A Parallel State Assignment Algorithm for Finite State Machines

David A. Bader*
Electrical and Computer Engineering Department
University of New Mexico, Albuquerque, NM 87131
dbader@ece.unm.edu

Kamesh Madduri†
Department of Electrical Engineering
Indian Institute of Technology Madras
Chennai, India
kamesh@ee.iitm.ernet.in

October 3, 2003

Abstract

This paper summarizes the design and implementation of a parallel algorithm for state assignment of large Finite State Machines. High performance CAD tools are necessary to overcome the computational complexity involved in the optimization of large sequential circuits. FSMs constitute an important class of logic circuits and state assignment is one of the key steps in combinational logic optimization. The SMP-based parallel algorithm – based on the sequential program JEDI targeting multilevel logic implementation – scales nearly linearly with the number of processors for FSMs of varying problem sizes chosen from standard benchmark suites while attaining quality of results comparable to the best sequential algorithms. The source code for our parallel implementation of JEDI is freely-available from our web site hpc.ece.unm.edu.

*This work was supported in part by NSF Grants CAREER ACI-00-93039, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654.

†Supported in part by an NSF Research Experience for Undergraduates (REU) grant.

1 Introduction

Parallel architectures have promised high performance computing, but their use remains largely restricted to well structured numeric applications. Exploiting parallelism at the level of large distributed memory systems is hindered by the cost of message passing. However, Symmetric Multiprocessors (SMPs) with modest shared memory have emerged as an alternative platform for the design of scientific and engineering applications. SMP clusters are now ubiquitous for high-performance computer, consisting of clusters of multiprocessors nodes (e.g., IBM Regatta, Sun Fire, Compaq AlphaServer, and SGI Origin) interconnected with high-speed networks (e.g., vendor-supplied, or third party such as Myricom, Quadrics, and InfiniBand). Current research has shown that it is possible to design algorithms for irregular and discrete computations [6, 7, 4, 3] that provide efficient and scalable performance on SMPs.

With the rapid strides in VLSI technology, circuit design and analysis are becoming increasingly complex. There is a growing need for sophisticated CAD tools that can handle large problem sizes and quicken the synthesis, analysis, and verification steps in VLSI design. CAD applications are inherently unstructured and non-numerical, making them difficult to parallelize effectively. The state assignment problem for Finite State Machines is one such application. It can be formulated as an optimization problem that is NP-complete. Sequential heuristics that try to solve this task are computationally intensive and fail for large problem sizes. The parallel implementation discussed in the paper, which is based on the sequential algorithm JEDI [14], tries to overcome this difficulty and attain better results, i.e., designs with fewer literals and hence, faster circuits with reduced size and power consumption, as well as faster execution times for the design and analysis.

1.1 Finite State Machines

The Finite State Machine (FSM) is a device that allows simple and accurate design of sequential logic and control functions. Any large sequential circuit can be represented as an FSM for easier analysis. For example, the control units of various microprocessor chips can be modeled as FSMs. FSM concepts are also applied in the areas of pattern recognition, artificial intelligence, language and behavioral psychology.

1.2 FSM Representations

An FSM can be represented in several ways, the most common being the State Transition Graph (STG) and the State Transition Table (STT) representations.

The STG is described as $G = (S, E, I(E), O(E))$ where S is the set of all states, E is the set of all edges of the graph, and $I(E)$, $O(E)$, the input and output sets, respectively, corresponding to a particular edge. The STT can be described as $T = (I, P, N, O)$ where I, P, N, O are the Input, Present state, Next state, and Output

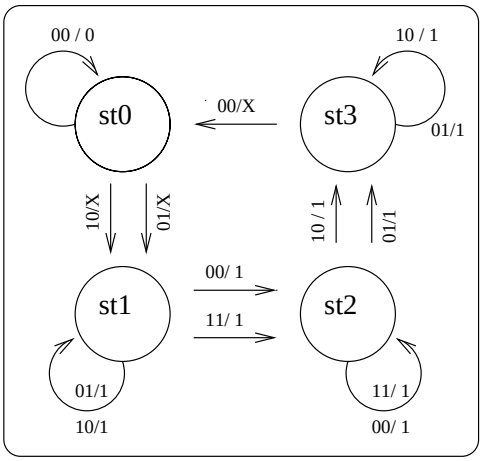

Figure 1: *train4* FSM - STG Representation

sets, respectively. A row in the STT corresponds to a unique edge in the STG, and the table has as many rows as the number of edges in the STG.

