1.0)	24.55(1.41)	20.03(1.73)	29.57(1.17)
bbara_bbtas	5779.20	2505.3(1.0)	1356.2(1.85)	679.6(3.7)	359.6(6.97)
scf	3140.12	554.56(1.0)	311.17(1.78)	178.36(3.11)	90.35(6.14)
s298	32450.78	14051.4(1.0)	7102.0(1.98)	3820.1(3.68)	1962.9(7.16)
fetchXlog	Timeout	Timeout	Timeout	30954.46	16291.8
dvramXmark1	Timeout	Timeout	Timeout	28916.92	15421.2

Table 3.3 ProperJEDI on a Network of Sun SPARC Station 5 Workstations
Time in seconds(Speedups)

FSM BENCHMARK	JEDI	ProperJEDI		
		1 PE	2 PE	4 PE
dk16	40.44	14.76(1.0)	10.41(1.42)	7.42(1.99)
dvram	81.44	18.53(1.0)	10.63(1.74)	9.67(1.92)
fetch	32.74	11.23(1.0)	7.10(1.58)	5.12(2.2)
keyb	17.67	10.70(1.0)	7.69(1.39)	5.02(2.13)
log	8.71	6.46(1.0)	4.23(1.53)	3.60(1.79)
planet	212.60	50.04(1.0)	32.82(1.52)	17.90(2.79)
ram_test	856.01	117.64(1.0)	71.35(1.65)	36.30(3.24)
s1488	288.81	86.04(1.0)	57.6(1.49)	29.72(2.9)
s1494	292.38	84.31(1.0)	54.64(1.54)	30.10(2.8)
s510	287.92	38.49(1.0)	22.32(1.72)	13.04(2.95)
sand	84.86	30.32(1.0)	18.12(1.67)	11.39(2.66)
sync	201.25	42.27(1.0)	26.28(1.61)	20.47(2.06)
bbara_bbtas	8498.94	3341.25(1.0)	2062.53(1.62)	1120.13(2.98)
scf	4438.79	735.34(1.0)	393.32(1.87)	253.32(2.92)
s298	48581.39	18815.1(1.0)	10380.32(1.81)	5682.76(3.3)
fetchXlog	Timeout	Timeout	Timeout	Timeout
dvramXmark1	Timeout	Timeout	Timeout	Timeout

Table 3.4 Quality of ProperJEDI (The dashed entries correspond to small problem sizes)[2]
Quality in Literal Counts

FSM BENCHMARK	JEDI	Parallel JEDI						
		1 PE	2 PE	4 PE	8 PE	16 PE	32 PE	64 PE
dk16(27)	219	219	209	239	255	240	-	-
dvram(35)	187	187	177	197	199	195	-	-
fetch(26)	129	129	132	140	155	160	-	-
keyb (19)	215	215	223	206	192	218	-	-
log (17)	109	109	115	117	123	125	-	-
planet(48)	569	569	572	593	589	578	-	-
s1488(48)	705	705	731	693	710	723	-	-
s1494(48)	704	704	701	729	718	711	-	-
s510 (47)	381	381	385	391	370	383	-	-
sand (32)	692	692	702	679	712	715	-	-
sync (52)	351	351	332	401	426	377	-	-
ram_test(72)	527	527	561	571	667	581	642	-
bbara_bbtas(128)	837	837	923	716	859	903	853	884
scf (121)	957	957	1060	981	968	1018	995	1048
s298 (218)	8034	8034	8256	8142	7971	8291	8591	8142
fetchXlog (442)	9291	9291	9423	9657	9304	9508	9342	9508
dvramXmark1(423)	10290	10290	10172	10201	10505	10215	10294	10268
Scaled Count	100	100	100.1	100.5	100.8	100.7	100.07	100.05

Table 3.5 Effect of the Number of Partitions per temperature point(NumP) on Literal Count for Input Dominant Heuristic

