Proceedings of the Supercomputing Conference, Reno, NV, November 1989.

EFFICIENT CIRCUIT PARTITIONING ALGORITHMS FOR PARALLEL LOGIC SIMULATION

Srinivas Patil *Prithviraj Banerjee*
Center for Reliable and
High Performance Computing
Coordinated Science Laboratory
University of Illinois
Urbana, IL-61801

Constantine D. Polychronopoulos
Center for Supercomputing
Research and Development
Talbot Laboratory
University of Illinois
Urbana, IL-61801

ABSTRACT

General purpose parallel processing machines are increasingly being used to speed up a variety of VLSI CAD applications. This paper addresses logic simulation on parallel machines by exploiting the concurrency in the circuit being simulated (called *data parallelism*) as opposed to exploiting parallelism inherent in the simulation algorithm itself (called *functional parallelism*). The most crucial step in obtaining the maximum parallelism using data parallelism is the partitioning of circuit elements. We introduce a cost function which tries to model the simulation of a logic circuit in a parallel environment. The cost function tries to estimate the parallel run time for logic simulation given the processor assignment and the underlying multiprocessor architecture. We then present different heuristic algorithms to partition the circuit and evaluate the efficiency of these algorithms using the proposed cost function. Partitioning algorithms for both event-driven and compiled code simulation are given.

1. INTRODUCTION

With general purpose parallel processing machines becoming more commonplace, parallel processing is being used extensively to solve a variety of VLSI CAD algorithms. Logic simulation forms an important part of many VLSI CAD applications. There are two ways in which we can exploit parallelism in logic simulation. We can use *functional partitioning* to exploit parallelism inherent in the simulation algorithm itself. But this type of parallelism is normally limited and does not scale very easily as the number of processors is increased. In this paper we concentrate on *data partitioning* for parallel logic simulation in which we exploit parallelism by partitioning the circuit being simulated.

A multicomputer network consists of many processing nodes connected together through bidirectional or unidirectional links. Each processor has its own local memory and processors interact with each other by passing messages over the communication links. Typically, the cost of communication over a link is an order of magnitude higher than the cost of accessing a processor's own local memory. Given a circuit graph where each node in the circuit graph is considered an atomic computation i.e., it cannot be split over two different processors, the objective is to arrive at a mapping of the circuit graph onto the processor interconnection graph so that the overall run time is minimized. This involves optimization of two objectives simultaneously: the concurrency and the communication. The mapping should be such that it maximizes concurrency and minimizes communication. The concurrency and communication requirements can be combined into a composite objective function which simplifies the optimization problem. Most optimal scheduling problems are NP-complete [1]. Because of this reason, heuristics are used to arrive at suboptimal solutions in a reasonable amount of CPU time.

In this paper, we present a computation model which tries to model the manner in which a circuit graph would be evaluated on a given multiprocessor with a given processor assignment. We then compute the estimated run time using this model which takes into account the communication delays. We use this estimated run time as an objective function for scheduling algorithms. We present scheduling algorithms based on different strategies: greedy allocation, allocation based on structure of the circuit graph, and allocation based on adaptive probabilistic heuristics, namely simulated annealing [2].

The main motivation for this work was circuit partitioning for parallel logic simulation where each circuit block can be considered as task and the whole circuit can be considered as a computation graph. The edges connecting two circuit blocks define the dependence relationships among the blocks for computation. We present the results of using the scheduling heuristics on circuits which are used as benchmarks for test generation algorithms in VLSI. A comparison of results from different heuristics is given. The results presented are completely general and can be applied to other graph partitioning problems for multiprocessing as well.