1.3 FSM Optimization - SIS

An FSM can be optimized for area, performance, power consumption, or testability. The various steps involved in optimizing an FSM are state minimization, state assignment, logic synthesis, and logic optimization.

SIS [17] is a popular tool for synthesis and optimization of sequential circuits. Many different programs and algorithms have been integrated into SIS, allowing the user a lot of choice at every step of the optimization process. The first step of optimization, state minimization is done using STAMINA [16]. Two state assignment programs, NOVA [19] and JEDI, are distributed with SIS. After state assignment, the resulting logic for the output can be minimized by the logic minimizer ESPRESSO [18] which tries to find a logic representation with the minimum literals while preserving the functionality of the FSM.

2 Problem Overview

The State assignment problem deals with assignment of unique binary codes to all the states of the FSM so that the resulting binary next-state and output functions can be implemented efficiently. Most of the algorithms optimize for the area of logic implementation. The number of literals in the factored form of logic is the accepted and standard estimate [8] for area requirements of a logic implementation. The parallel algorithm discussed here tries to minimize this measure and thus optimize for the area.

To illustrate the significance of state assignment, consider the following example.

The FSM *train4* (Fig. 1) consists of four symbolically encoded states *st0*, *st1*, *st2* and *st3*. These states can be assigned unique states by using a minimum of two bits, say y_1 and y_2. The input can be represented by two bits x_1 and x_2 and the the single bit output by z. y_{1ns} and y_{2ns} represent the next state bits of y_1 and y_2 respectively.
Suppose the following state assignment is done:
st0 – 00, *st1* – 01, *st2* – 11, *st3* – 10.
The resulting logic equations after *multilevel logic optimization* would be

$$\begin{aligned} y_{1ns} &= z(x_1y_1 + x_2y'_{2ns} + k) + x'_1x'_2y_{2ns} \\ y_{2ns} &= y'_1k'(x_1 + x_2) + y_2(x'_1x'_2 + k) \\ z &= y'_2(y'_1 + k) \\ k &= x_1x_2 \end{aligned}$$

This implementation results in a literal count of 22 after logic optimization and requires 15 gates to realize assuming the complements of inputs are present.
However, if the state assignment were
st0 – 00, *st1* – 10, *st2* – 01, *st3* – 11,
the logic equations after optimization would be

$$\begin{aligned} y_{1ns} &= z(y_1 + y'_{2ns}) \\ y_{2ns} &= x_1x'_2 + x'_1x_2 \\ z &= y_1y_2y'_{2ns} + y'_1y'_2 \end{aligned}$$

The literal count is only 12 in this case and 8 gates are sufficient to implement the logic.

Optimal State assignment becomes computationally complex as well as crucial for larger FSMs.

2.1 Previous Research

A lot of research has been done in the area of state assignment algorithms. KISS was one of the first algorithms proposed targeting a PLA based implementation. NOVA improved on KISS and was based on a graph embedding algorithm. Genetic algorithms [1] and algorithms based on FSM decomposition [2] were also proposed.

The sequential logic optimization tool SIS uses algebraic techniques to factor the logic equations. It minimizes the logic by identifying common subexpressions. The algorithms MUSTANG [9] and JEDI use this fact and try to maximize the size as well as the number of common subexpressions. These algorithms target a multilevel logic implementation and the number of literals in the combinational logic network after logic optimization is taken as the measure of the quality of the solution.

MUSTANG is a greedy heuristic which tries to encode pairs of clusters of states

with similar behavior closely so that the Boolean logic results in a large number of common subexpressions. The affinity between states is calculated using various weight assignment algorithms and the states are encoded such that the cost function

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

is minimized, where $we(s_i, s_j)$ is the edge weight calculated by the algorithm and $dist(enc(s_i), enc(s_j))$ denotes the Hamming distance between two codes s_i and s_j.

The sequential algorithm JEDI on which the parallel algorithm is based is discussed in detail in the next section.