FSM	4 Processors		8 Processors		16 Processors	
BENCHARK	2 NumP	1 NumP	2 NumP	1 NumP	3 NumP	1 NumP
s1488	726	693	714	710	766	745
ram_test	701	744	763	760	948	867
bbara_bbtas	1051	716	1047	990	1252	1178
scf	1220	1299	1220	1180	1271	1176
s298	8843	8760	8466	8791	8342	8291
fetchXlog	9591	9657	9657	9304	9346	9508
dvramXmarkl	10290	10201	10201	10505	10501	10215

for 64 processor runs on CM-5. We attribute the superlinear speedups as seen in Table 3.1 to the cache effects. Since the amount of data accessed by a processor decreases with increasing number of processors, the cache hit ratio increases. Further, the stopping criterion for simulated annealing may get satisfied early and result in superlinear effects. There were some optimizations performed in the serial code during the task of parallelization. As a result, the uniprocessor implementation of ProperJEDI is also more efficient than JEDI as seen in Tables 3.2 and 3.3. Note that for the smaller benchmarks runs have not been taken on 32 or 64 processor configurations. For example, *s1488* and *planet* have an encoding hypercube of size 64. For a 64-processor run each processor will have exactly 1 vertex for any partition. Thus no simulated annealing move can be made. Hence the results have not been reported for these cases. Similarly a 32-processor run would hardly have any exchange alternatives.

Table 3.4 gives the literal count of the solutions produced by ProperJEDI after optimization through SIS. The quality reported is the best of the four weight generation heuristics in each case. The results produced by JEDI are the same as those of ProperJEDI on 1 processor. This is because the same pattern of moves as JEDI, is tried with the same outcome in the uniprocessor run of ProperJEDI. The quality degradation for multiprocessor runs is minimal. The worst degradation is around 3% for an 8 processor run. Note that with increasing number of processors the quality remains stable showing that ProperJEDI scales well with the number of processors. To illustrate the power of ProperJEDI observe that the time requirements for *fetchXlog* have been brought down from 11 hours to less than 10 minutes for a 64 processor run on CM-5. The corresponding quality loss suffered is just about 3%.

Since dynamic repartitioning improves the quality, it seems logical that trying out more partitions at a given temperature during simulated annealing would further improve the quality. We experimented and found the results contrary to our expectation as seen in Table 3.5. Since the

Table 3.6 PartJEDI on a Network of Sun Workstations
Time in seconds(Speedups)

FSM BENCHMARK	JEDI	PartJEDI		
		1 PE	2 PE	4 PE
dk16	40.44	23.65(1.0)	15.99(1.48)	10.49(2.25)
dvram	81.44	26.00(1.0)	20.88(1.25)	16.67(1.56)
fetch	32.74	14.66(1.0)	10.75(1.36)	7.68(1.91)
keyb	17.67	17.18(1.0)	13.42(1.28)	10.50(1.64)
log	8.71	9.00(1.0)	8.52(1.06)	6.03 (.49)
planet	212.60	83.53(1.0)	48.01(1.74)	32.16(2.6)
ram_test	856.01	200.74(1.0)	115.55(1.74)	71.04(2.83)
s1488	288.81	151.77(1.0)	99.47(1.53)	59.64(2.54)
s1494	292.38	169.72(1.0)	92.56(1.83)	63.21 (2.68)
s510	287.92	69.69(1.0)	40.49(1.72)	23.58(2.96)
sand	84.86	52.05(1.0)	31.66(1.64)	18.94(2.75)
sync	201.25	78.01(1.0)	48.58(1.61)	26.16(2.98)
bbara_bbtas	8498.94	6288.76(1.0)	3931.37(1.6)	2076.61(3.03)
scf	4438.79	1335.79(1.0)	829.80(1.61)	483.14(2.77)
s298	48581.39	34320.44(1.0)	19768.14(1.74)	8844.34 (3.88)
fetchXlog	Timeout	Timeout	Timeout	Timeout
dvramXmark1	Timeout	Timeout	Timeout	Timeout

number of attempted moves is fixed for a particular temperature, more partitions imply lesser attempted moves per partition. Hence we are implicitly restricting our exploration by not allowing sufficient annealing attempts to be made for a given partition. In order to overlap computation with communication, temperature update and repartitioning should be carried out at the same time as the global space update. This makes the choice of exploring 1 partition per temperature point natural to the algorithm.