2. PREVIOUS WORK

Different heuristics have been proposed in the literature for partitioning graphs. The popular Kernighan-Lin heuristic proposed in [3] tries a *max-flow min-cut* approach. The objective here is to partition n nodes in a graph into two classes A and B such that A and B contain a given number of nodes n_A and n_B and the total cost of edges running between A and B is minimized. The algorithm has a polynomial time complexity and produces a close to optimal solution. The algorithm can be applied for more than two partitions in a recursive manner. But this particular strategy works only if the tasks themselves do not have any weight associated with them i.e., the tasks are homogeneous and take the same time to execute. The solution does not always lead to a mapping with a minimal run time. In practice, it is not known beforehand as to how many tasks should be included in each partition for optimal execution. Also, the partitioning strategy assumes that minimization of communication overhead is the prime objective. The partitioning strategy can be modified to include the host architecture communication requirements.

An implementation of the *max-flow min-cut* approach for a parallel branch and bound algorithm on a hypercube was reported in [4]. A set of bounding functions were derived which help to prune the search space. The parallel branch and bound approach generates an *optimal* solution starting from the solution produced by the Kernighan-Lin heuristic as an initial solution. However, this approach suffers form the same drawback as that of the Kernighan-Lin approach.

One most commonly used objective function in optimization problems is the one used for quadratic assignment problems (cell placement, etc.) [5]. In terms of the mapping (or graph partitioning) problem, it can be stated as follows. Consider a task graph G_T which has to be mapped on to a multiprocessor defined by the topology G_P (also referred to as the host architecture). The distance between processors p_i and p_j is given by $d_{i,j}$. The communication requirement between two tasks T_i and T_j is given by m_{ij}. A task T_i is assigned to processor $p(T_i)$. Then the communication overhead due to a mapping of G_T onto G_P is given by:

$$\sum m_{ij} * d_{p(T_i),p(T_j)}$$

The above objective function can be used to arrive at a mapping that reduces the communication requirements under certain conditions. But as is apparent, the objective function does not take care of the concurrency requirements and may not necessary result in the least overall run time when the parallel task is run on a multiprocessor. Also, bottlenecks can occur due to high traffic densities on a particular network link.

Bokhari [6] studied the assignment of tasks to processors under the restriction that the number of tasks should be less than or equal to the number of processors. The cardinality of the mapping was used as an objective function for assignment. The objective here is to maximize the number of pairs of communicating tasks which directly fall on pairs of directly connected processors. Thus, the number of edges of G_T which fall on the edges of G_P are maximized. Uniform traffic is assumed and results similar to the quadratic assignment problem are obtained. Bianchini and Shen [7] have studied task assignment in the case of specialized architectures e.g., digital signal processing architectures. Lee and Aggarwal [8] assume a fixed routing strategy to minimize an objective function which quantifies the communication requirements accurately. However, the work still lacks an objective function which models both the communication and concurrency requirements accurately. The work by Kim and Browne [9] uses *linear clustering* as a heuristic to minimize the total execution time. A linear cluster is a connected subgraph of a computation graph which is in the form of a linear list of schedulable units of computation. Linear clustering is used to arrive at a compromise between the two conflicting goals: minimization of communication and maximization of concurrency. The restriction of linear clustering is that the computation graph be acyclic. A static task allocation scheme based on minimizing the overall run time was given in [10]. A number of different optimal as well as heuristic partitioning schemes for parallel execution of program graphs was proposed in [11].

More recently, algorithms have been proposed which use *simulated annealing* [2] as an optimization method. Bollinger and Midkiff proposed a simulated annealing based method which uses an objective function which tries to minimize the communication while at the same time does a link assignment to reduce the potential bottlenecks due to high traffic density [12]. However, this particular algorithm also does not focus on the minimization the overall run time. The work described in [13] uses simulated annealing to optimize a composite objective function which includes: (1) distance between communicating processes, (2) the memory utilization, (3) the I/O requirements on each node and (4) the total utilization of physical channel I/O capacity.