The ProperCAD project [15] aims to develop portable parallel algorithms for VLSI CAD applications. Parallel algorithms for state assignment [11] based on MUSTANG and JEDI have also been proposed as part of this project. However, the speedups in case of shared memory implementations were not significant. The shared memory algorithm discussed here improves on this work as it attains better speedups without compromising the quality of results. It is also the fastest state assignment program proposed so far.

3 JEDI

JEDI is a state assignment algorithm targeting a multi-level logic implementation. The program involves two main stages, the weight assignment stage and the encoding stage.

3.1 Weight assignment

In this stage, JEDI assigns weights to all the edges in the STG, which is an estimate of the affinity of the states to each other. It chooses the best of four heuristics for this purpose: input dominant, output dominant, coupled, and variation.

The input dominant algorithm assigns higher weights to pairs of present states, which assert similar outputs and produce similar sets of next weights. It has the effect of maximizing the size of common cubes in the implemented logic function. The output dominant algorithm assigns higher weights to pairs of next states, which are generated by similar input combinations and similar sets of present stages. It has the effect of maximizing the number of common cubes in the logic function. The coupled algorithm adds up the weights generated by both the input and output dominant algorithms.

To illustrate the weight assignment algorithms, consider again the example of the FSM *train4* (Fig. 1). It has four states: *st0*, *st1*, *st2*, and *st3*. The input is two bit and the output is a single bit. Suppose the input dominant algorithm is applied. Let us determine the weight of the edge (*st0*, *st1*).

Firstly, for each state *st0* and *st1*, the states from which they fan out and their frequency are listed.

$$st0 \rightarrow (st0^1, st3^1), st1 \rightarrow (st0^2, st1^2)$$

Next, the inputs that lead to these states are identified and the corresponding sets are listed.

$$i_1(0) \rightarrow (st0^2, st1^3), \quad i_1(1) \rightarrow (st0^0, st1^1),$$
$$i_2(0) \rightarrow (st0^2, st1^1), \quad i_2(1) \rightarrow (st0^0, st1^3).$$

An *Input weight assignment matrix* M_I is constructed. It is a $N_s \times N_s$ symmetric matrix with each element m_{ij} corresponding to the weight of the edge (s_i, s_j).
In general, m_{ij} is defined by

$$m_{ij} = \sum_{a=1}^{O_i} \sum_{b=1}^{O_j} P(o_{ia}, o_{jb})$$

where O_i and O_j are the number of transitions out of states s_i and s_j of the STG and o_{ia} is the set of binary outputs produced by transition a out of state s_i.
In this case, the value of $m_{01} = m_{10} = 2 * 3 + 2 * 1 = 8$.
The weights can be computed similarly for the Output dominant algorithm.

3.2 Encoding phase – Simulated Annealing

JEDI tries to encode states with high edge weights closely. This step can be formulated as minimization of the sum

$$\sum_{i=1}^{N_s} \sum_{j=1}^{N_s} m_{ij} * dist(s_i, s_j)$$

where m_{ij} is the edge weight from the input assignment matrix and $dist(s_i, s_j)$ denotes the Hamming distance between two codes s_i and s_j when they are encoded using the minimum number of bits. The Hamming distance between two binary codes is the number of bits in which the codes differ. For example, the codes 1000 and 1110 have a Hamming distance of 2.

Encoding is done using Simulated annealing, a probabilistic hill climbing heuristic. The general algorithm is as follows:

1. Start with an initial temperature T and a random configuration of states.
2. For a given temperature, pick two states at random and assign new encodings or exchange their encodings.

3. Calculate the change in cost function.
4. Accept the exchange for a decrease in the cost function. Allow some encodings to be accepted even if it leads to an increase in the cost function in order to avoid local minima.
5. Repeat steps 2–4 until a certain number of moves are made. Then lower the temperature and continue with the process.
6. Stop when the temperature reaches the minimum temperature.

After the encoding and simulated annealing is done, the output can be written in a BLIF (Berkeley Logic Interchangeable Format) file. This file is then passed into the sequential logic synthesizer, and logic optimization is carried out using ESPRESSO. The required parameters such as the literal count and final output logic then can be retrieved for further analysis.

4 Parallel Implementation

JEDI is implemented in parallel using the SMP Node primitives of SIMPLE [5], a methodology for developing high performance programs on clusters of SMPs. Both the weight computation and the encoding stages have been parallelized.