3.5 Data Partitioned Implementation of JEDI

The prominent memory requirements for ProperJEDI are in the intermediate data structures which store the N_s^2 gain estimates and N_c^2 Hamming distances in the form of matrices, where N_s is the number of states in the FSM and N_c is the number of binary codes in the encoding hypercube. To avoid conflicts and inconsistencies during parallel simulated annealing phases, each processor works on a disjoint partition of the encoding hypercube. It exchanges vertices of only its own local part. Consequently, in order to estimate the delta cost of a move a processor needs only a part of the distance and gain matrices. The relevant part of the distance matrix, $\frac{N_c^2}{P}$ in size corresponds to $\frac{N_c}{P}$ columns of N_c Hamming distances, each column for a local code. Correspondingly, the only

Table 3.7 Run time Comparison of Replicated and Partitioned JEDI for single processor runs on SunSparcServer for the four weight generation heuristics
Time in seconds

FSM	Input		Output		Coupled		Modified	
BENCHMARK	Part.	Repl.	Part.	Repl.	Part.	Repl.	Part.	Repl.
dk16	2.53	1.97	2.51	1.71	7.12	3.95	4.54	2.85
dvram	3.90	2.87	3.14	2.20	7.19	4.36	5.07	3.32
fetch	2.76	2.29	2.20	1.69	3.51	2.31	2.87	2.24
keyb	2.18	1.53	2.45	1.55	4.52	3.44	3.42	2.08
log	1.58	1.14	1.43	0.96	1.94	1.44	1.89	1.40
planet	6.02	4.06	5.46	3.83	34.23	18.08	19.87	10.83
ram_test	11.82	8.71	12.44	9.34	91.21	46.35	53.61	25.82
s1488	13.85	7.84	7.49	5.02	67.93	31.81	31.91	16.89
s1494	12.86	7.62	7.71	5.12	61.67	32.02	31.32	16.74
s510	4.76	3.23	6.79	4.51	21.14	11.16	15.17	8.73
sand	4.79	4.09	5.57	3.86	18.68	9.55	11.23	7.68
sync	6.14	4.56	7.53	5.42	26.46	14.41	17.22	10.23
bbara_bbtas	39.39	22.96	43.45	25.91	3231.97	1611.99	1694.18	843.05
scf	26.34	20.74	22.30	15.82	687.71	341.87	355.05	177.04
s298	111.79	68.10	80.32	49.62	20523.12	9278.87	9668.07	4656.19

relevant gain estimates are the ones whose states are encoded on the local partition of the encoding hypercube. In the worst case all of the N_c codes may be assigned to states. So the worst case size of the gain matrix partition which a processor needs to keep is $\frac{N_c}{P}$ columns of size N_c each. Introducing this modification reduces the memory requirement from N_c^2 to $\frac{N_c^2}{P}$. Thus the memory bottleneck in these $O(N^2)$ data structures is reduced by a factor of P, the number of processors.

A lot of bookkeeping and communication of massive chunks has to be performed for implementing this approach. It was incrementally developed from the data replicated version of ProperJEDI. All moves are proposed in the global address space. The cost evaluation of the moves is done by appropriately mapping the global values of code and state numbers to their corresponding local values. For this purpose two matrices of size $\frac{N_c}{P}$ are maintained.

At each repartitioning stage, a processor gets a different part of the encoding space reassigned to it. The information about the new local codes and states is communicated to it by the processors which owned those particular states and codes in the previous partition. This communication step involves P^2 messages of size $\frac{2N_c^2}{P^2}$ each and is responsible for significant time overheads in the implementation. A look at Table 3.7 shows that the time taken by the data partitioned version is almost twice the time taken by the data replicated version of ProperJEDI. Table 3.6 shows the time requirements and speedups of Data Partitioned ProperJEDI on a network of workstations where the

Table 3.8 Parallel JEDI Quality for Uniprocessor runs of different heuristics
Quality in literal counts

FSM	Parallel JEDI			
BENCHMARK	Input	Output	Coupled	Modified
dk16	219	263	245	263
dvram	187	208	209	229
fetch	163	147	129	150
keyb	215	236	224	228
log	116	115	109	109
planet	581	623	569	650
ram_test	646	527	647	682
s1488	741	730	732	705
s1494	749	742	704	741
s510	516	399	399	381
sand	732	692	752	714
sync	497	351	417	443
bbara_bbtas	951	1149	837	1063
scf	1141	1041	957	1057
s298	8787	8034	8326	8512
fetchXlog	9291	Timeout	Timeout	Timeout
dvramXmark1	10290	Timeout	Timeout	Timeout

communication latency is high and the massive communication overhead is very severe. The results are very encouraging. The quality of the results for both the implementations is the same because the same moves are attempted with identical results on all the processors. Tables 3.8, 3.9, 3.10 and 3.11 show the quality of ProperJEDI for different heuristics for different number of processors.