Parallel logic simulation for VLSI design has received wide attention in literature [14-17]. Circuit partitioning strategies specifically for logic simulation have been reported in the past. A method called partitioning by *element strings* was proposed in [18] which tries to assign linear strings of logic blocks as single entity to different processors. Several approaches to circuit partitioning were evaluated in [19] by using two different metrics: one which measures the concurrency and another which measures the communication. The partitioning strategies were finally evaluated by actually counting the number of messages in an actual event-driven simulator. An analysis of parallel logic simulation on several architectures was presented in [20] by assuming different models for different architectures. Most of these algorithms are structure based and do not take the underlying multiprocessor architecture into account.

In this paper we present a composite objective function which models the evaluation of a circuit given by the circuit graph G_C on a multiprocessor whose topology is given by G_P. The objective function takes care of the conflicting goals of maximizing concurrency vs. minimizing communication. Different heuristic algorithms for circuit partitioning are presented based on this cost function. The heuristic algorithms presented try to minimize the overall run time (including the communication time). Results are presented using practical benchmark circuits as circuit graphs and using the hypercube topology as the multiprocessor architecture.

3. COMPUTATIONAL MODEL AND COST FUNCTION

An example circuit is shown in Figure 1 with its graph representation. The circuit graph is denoted as a directed graph G_C with the vertex set $G_C(V)$ and the edge set $G_C(E)$. An edge $e_{ij}\ \varepsilon\ G_C(E)$ means that node C_i is an input to node C_j, where $C_i, C_j\ \varepsilon\ G_C(V)$. Each edge e_{ij} in G_C has a weight m_{ij} associated with it which gives the expected number of messages which would be passed between C_i and C_j. Also each node C_i has a label t_i associated with it which gives the expected time for evaluation of the node. A node in the circuit graph with no predecessors is called a *primary input node* and a node with no successors is called a *primary output node*. The current assignment of a node C_i to some processor is given by $p(C_i)$.

The host architecture graph (i.e., the multiprocessor topology) is represented as an undirected graph G_P with the vertex set $G_P(V)$ and the edge set $G_P(E)$. Each node corresponds to a processor p_i in the network and an edge e_{ij} exists between p_i and p_j if there exists some path connecting the two. Each edge e_{ij} has a weight d_{ij} associated with it which is an estimate of the minimum distance between processors p_i and p_j. The various symbols used in this paper are summarized below.

G_C	Circuit graph
G_P	Processor architecture
t_i	evaluation time for node C_i
m_{ij}	Size of message passed by node C_i to C_j
$p(C_i)$	Processor assignment for node C_i
d_{ij}	Distance between processor p_i and p_j
$eval\_time(C_i)$	Time when evaluation of node C_i completed

We give an example of how the cost function is calculated. Consider the simple circuit graph and the architecture graph given in Figure 1. The circuit graph consists of 6 nodes C_i, i=1....6. Let t_i=4 for all i and $m_{1,3}$=$m_{1,4}$=$m_{2,3}$=$m_{2,5}$=$m_{4,6}$=$m_{5,6}$=1. C_1 and C_2 are input nodes (no predecessors) and C_6 is an output node (no successors). If we ignore the costs (or labels) on the edges, then the critical path through the circuit graph has the total cost = 3∗4 = 12. This gives the lower bound on the execution time for the circuit given by the circuit graph on *any* number of processors. If all four nodes are allocated to the same processor, then it takes 18 time units for the evaluation and the theoretical upper bound on the speedup possible by using an infinite number of processors is 18/12 = 1.5. Hence, the theoretical lower bound on the execution time for the computation using an infinite number of processors is just the path of maximum length through the circuit graph where the length is calculated by adding up all t_i's along the path and ignoring all m_{ij}'s (ideally, the communication overhead is nil between two tasks if they are assigned to the same processor). All algorithms we are going to present assume that the circuit graph is levelized. The levelization algorithm is given in Figure 2. The *lower_bound* algorithm given in Figure 3 gives the lower bound estimate on the evaluation time for the circuit. The uniprocessor execution time is found by adding up all the t_i's in the network. The theoretical limit on the speedup is found by dividing the uniprocessor run time by the lower bound.