The programming environment for SMPs is based upon the SMP node library component of SIMPLE, that provides a portable framework for developing SMP algorithms using the single-program multiple-data (SPMD) programming style. This framework is a software layer built from POSIX threads that allows the user to use either the already developed SMP primitives or the direct thread primitives. The SMP Node library contains a number of SMP node algorithms for barrier synchronization, broadcasting the location of a shared buffer, replication of a data buffer and memory management. In addition to these functions, there is a *parallel do* that schedules n independent work statements implicitly to p processors as evenly as possible.

4.1 Weight computation stage

The parallel algorithm for weight computation is detailed in Algorithm 1. To calculate the weight matrix, all the state pairs are inspected first. For state i, the edge weights of state pairs $(1, i)$ to $(i-1, i)$ are checked. The i states are distributed among the processors using the *pardo* primitive. Thus no two processors get the same edge pair and so there are no conflicts. Each processor updates the weight matrix independently. Since it is a shared memory implementation, no merging of local weight matrices is required.

```
Result : Weight computation of the states in parallel
compute the weight assignment matrix;
(the inner loop is executed in parallel:)
for i = 1 to N_s do
    for j = 1 to i − 1 in parallel do
        calculate the edge weight (s_i, s_j);
    end
end
synchronize;
```

Algorithm 1: Weight computation stage after the KISS format file is read into the appropriate data structure.

```
Result : Simulated Annealing by 'Parallel moves'
(initialization:)
1. Set the initial and stopping temperatures;
2. Set the cooling parameter;
3. Calculate the initial no. of moves per stage;
(annealing in parallel:)
while Current temp > Stopping Temp do
    for i = 1 to maxgen do
        (maxgen is the max. no. of moves for a temp. value);
        (the maxgen moves are divided among p processors);
        compute cost function;
        generate two random codes;
        check whether they are already assigned;
        if codes are assigned then
            exchange the codes of the two states;
        else
            assign new codes to two states;
        end
        calculate the additional cost;
        accept or reject the exchange;
        update the data structure;
    end
    synchronize;
    update the temperature value;
end
```

Algorithm 2: Simulated Annealing by 'parallel moves'

4.2 Encoding Stage – Simulated Annealing

The encoding stage involves assignment of unique binary codes to each state such that the literal count of the combinational logic is minimized. This is done using simulated annealing, which is a computationally intensive process. A lot of research has been done in parallelizing it for the placement problem in VLSI CAD applications [12].

The current implementation tries to implement simulated annealing using the *divide and conquer* strategy as well as the *parallel moves* technique. Previous research has shown that these techniques of parallel simulated are suited for shared memory multiprocessors [13].

A unique global configuration is maintained in *divide and conquer* and *parallel moves*, which makes it easy to implement on shared memory. The principle of parallel moves is to apply multiple moves to a single configuration simultaneously. On the other hand, the divide and conquer method lets processors make simultaneous moves within preset boundaries of the configuration. Both the techniques are detailed here.

4.3 Error and Quality control

The parallel simulated annealing stage has been implemented such that there are no conflicts. In the parallel moves method of annealing, each processor can choose moves independently from the entire set of available moves. When moves are being evaluated in parallel, it is important to control how the moves are to be accepted. Firstly, it must be ensured that moves in parallel are not contradictory. For example, two different codes must not be assigned to the same state in parallel. This case does not arise due to the global data structure and shared memory configuration.

In case of sequential simulated annealing, the cost function needs to be determined only once for a particular temperature, and then, only the change in the cost is evaluated for deciding whether to accept or reject a state. But in parallel, the cost has to be evaluated every time a decision has to be made, since the global data structure can be changed by any processor. This is an additional overhead in case of the parallel moves implementation. But this does not affect the quality of the annealing. However, it is possible that there is a repetition of the same moves by different processors leading to redundant moves.