3.6 Summary

The compute intensive nature of simulated annealing forces a designer to spend a lot of time or else suffer a loss in quality of the results. The parallelization of such applications provide the designer an option of saving time or annealing longer to get an improvement in quality. Until now the parallel implementation of annealing applications was restricted mainly to cell placement. We have successfully applied it to the problem of state assignment. We have demonstrated that it is possible to design and implement a portable parallel state assignment tool that is cleanly and efficiently interfaced with JEDI using the ProperCAD environment. We have shown that ProperJEDI gives very good speedups on both, shared memory machines and message passing machines in this manner. Further, the quality loss is minimal making it an attractive tool for reducing the time requirement in the design cycle.

Table 3.9 Parallel JEDI Quality for 2-processor runs of different heuristics
Quality in literal counts

FSM	Parallel JEDI			
BENCHMARK	Input	Output	Coupled	Modified
dk16	239	238	209	238
dvram	212	219	177	219
fetch	160	155	132	152
keyb	253	223	243	226
log	115	124	116	119
planet	627	647	572	607
ram_test	711	608	561	651
s1488	758	731	737	755
s1494	701	787	768	752
s510	475	409	385	401
sand	729	723	702	709
sync	483	332	386	432
bbara_bbtas	923	1104	948	1056
scf	1179	1071	1091	1060
s298	8957	8282	8256	8410
fetchXlog	9423	Timeout	Timeout	Timeout
dvramXmark1	10172	Timeout	Timeout	Timeouta

Table 3.10 Parallel JEDI Quality for 4-processor runs of different heuristics
Quality in literal counts

FSM	Parallel JEDI			
BENCHMARK	Input	Output	Coupled	Modified
dk16	245	239	263	239
dvram	197	223	201	212
fetch	164	152	140	150
keyb	215	206	213	223
log	124	119	121	117
planet	624	593	617	625
ram_test	744	659	571	655
s1488	693	711	758	732
s1494	744	748	729	740
s510	508	415	391	458
sand	748	700	679	694
sync	480	414	421	401
bbara_bbtas	716	1015	963	1050
scf	1299	1037	1028	981
s298	8760	8735	8142	8483
fetchXlog	9657	Timeout	Timeout	Timeout
dvramXmark1	10201	Timeout	Timeout	Timeout

Table 3.11 Parallel JEDI Quality for 8-processor runs of different heuristics
Quality in literal counts

FSM	Parallel JEDI			
BENCHMARK	Input	Output	Coupled	Modified
dk16	262	279	255	281
dvram	222	237	199	221
fetch	160	155	163	184
keyb	192	220	217	219
log	134	127	124	123
planet	589	685	620	635
ram_test	760	671	667	668
s1488	710	751	720	803
s1494	718	762	783	807
s510	543	440	370	446
sand	739	712	732	746
sync	562	426	437	477
bbara_bbtas	990	998	859	1037
scf	1180	1082	968	1025
s298	8791	7971	8119	8757
fetchXlog	9304	Timeout	Timeout	Timeout
dvramXmark1	10505	Timeout	Timeout	Timeout

CHAPTER 4

CONCLUSIONS

4.1 Summary of Thesis Work

In our work, we have explored two popular state assignment algorithms MUSTANG and JEDI. We have come up with partitioned parallel implementations of these algorithms which are portable across a wide range of MIMD architectures. In this chapter we revisit the motivation for our work and summarize our results. We end this chapter with a discussion on future research directions.

The complexity of VLSI circuits is increasing very rapidly and the existing CAD algorithms will not be able to handle large circuits in the near future due to prohibitive run times and memory requirements. Researchers have started using parallel processing to speed up the computation time for VLSI design. But the parallel algorithms proposed for CAD applications are designed with a specific underlying architecture in mind. As a result, these programs perform poorly on architectures other than the one for which they were designed. Moreover, incompatibilities in programming environments also make it difficult to port these programs across different parallel architectures.

This limitation has serious consequences. A new parallel algorithm has to be developed for every target MIMD 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.

The major goal of the ProperCAD project [6] is to develop portable parallel algorithms for VLSI CAD applications that run on a range of parallel machines including shared memory multiprocessors such as the SGI Challenge and the SunSparcServer1000, distributed memory multicomputers such as the Intel Paragon, and the Thinking Machines' CM5, and a network of workstations. The ProperCAD group at the University of Illinois is currently involved in building a parallel portable synthesis tool prototype which would automate the various stages involved in taking the behavioral description of a circuit to its physical layout.