The proposed cost function is based on the example given above. First, the circuit graph is levelized using the levelization algorithm given in Figure 2. Since the levelization algorithm assumes that the computation graph is acyclic, cycles are detected and broken. The level of a node C_j in the computation graph is given by $level(C_j)$. All input nodes are at level 1. The cost function is based on one fundamental observation. Two nodes C_i and C_j can be evaluated in parallel if $level(C_i)=level(C_j)$. Hence, if we apply the levelizing algorithm to the circuit graph, the nodes with the same level number can be executed in parallel. If two nodes C_i and C_j are at the same level but are assigned to the same processor, then they must be executed one after the other in some particular order while if they are assigned to different processors they can be executed in parallel. Let $eval\_time(C_i)$ be the time at which node C_i is evaluated. Hence, if two tasks at the same level are assigned to the same processor, then there is an implicit sequential dependency between the two tasks. The algorithm which is used to compute the cost function takes care of such dependencies. Let $level\_time(i, p_j)$ be the time at which processor p_j evaluates all gates at $level(i)$ assigned to it. Assume all tasks at $level(i)$ assigned to processor p_j are assigned some ordering. The estimated run time strongly depends on the ordering so it is assumed the ordering is kept fixed for all evaluations of the cost function. The list of nodes at $level(i)$ for processor p_j is traversed and the time of completion of task t_k is estimated on processor p_j. The algorithm given in Figure 4 computes the estimated run time for the given task on the host architecture for the current configuration (i.e., the current assignment) and returns it as the cost. This cost can be used by any optimization algorithm. The logic behind the algorithm is very simple. The nodes in a circuit graph are traversed in level order for each processor. Let L_j be the list of nodes assigned to p_j and arranged in nondecreasing level order. We start with a dummy node C_0 whose evaluation time C_0 is zero. Let C_j be the node whose evaluation time has been already calculated. Let C_k be its successor in the ordered list L_j. Let C_{ki} be the predecessors of C_k in the circuit graph. The evaluation time of C_k is calculated as follows:

$$eval\_time(C_k) = t_k + max(C_j, pred\_time)$$

where

$$pred\_time = max(d_{p(C_k), p(C_{ki})} * m_{k,ki} + eval\_time(C_{ki}))$$

4. ALGORITHMS FOR CIRCUIT PARTITIONING

All algorithms described briefly below follow the computation model given in the previous section and use the same cost function. Let n be the number of nodes in the circuit graph and p be the number of available processors. All the algorithms assume that the graph is levelized using the algorithm given in Figure 2.

4.1. Random Partitioning

Each node C_i is assigned a randomly chosen processor. It takes $O(n)$ time for the algorithm to complete where n is the number of nodes in the circuit graph. The cost function of Figure 4 is then used to calculate the cost or 'goodness' of the assignment. This assignment is just used to compare with other partitioning methods or as an initial solution for simulated annealing.

4.2. Input/Output Partitioning

The task graph is traversed in a depth-first manner from the inputs and $\left\lceil \frac{n}{p} \right\rceil$ or $\left\lfloor \frac{n}{p} \right\rfloor$ tasks are assigned to each processor. The logic in this partitioning scheme is that strongly connected components of the task graph are assigned to the same processor so as to minimize communication. However, loads on different processors may not be balanced since the time t_i to execute a task T_i is not taken into account. A similar partitioning strategy can be implemented by traversing the computation graph in a depth-first manner from the output nodes. This called output partitioning. This algorithm takes $O(n)$ time.

4.3. Greedy Heuristic

The graph is first levelized using the algorithm given in Figure 2. Then we start with a null circuit graph G'_C. Let L be the list of nodes in graph G_C in nondecreasing level order. The graph G'_C is gradually transformed into G_C by adding nodes from G_C such that the total cost of the resulting graph is minimized. The greedy algorithm is given in Figure 5. Hence, the graph G'_C is gradually transformed into G_C by adding a node C_i in level order to G'_C and assigning C_i to a processor such that the cost of $G'_C \cup C_i$ is minimized. The cost function given in Figure 4 is used but a procedure called *compute_delta_cost* is used which calculates the incremental cost due to the addition of C_i. The procedure *compute_delta_cost* takes a time proportional to the number of predecessors of a task C_i which can be taken to be constant for most cases. The greedy algorithm given in Figure 5 takes $O(p*n)$ time to execute and is reasonably efficient.