In case of the divide and conquer strategy, a serious issue needs to be considered. If the local state space partitions are static, then the number of potential moves is reduced as the number of processors increase. This would lead to a degradation of quality, and the algorithm may not even converge. To avoid this, the states need to be shuffled and distributed in such a manner that it is possible to realize all possible moves in the global state space. This is ensured by changing the partitions for a change in temperature so that all the possible moves are probable. However, the probability that a move is realized decreases when compared to the sequential algorithm. This leads to a degradation in the quality for some runs.

```
Result : Simulated Annealing by 'Divide and Conquer'
(initialization:)
1. Set the initial and stopping temperatures;
2. Set the cooling parameter;
3. Calculate the initial no. of moves per stage;
4. Divide the state space into P partitions;
(annealing in parallel:)
while Current temp > Stopping Temp do
    for i = 1 to maxgen do
        (maxgen is the max. no. of moves for a temp. value);
        compute cost function in the local encoding space;
        generate two random codes;
        check whether they are already assigned;
        if states are assigned then
            exchange the codes of the two states;
        else
            assign new codes to states;
        end
        calculate the additional cost in the local space;
        accept or reject the exchange;
        update the local data structure;
    end
    synchronize;
    update the temperature value;
    calculate the cost function of the entire state space;
end
```

Algorithm 3: Simulated Annealing by 'divide and conquer'

To illustrate how the repartitioning is done, consider the following example. Suppose an FSM has 28 states. The minimum number of bits needed to encode all states are 5 and this gives a state space of size 32. Consider a four processor parallel annealing algorithm carried out in this space. Each processor is assigned 8 codes initially, processor 1 getting codes 0–7, processor 2 getting codes 8–15, and so on. If no repartitioning is done, then the exchange of states 1 & 8, 2 & 9, etc., is not possible. However, for the next temperature change, if the partitions are modified such that processor i gets codes $4k + i$, for $0 \leq k < 8$, (processor 0 getting $0, 4, 8, 12, \ldots, 28$, and so on), then all exchanges are theoretically possible. This partitioning scheme can be be extended for a general case, an N-bit encoding space and $P = 2^k$ processors.

5 Implementation and Results

A simplified version of JEDI, the sequential algorithm distributed with SIS, has been developed for comparison with the parallel version. This algorithm converges to the same solutions and preserves most of the features of the original algorithm. The FSMs used for testing are from the MCNC benchmark suite. After state assignment, the output in PLA file format is passed through MV-SIS [10], a multilevel multi-value logic synthesis tool from Berkeley. The quality of the solution is given by the literal count after multilevel logic optimization. Lower literal count implies less area of implementation. We use the SMP programming environment previously described and discuss next the multiprocessor environment used in this empirical study.

5.1 Parallel Computing environment

The shared-memory implementation has been tested on the Sun E4500, a uniform-memory-access (UMA) shared memory parallel machine with 14 UltraSPARC II 400MHz processors and 14 GB of memory. Each processor has 16 Kbytes of direct-mapped data (L1) cache and 4 Mbytes of external (L2) cache.

5.2 Experimental Results

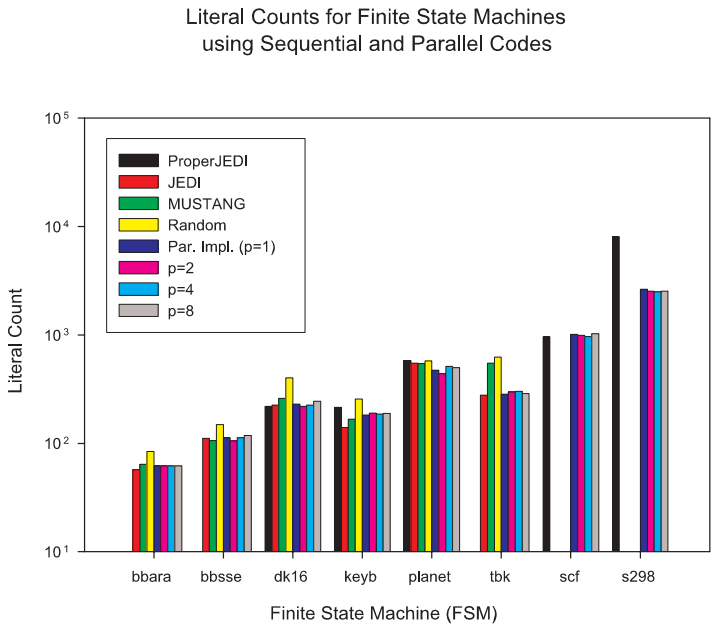

Figure 2: Literal Counts for Various Finite State Machines using Sequential and Parallel Codes. $p = 1$ is the parallel code run on a single processor; $p = 2, 4,$ and, 8, is the number of processors running the parallel code.