Finite State Machines form an important class of VLSI circuits and are especially useful for designing the control unit of various microprocessors. In general, a huge sequential circuit can be considered to be a gaint FSM. In such a scenario, even a circuit with modest number of latches (say 20) has more than a million states. Researchers have tried to use techniques like FSM decomposition and isolation of smaller interacting FSMs to make the time and memory requirements manageable [23]. These smaller sub-circuits are locally optimized by conventional techniques like state minimization and optimal state assignment [24].

Table 4.1 ProperJEDI vs ProperSTATE on 4 processors on SunSparcServer

FSM	ProperJEDI		ProperSTATE		
BENCHMARKS	Time	Literal Count	Time	Literal Count	Ratio to JEDI
dk16(27)	11.94	239	0.1	279	1.167
dvram(35)	20.02	197	0.1	192	0.975
fetch(26)	15.42	140	0.1	146	1.043
keyb (19)	7.89	206	0.1	250	1.21
log (17)	6.03	117	0.1	112	0.96
planet(48)	44.52	593	0.2	620	1.045
ram_test(72)	40.54	571	0.2	704	1.233
s1488(48)	56.26	693	0.1	739	1.066
s1494(48)	44.31	729	0.1	731	1.003
s510 (47)	22.35	391	0.1	387	0.99
sand (32)	23.32	679	0.1	772	1.137
sync (52)	20.03	401	0.2	427	1.065
bbara_bbtas	679.61	716	0.8	1049	1.465
scf	178.36	981	2.8	1016	1.036
s298	3820.13	8142	5.6	8739	1.073
fetchXlog	30954.46	9657	39.2	9265	0.96
dvramXmarkl	28916.92	10201	155.6	9973	0.978
Scaled Count	-	17(100)	-	108.3	18.41

The penalty for lack of global optimization techniques severely affects the quality of the results. In other words, the area requirement is significantly more for a piecewise locally optimized sequential machine than for a globally optimized one. To this effect our attempts to parallelize the recent state assignment algorithms assume significance. It has been amply demonstrated that a significant reduction in time and memory requirements can be achieved for the state assignment of huge FSMs, with negligible compromise in quality.

While ProperSTATE provides the option of very small run times, ProperJEDI gives better quality at the cost of more computation time. Partitioned implementation of JEDI as well as ProperSTATE show a reduction in memory requirements with increasing number of processors.

4.2 ProperSTATE vs Partitioned JEDI

Table 4.1 shows that ProperJEDI gives better quality than ProperSTATE. Further, with increasing number of processors the quality of ProperSTATE deteriorates but that of ProperJEDI remains stable. Although the time requirement of ProperJEDI is much more than ProperSTATE, it is not prohibitive and has been brought down considerably by the task of parallelization. Hence

ProperJEDI reduces the enormous time requirement of JEDI and gives a quality improvement over ProperSTATE, combining the best of both the algorithms.

4.3 Future Research Directions

Huge Finite State Machines state assigned by ProperSTATE or ProperJEDI are too large to be handled by SIS for logic optimization. Thus logic optimization kernels of SIS prove to be a bottleneck in the global optimization of huge circuits. A partitioned parallel implementation of SIS, ProperSIS is in progress in the ProperCAD group which will handle bigger circuits and take lesser time [5]. A TimberWolf based parallel cell placement tool has been already developed in our group [28]. In the near future, we hope to integrate the three tools to provide a path from the behavioral description of a circuit to its physical layout. This would be our first primitive parallel, portable synthesis tool prototype.

The algorithms that we concentrated on in this work, namely MUSTANG and JEDI, both target multi-level implementations [19, 21]. PLA based two-level implementations of FSMs are still very popular, typically for the synthesis of control part of a circuit. Algorithms targetting two-level implementations like NOVA are also compute intensive and deserve some attention [20]. Parallelization of such an algorithm would give us a rich library of parallel state assignment tools which would offer an attractive choice based on desired design time, quality and type of implementation.