4.4. Simulated Annealing

Simulated annealing is a general heuristic algorithm which has been used widely to get sub optimal solutions to NP-complete combinatorial optimization problems [2]. The general structure of a simulated annealing algorithm is shown in Figure 6. The different variables and functions in the algorithm are described below:

S
: The current solution to the optimization problem. S_0 is the initial configuration which may be generated using a random selection heuristic or some other heuristic algorithm. S' is a new configuration generated from the current solution using *perturb*.

perturb
: This routine generates new configurations from the current configuration. Various strategies which can be used to generate new configurations will be discussed later.

random
: Is a random number generated usually from a uniform distribution in the interval [0,1]. $e^{(Cost(S')-Cost(S))/T}$ is the exponential acceptance function, where $Cost(S)$ is the cost of the current configuration S and T is a heuristic parameter called "temperature". All configurations which reduce the cost are *always* accepted while those which increase the cost are accepted with a probability determined by the exponential acceptance function.

The simulated annealing algorithm given in Figure 6 is used with cost function given in Figure 4. Since the cost function has to be executed many times before a solution is reached, a slightly modified form of the algorithm given in Figure 4 is used so that only those parts of the graph G_C have to be evaluated which are changed by the *perturb* function. The implementation of the *perturb* function is simple. A task C_i is selected at random from G_C and assigned to a processor selected at random. The initial temperature T is the cost of a random initial assignment. In our application, the values of the parameters α and β for simulated annealing were chosen to be 0.95 and 1.10 respectively. The initial value of the number of iterations of the inner loop in Figure 6 was chosen to be 100 (parameter i_0). The number of moves allowed for simulated annealing was $n * p$ where n is the number of nodes in the circuit graph and p is the number of processors.

5. RESULTS

The ISCAS combinational benchmark circuits [21] normally used as benchmark circuits for test generation and logic simulation, were used as random circuit graphs to evaluate the partitioning algorithms given above. The circuit graphs contain approximately 400 to 7500 nodes and are denoted as *cNNNN* where *NNNN* is the number of nodes in the circuit graph. The multiprocessor architecture was assumed to be a hypercube architecture. Two sets of experiments were performed, one set for compiled code simulation and another set for event driven simulation.

5.1. Compiled Code Simulation

In compiled code simulation, all nodes in a circuit graph are evaluated irrespective of whether they are affected by the current inputs. Each node in the circuit graph is implemented as a separate procedure which can be evaluated. Since the whole circuit graph is written as a set of procedure calls, compiler optimizations may be used to speed up evaluation even further. Also, the list manipulation overheads of event-driven simulation are absent. Tables I and II show the results for the ISCAS combinational benchmark circuits assuming compiled code simulation. Each node was assumed to have a computation cost of 100 time units. Table I shows the detailed results for three of the benchmark circuits assuming a hypercube architecture with 16 processors. The communication/computation ratio was varied from 0.25 to 10 and the estimated speedups and the time taken for the partitioning algorithms to complete (in seconds on a SUN 3/50 workstation) are shown. The speedup was calculated as the ratio of the estimated run time on the uniprocessor (sum of all the node evaluation times) to the estimated run time on 16 processors given by the cost function. It can be seen that the greedy algorithm outperforms all other algorithms and takes a moderate amount of CPU time. It can be seen that other heuristics produce a speedup of less than 1 for high communication/computation ratios while the greedy heuristic always produces a speedup of greater than 1 due to its adaptive nature. Also, the time taken by simulated annealing becomes prohibitive as the circuit size increases. The quality of the solution using simulated annealing is always worse than that for the greedy algorithm. Table II shows the results for seven of the remaining ISCAS circuits when the communication to computation ratio was assumed to be 2.5. The results for simulated annealing are not shown since the CPU times were extremely high. It can be seen that the greedy algorithm still outperforms all other heuristics.