The quality of the solution is measured by the literal count after multilevel logic optimization (see Fig. 2). The literal counts of FSMs of varied problem sizes, when the parallel algorithm is run on one processor, are reported in Table 1 in the appendix. The count reported is the best value obtained among the four weight generation heuristics. The literal count reported by JEDI in [14] and the uniprocessor results obtained by the parallel implementation ProperJEDI [11] are also listed for comparison.

The quality of the solution for multiprocessor runs is reported in detail in Table 2 in the appendix. Simulated annealing on multiple processors leads to slightly varying results for successive runs of the same problem, so we report the best literal count obtained over five test runs. The results listed are the ones obtained from the parallel moves simulated annealing technique as they give better results than the divide and conquer technique.

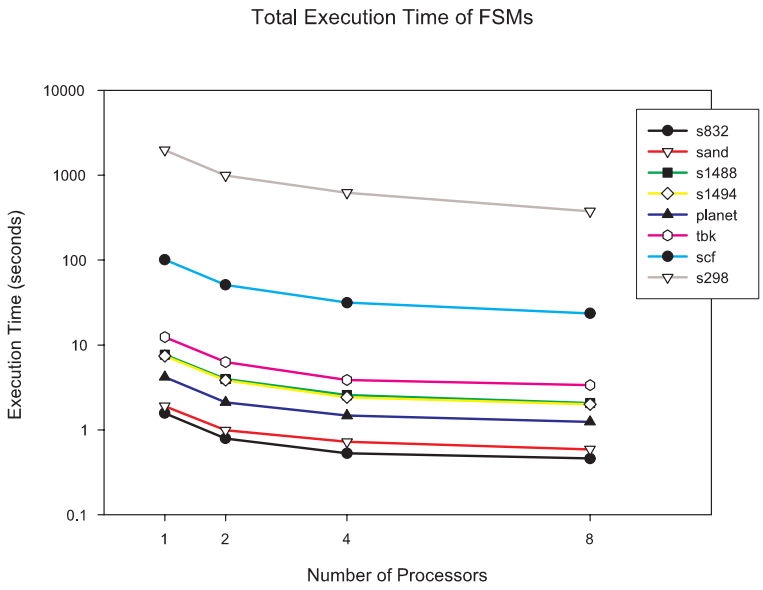

Figure 3: Total Execution time (in seconds) for a suite of Finite State Machines.

Figure 3 shows the total execution time for different FSMs on multiprocessor runs. (Table 3 in the appendix provides detailed running times.) The reported speedups are good for all problem sizes (number of states) but are higher for larger problem sizes.

Figure 4 shows the execution time and corresponding speedup for the weight computation and simulated annealing steps separately for two different FSMs. It is observed that the execution time for the weight computation phase scales nearly linearly with the number of processors for all problem sizes.