Another problem closely associated with State Assignment is that of State Minimization. This is an important technique for reducing the complexity of FSMs and providing more flexibility in terms of don't cares to be exploited by the state assignment and the subsequent logic optimization phases. Although this problem is well studied for completely specified FSMs, it has recently been revisited for incompletely specified FSMs. Like state assignment, minimization is also compute intensive and would benefit by parallelization especially when the search space consists of millions of states.

Recently, design for low power has caught the attention of the researchers [37, 38]. With diminishing chip size and increasing level of integration the power consumption has become a critical parameter in the design process. Most of the ideas in the sequential synthesis domain propose a comprehensive cost model for potential power consumption in the circuit. State assignment is then guided by this cost function to yield a more power efficient implementation [39, 40]. the cost model involves calculation of probabilities of various transitions and being in a particular state. We believe that the model is very compute intensive and will benefit either in accuracy or in time requirement by a parallel implementation. Moreover, the modular implementations of ProperSTATE and ProperJEDI may enable a replacement of the area-estimate based cost function by a power-estimate based one. This would significantly reduce the development time for such a parallel application, thus demonstrating yet another advantage of ProperCAD as an object-oriented platform for parallel CAD applications.

REFERENCES

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

[2] J. S. Rose, W. M. Snelgrove, and Z. G. Vranesic, "Parallel cell placement algorithms with quality equivalent to simulated annealing," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 387–396, Mar. 1988.

[3] K. P. Belkhale and P. Banerjee, "Parallel algorithms for VLSI circuit extraction," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 604–618, May 1991.

[4] S. Patil and P. Banerjee, "A parallel branch and bound algorithm for test generation," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 313–322, Mar. 1990.

[5] K. De, J. A. Chandy, S. Roy, S. Parkes, and P. Banerjee, "Portable parallel algorithms for logic synthesis using the mis approach," in *Proceedings of the International Parallel Processing Symposium*, Santa Barbara, CA, pp. 579–585, Apr. 1995.

[6] B. Ramkumar and P. Banerjee, "ProperCAD: A portable object-oriented parallel environment for VLSI CAD," in *Proceedings of the International Conference on Computer Design*, Cambridge, MA, pp. 544–548, Oct. 1992.

[7] W. Fenton, B. Ramkumar, V. A. Saletore, A. B. Sinha, and L. V. Kalé, "Supporting machine independent programming on diverse parallel architectures," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, Aug. 1991.

[8] B. Ramkumar and P. Banerjee, "ProperEXT: A portable parallel algorithm for VLSI circuit extraction," in *Proceedings of the International Parallel Processing Symposium*, pp. 434–438, 1993.

[9] B. Ramkumar and P. Banerjee, "Portable parallel test generation for sequential circuits," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, pp. 220–223, Nov. 1992.

[10] K. De, B. Ramkumar, and P. Banerjee, "ProperSYN: A portable parallel algorithm for logic synthesis," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, pp. 412–416, Nov. 1992.

[11] S. Kim, "Improved algorithms for cell placement and their parallel implementations," Ph.D. Dissertation, University of Illinois at Urbana-Champaign, July 1993. Tech. Rep. CRHC-93-18/UILU-ENG-93-2231.

[12] S. Parkes, J. A. Chandy, and P. Banerjee, "ProperCAD II: A run-time library for portable, parallel, object-oriented programming with applications to VLSI CAD," Center for Reliable and High-Performance Computing, University of Illinois, Urbana, Illinois, Tech. Rep. CRHC-93-22/UILU-ENG-93-2250, Dec. 1993.

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

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

[15] S. Parkes, P. Banerjee, and J. H. Patel, "ProperHITEC: A portable, parallel, object–oriented approach to sequential test generation," in *Proceedings of the Design Automation Conference*, San Diego, CA, pp. 717–721, June 1994.

[16] S. Parkes, P. Banerjee, and J. Patel, "A parallel algorithm for fault simulation based on proofs," in *Proceedings of the International Conference on Computer Design*, Austin, TX, Oct. 1995. To appear.

[17] S. Kim, J. A. Chandy, S. Parkes, B. Ramkumar, and P. Banerjee, "ProperPLACE: A portable parallel algorithm for cell placement," in *Proceedings of the International Parallel Processing Symposium*, Cancun, Mexico, pp. 932–941, Apr. 1994.

[18] A. A. Chien, *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs.* Cambridge, MA: The MIT Press, 1993.

[19] S. Devdas, H. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: state assignment of finite state machines targetting multilevel logic implementations," *IEEE Transactions on Computer Aided Design*, pp. 1290–1300, Dec. 1988.