5.2. Event Driven Simulation

In event driven simulation, only those parts of the network are simulated which are affected by the present input change. Hence, the probability that a particular node is evaluated or the probability that a message is passed between two nodes is not always 1 and is dependent on the input vectors applied to the network. To get a rough estimate of these probabilities, each ISCAS circuit was simulated with a set of 1000 random patterns and the number of times a node was evaluated and the number of times a signal change was propagated along a particular line of a network was calculated. Dividing these numbers by the total number of random vectors (1000) gives us an estimate of the probabilities discussed above. In a real application, these probabilities can be calculated using the past history of the simulation and can be used for dynamic load reallocation as the simulation parameters change. The quantities used in compiled code simulation were then multiplied by these probabilities to get the corresponding quantities for event-driven simulation. The corresponding results for event-driven simulation are shown in Tables III and IV. Table III shows the detailed results for three of the ISCAS benchmark circuits when communication/computation ratio was varied from 0.25 to 10. It can be seen that the greedy algorithm outperforms all other algorithms in this case also. Table IV shows the summary results for the remaining 7 circuits where the communication to computation ratio was assumed to be 2.5.

Figure 7 shows the estimated speedup curves using the greedy heuristic for c880 as the number of processors are varied. The processors are assumed to be connected in a hypercube configuration. Since heuristics like input and output partitioning assume a prior knowledge of the number of nodes to be allocated to each processor (each processor gets either $\left\lceil \frac{n}{p} \right\rceil$ or $\left\lfloor \frac{n}{p} \right\rfloor$ nodes), as the number of processors is increased, the communication time becomes dominant due to the nodes being assigned to processors which are far apart which produces a "speed downs" instead of speedups as the number of processors is increased. However, since the greedy heuristic does not make any such assumptions, it produces speedups (estimated) which are monotonically increasing with the number of processors. Using the greedy heuristic, it is also possible to estimate the number of processors $p_{\max}$ after which there is very little likelihood of gaining additional speedups for a logic circuit. This can be done in the following manner:

(1) Increase the number of processors till the greedy algorithm gives an assignment where some of the processors are not allocated any nodes.

(2) The total number of processors less the number of processors which were allocated zero nodes gives an estimate of $p_{\max}$.

$p_{\max}$ can also be found from the speedup curve for a circuit graph by noting the starting point of the flat portion of the curve. For c880, it was found that the value of $p_{\max}$ was approximately 30 for compiled code simulation.

6. CONCLUSIONS

A computation model was proposed which allows us to calculate the estimated run time for simulation of a logic circuit on any multiprocessor architecture. Also, heuristics were proposed which use this estimated run time as a cost function to find a circuit graph partitioning which minimizes the estimated run time. An efficient greedy heuristic was proposed which consumes time proportional to $p * n$ where p is the number of processors in the network and n is the number of tasks in the computation graph and produces costs close to the optimal. It was shown how the heuristics could be used in both compiled code and event driven environments. The heuristics can also be used for dynamic load reallocation using run time data. The computation model can be extended to include cases where the processors in the multiprocessing system are non-homogeneous i.e., with different computing power or specialized functions like I/O or different amount of memory. Finally, the heuristics were evaluated on some example circuit graphs using the hypercube architecture as the host architecture and it was found that general purpose heuristics like simulated annealing perform much worse, both in terms of CPU time and the quality of the solution compared to the greedy heuristic.

ACKNOWLEDGEMENTS

This research was supported in part by the Joint Services Research Program under Contract N00014-84-C-0149, in part by the Semiconductor Research Corporation under Contract SRC 87 DP-109 and in part by the National Science Foundation under grant NSF CCR 87-05240.

REFERENCES

[1] M. R. Garey and D. S. Johnson, in *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York.