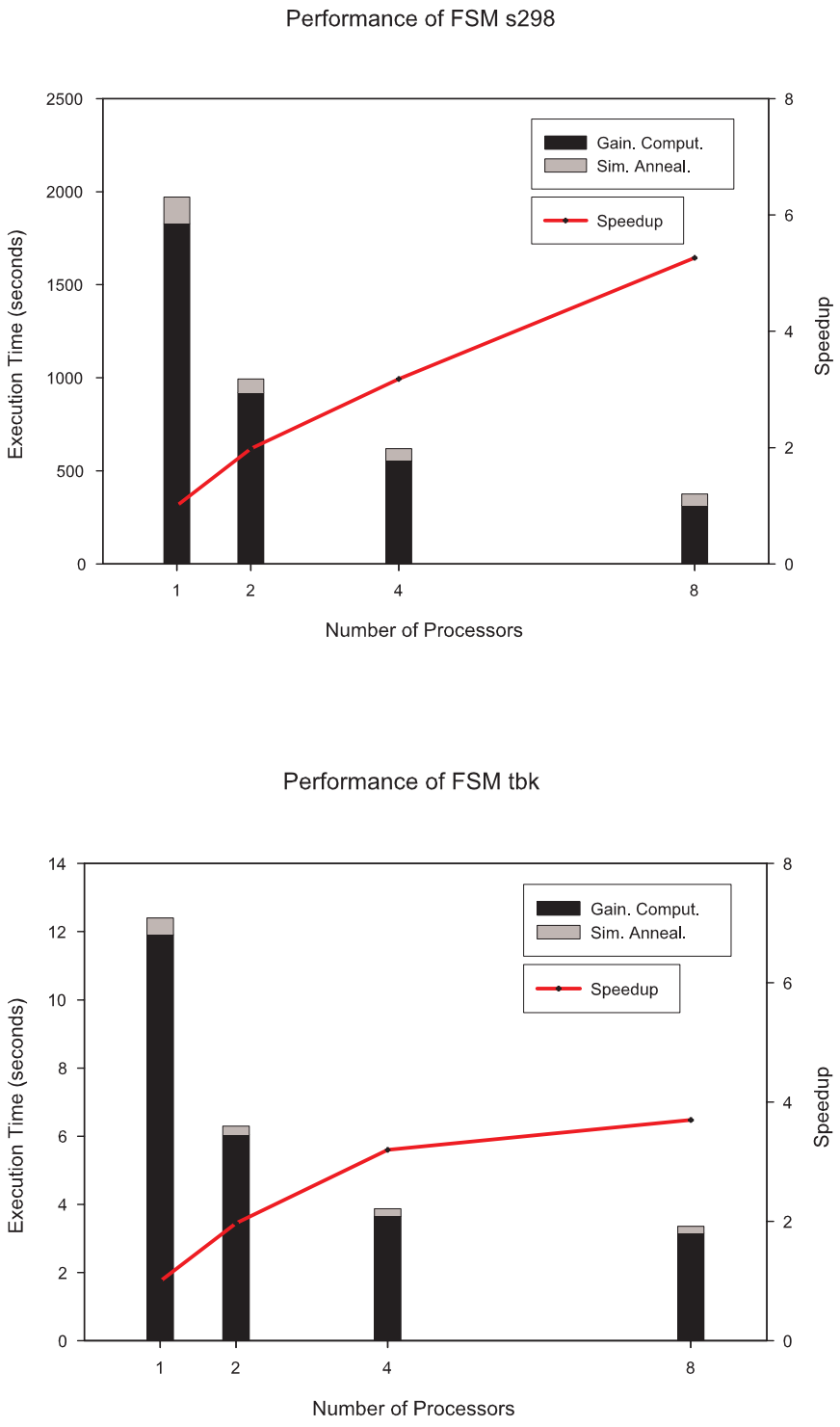

Figure 4: Performance for the weight computation and simulated annealing steps for two different Finite State Machines, s298 (top) and tbk (bottom).

6 Conclusions

A parallel implementation of the popular state assignment algorithm JEDI has been developed, specifically targeting shared memory multiprocessors. The present sequential algorithms fail to give good results for large Finite State machines. However with the parallel implementation, significant reductions in time and memory requirements can be achieved without compromising the quality of the solution. Better results can be obtained if parallel logic optimization tools are also developed.

References

[1] A.E.A. Almaini, J.F. Miller, P. Thomson, and S. Billina. State assignment of finite state machines using a genetic algorithm. *IEEE Proc. Computers and Digital Techniques*, 142(4):279–286, 1995.

[2] P. Ashar, S. Devadas, and A.R. Newton. A unified approach to the decomposition and re-decomposition of sequential machines. In *Proc. 27th ACM/IEEE Design Automation Conf.*, pages 601–606, Orlando, FL, June 1990.

[3] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. Technical Report 03-002, Electrical and Computer Engineering Department, University of New Mexico, July 2003.

[4] D. A. Bader and G. Cong. A new, fast parallel spanning tree algorithm for symmetric multiprocessors (SMPs). Technical Report 03-001, Electrical and Computer Engineering Department, University of New Mexico, March 2003.

[5] D. A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *J. Parallel & Distributed Comput.*, 58(1):92–108, 1999.

[6] D.A. Bader, A.K. Illendula, B. M.E. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In G.S. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 129–144, Århus, Denmark, 2001. Springer-Verlag.