[20] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: state assignment of finite state machines for optimal two-level logic implementation," *IEEE Transactions on Computer Aided Design*, pp. 905–924, Sept. 1990.

[21] B. Lin and A. Newton, "Synthesis of multiple level logic from symbolic high-level description languages," *Proceedings of the International Conference on VLSI*, pp. 187–196, Aug. 1989.

[22] E. M. Sentovich *et al.*, "SIS: a system for sequential circuit synthesis," Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Berkeley,, Tech. Rep., 1992.

[23] P. Ashar, A. Ghosh, S. Devadas, and A. Newton, "Implicit state transition graphs: Applications to sequential logic synthesis and test," *International Conference on Computer Aided Design*, pp. 84–87, Nov. 1990.

[24] J.-K. Rho, G. Hatchel, and F. Somenzi, "Don't care sequences and the optimization of interacting finite state machines," *International Conference on Computer Aided Design*, pp. 418–421, Nov. 1991.

[25] M. D. Durand, "Accuracy vs. speed in placement," *IEEE Design & Test of Computers*, pp. 8–34, June 1989.

[26] D. R. Greening, "Parallel simulated annealing techniques," *Physica*, vol. D42, pp. 293–306, 1990.

[27] S. A. Kravitz and R. A. Rutenbar, "Placement by simulated annealing on a multiprocessor," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 534–549, July 1987.

[28] J. A. Chandy and P. Banerjee, "Parallel simulated annealing strategies for VLSI cell placement." submitted to Ninth International Conference on VLSI Design, Jan. 1996.

[29] P. Banerjee and M. Jones, "A parallel simulated annealing for standard cell placement on a hypercube computer," in *Digest of Papers, International Conference on Computer-Aided Design*, pp. 34–37, Nov. 1986.

[30] P. Banerjee, M. H. Jones, and J. S. Sargent, "Parallel simulated annealing algorithms for standard cell placement on hypercube multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, pp. 91–106, Jan. 1990.

[31] A. Casotto and A. Sangiovanni-Vincentelli, "Placement of standard cells using simulated annealing on the Connection Machine," in *Digest of Papers, International Conference on Computer-Aided Design*, Santa Clara, CA, pp. 350–353, Nov. 1987.

[32] C.-P. Wong and R.-D. Fiebrich, "Simulated annealing-based circuit placement algorithm on the Connection Machine system," in *Proceedings of the International Conference on Computer Design*, Rye Brook, NY, pp. 78–82, Oct. 1987.

[33] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A parallel simulated annealing algorithm for the placement of macro-cells," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 838–847, Sept. 1987.

[34] F. Darema, S. Kirkpatrick, and V. A. Norton, "Parallel algorithms for chip placement by simulated annealing," *IBM J. Research and Development*, vol. 31, pp. 391–402, May 1987.

[35] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin Cummings, 1993.

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

[37] M. Aldina, J. Monteiro, A. Ghosh, and S. Devadas, "Precomputation-based sequential logic optimization for low power," *International Conference on Computer Aided Design*, pp. 74–81, Nov. 1994.

[38] G. D. Hatchel, M. Hermida, A. Pardo, M. Poncino, and F. Somenzi, "Re-encoding sequential circuits to reduce power dissipation," *International Conference on Computer Aided Design*, pp. 70–73, Nov. 1994.

[39] E. Olson and S. Kang, "Low power state assignment for finite state machines search," *International Workshop on Low Power Design*, pp. 63–68, Oct. 1992.

[40] C.-Y. Tsui, M. Pedram, C.-A. Chen, and A. M. Despain, "Low power state assignment targetting two and multi-level logic implementations," *International Conference on Computer Aided Design*, pp. 82–87, Nov. 1994.

VITA

Gagan Hasteer did his undergraduate education in Indian Institute of Technology, New Delhi (IIT Delhi) and received the B.Tech. degree in Computer Science in 1994. In August 1994, Gagan enrolled in the University of Illinois at Urbana-Champaign; he received the M.S. in January 1996.

Gagan has received the prestigious National Talent Search scholarship in India. He has the honor of securing the 3rd position in the All India Joint Entrance Examination for IITs. He was a recepient of IIT merit scholarship throughout his undergraduate education. He is active in Sequential Synthesis and Optimizations for CAD and parallel and distributed computing.