[2] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, ‘‘Optimization by simulated annealing,’’ *Science*, vol. 220, pp. 671-680, May 1983.

[3] B. W. Kernighan and S. Lin, ‘‘An efficient heuristic procedure for partitioning graphs,’’ *The Bell Sys. Tech. Journal*, pp. 291-307, February 1970.

[4] J. Clausen and J. L. Traff, ‘‘Parallel graph partitioning using branch-and-bound with dynamic distribution of subproblems,’’ Dept. of Computer Science, University of Copenhagen, Copenhagen, Denmark, Technical Report No. 88/13 ISSN. 0107-8283.

[5] M. Hanan and J. M. Kurtzberg, ‘‘A review of the placement and quadratic assignment problems,’’ *SIAM Review*, vol. 14, pp. 324-342, April 1972.

[6] S. H. Bokhari, ‘‘On the mapping problem,’’ *IEEE Trans. Computers*, vol. C-30, pp. 207-214, March 1981.

[7] R. P. Bianchini and J. P. Shen, ‘‘Interprocessor traffic scheduling algorithm for multiple-processor networks,’’ *IEEE Trans. Computers*, vol. C-36, pp. 396-409, April 1987.

[8] S. Y. Lee and J. K. Aggarwal, ‘‘A mapping strategy for parallel processing,’’ *IEEE Trans. Computers*, vol. C-36, pp. 433-442, April 1987.

[9] S. J. Kim and J. C. Browne, ‘‘A general approach to mapping of parallel computations upon multiprocessor architectures,’’ *Proc. Int. Conf. Parallel Processing*, pp. 1-8, June 1988.

[10] J. J. Baxter and J. Patel, ‘‘The LAST algorithm: A heuristic-based static task allocation algorithm,’’ *Proc. Int. Conf. Parallel Processing*, August 1989.

[11] M. Girkar and C. Polychronopoulos, ‘‘Partitioning program graphs for parallel execution,’’ *Proc. 1988 ACM Int. Conf. Supercomputing*, pp. 216-229, July 1988.

[12] S. W. Bollinger and S. F. Midkiff, ‘‘Processor and link assignment in multicomputers using simulated annealing,’’ *Proc. Int. Conf. Parallel Processing*, pp. 1-7, June 1988.

[13] C. S. Steele, ‘‘Placement of communicating processes on multiprocessor networks,’’ M. S. Thesis, Technical Report 5184:TR:85, Caltech, April 1985.

[14] G. F. Pfister, ‘‘The Yorktown Simulation Engine: Introduction,’’ *Proc. 19th Design Automation Conf.*, pp. 51-54, 1982.

[15] K. Wong and M. A. Franklin, ‘‘Performance analysis and design of a logic simulation machine,’’ *Proc. 14th Int. Symp. Computer Architecture*, pp. 46-55, 1987.

[16] R. J. Smith, ‘‘Fundamentals of parallel logic simulation,’’ *Proc. 23rd Design Automation Conf.*, pp. 2-12, 1986.

[17] L. Soule and T. Blank, ‘‘Parallel logic simulation on general purpose machines,’’ *Proc. 25th Design Automation Conf.*, pp. 166-171, 1988.

[18] Y. H. Levendel, P. R. Menon, and S. H. Patel, ‘‘Special-purpose computer for logic simulation using distributed processing,’’ *The Bell Sys. Tech. J.*, vol. 61, pp. 2873-2907, December 1982.

[19] S. P. Smith, W. Underwood, and M. R. Mercer, ‘‘An analysis of several approaches to circuit partitioning for parallel logic simulation,’’ *Proc. Int. Conf. Computer Design*, pp. 664-667, 1987.

[20] S. P. Smith, W. Underwood, and J. Newman, ‘‘An analysis of parallel logic simulation on several architectures,’’ *Proc. Int. Conf. Parallel Processing*, pp. 65-68, 1988.