[7] D.A. Bader, S. Sreshta, and N. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In S. Sahni, V.K. Prasanna, and U. Shukla, editors, *Proc. 9th Int'l Conf. on High Performance Computing (HiPC 2002)*, volume 2552 of *Lecture Notes in Computer Science*, pages 63–75, Bangalore, India, December 2002. Springer-Verlag.

[8] R.K. Brayton and C.T. McMullen. Synthesis and optimization of multistage logic. In *Proc. IEEE Int'l Conf. Computer Design (ICCD)*, pages 23–30, Portchester, NY, October 1984.

[9] S. Devadas, H.-K.T. Ma, A.R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State assignment of finite state machines for optimal multi-level logic implementations. *IEEE Trans. Computer-Aided Design*, 7(12):1290–1300, 1988.

[10] M. Gao, J.-H. Jiang, Y. Jiang, Y. Li, S. Sinha, and R. Brayton. MVSIS. In *Proc. Int'l Workshop on Logic Synthesis*, pages 138–144, Tahoe City, CA, June 2001.

[11] G. Hasteer and P. Banerjee. A parallel algorithm for state assignment in finite state machines. *IEEE Trans. Computers*, 47(2):242–246, 1998.

[12] S. Kim, J.A. Chandy, S. Parkes, B. Ramkumar, and P. Banerjee. ProperPLACE: A portable parallel algorithm for standard cell placement. In *Proc. 8th Int'l Parallel Processing Symp. (IPPS'94)*, pages 932–941, Cancún, Mexico, April 1994.

[13] S.A. Kravitz and R.A. Rutenbar. Placement by simulated annealing on a multiprocessor. *IEEE Trans. Computer-Aided Design*, 6(4):534–549, 1987.

[14] B. Lin and A.R. Newton. Synthesis of multiple level logic from symbolic high-level description language. In *Proc. of the IFIP TC 10/WG 10.5 Int'l Conf. on Very Large Scale Integration*, pages 414–417, Germany, August 1989.

[15] B. Ramkumar and P. Banerjee. ProperCAD: A portable object-oriented parallel environment for VLSI CAD. *IEEE Trans. Computer-Aided Design*, 13(7):829–842, 1994.

[16] J.-K. Rho, G.D. Hachtel, F. Somenzi, and R.M. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. *IEEE Trans. Computer-Aided Design*, 13(2):167–177, 1994.

[17] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A. Sangiovanni-Vincentelli. *SIS: A system for sequential circuit synthesis*. Electronics Research Laboratory, University of California, Berkeley, ucb/erl m92/41 edition, May 1992.

[18] M. Theobald, S.M. Nowick, and T. Wu. Espresso–HF: A heuristic hazard-free minimizer for two-level logic. In *Proc. 33rd ACM Design Automation Conf.*, pages 71–76, Las Vegas, NV, June 1996.

[19] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment of finite state machines for optimal two-level logic implementation. *IEEE Trans. Computer-Aided Design*, 9(9):905–924, 1990.

A Tables

FSM	Par. Impln.	ProperJEDI	JEDI	MUSTANG	Random
bbara	62	–	57	64	84
bbsse	112	–	111	106	149
dk16	230	219	225	259	402
keyb	182	215	140	167	256
planet	473	581	547	544	576
tbk	284	–	278	547	625
scf	1012	957	–	–	–
s298	2640	8034	–	–	–

Table 1: Literal Count for Single Processor Runs

FSM	1 proc.	2 proc.	4 proc.	8 proc.
bbara	62	62	62	62
bbsse	112	105	112	118
dk16	230	219	225	245
keyb	182	190	186	189
planet	473	438	512	498
tbk	284	298	302	288
scf	1012	988	964	1028
s298	2640	2530	2500	2530

Table 2: Literal Count of Parallel Implementation

FSM	1 proc.	2 proc.	4 proc.	8 proc.
s832	1.57	0.79	0.53	0.46
sand	1.9	0.99	0.72	0.59
s1488	7.64	3.97	2.56	2.07
s1494	7.43	3.85	2.42	1.99
planet	4.19	2.11	1.47	1.24
tbk	12.4	6.3	3.87	3.36
scf	100.8	51.04	31.44	23.56
s298	1971.03	992.95	619.4	375.44

Table 3: Total Execution time (in seconds)