[21] F. Brglez and H. Fujiwara, ‘‘Neutral netlist of ten combinational benchmark circuits and a target translator in FORTRAN,’’ *Special session on ATPG and fault simulation, Proc. IEEE Int. Symp. Circuits and Systems*, June 1985.

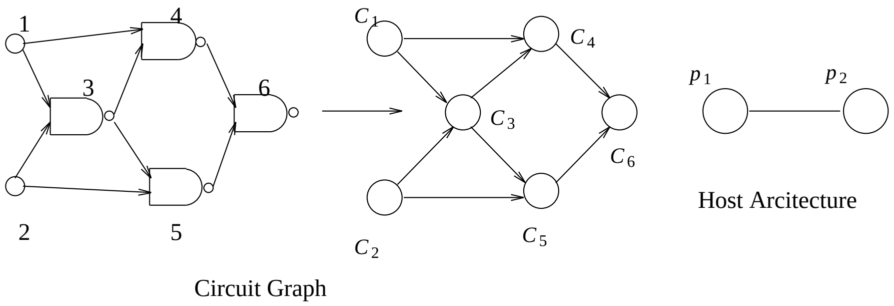

Figure 1. Example circuit and graph representation

```
procedure levelize;
    for all input nodes C_j do
        level(C_j) = 1;
    end for;
    L = list of nodes not yet assigned a level;
    while L not empty do
        for each node C_j in L do
            if all predecessor nodes of C_j have been assigned a level then
                level(C_j) = max(levels of predecessors of C_j) + 1;
                delete C_j from L;
            end if;
        end for;
    end while;
end levelize;
```

Figure 2. Levelizing procedure

```
procedure lower_bound;
    lower_bound = 0;
    for i ← 1 to maximum_number_of_levels do
        for each task C sub j at level i do
            eval_time(C_j) = t_j + max( eval_time for all predecessors of C_j);
            lower_bound = max(lower_bound, eval_time(C_j));
        end for;
    end for;
end lower_bound;
```

Figure 3. Lower bound calculation

```
procedure compute_cost;
    for i ← 1 to p do
        L_i is the list of nodes assigned to p_i;
        prev ← C_0
        for each C_j do
            pred_time = 0;
            for each predecessor C_jk of C_j do
                pred_time = max( d_{p(C_j),p(C_jk)} * m_{j,jk} + eval_time(C_jk);
            end for;
            eval_time(C_j) = C_j + max(eval_time(prev), pred_time);
            prev = C_j;
        end for;
    end for;
    return(max(eval_time(C_j))); /* return the maximum of the evaluation
                                     time over all the tasks T_j */
end compute_cost;
```

Figure 4. Calculating the cost function

```
procedure greedy_heuristic;
    G´_C ← nil;
    L ← List of nodes in G_C in nondecreasing level order;
    for for each node C_i in L do
        G´_C ← G´_C ∪ C_i;
        p(C_i) = 1;
        min_cost = compute_delta_cost(G´_C);
        min_proc = 1;
        for j ← 2 to p do
            p(C_i) = j;
            cost = compute_delta_cost(G´_C);
            if cost < min_cost then
                min_cost = cost;
                min_proc = j;
            end if;
            p(C_i) = min_proc;
        end for;
    end for;
end greedy_heuristic;
```

Figure 5. Greedy Heuristic

```
procedure Simulated_annealing:
    /*general form of simulated annealing */
    S ← S_0; /* initial solution */
    T ← T_0; /* initial temperature */
    iter ← i_0; /* number of iterations */
    repeat until time exceeded
        repeat iter times
            S´ ← perturb(S); /* generate a new configuration */
            if (Cost(S´) < Cost(S)) or (random < e^{(Cost(S´)-Cost(S))/T})
            then S ← S´; /* accept solution */
            else reject solution;
        end repeat;
        T ← α*T; /* lower the temperature */
        iter ← β*iter; /* increase the iterations in the inner loop */
    end repeat;
end Simulated_annealing;
```

Figure 6. General structure of a simulated annealing algorithm

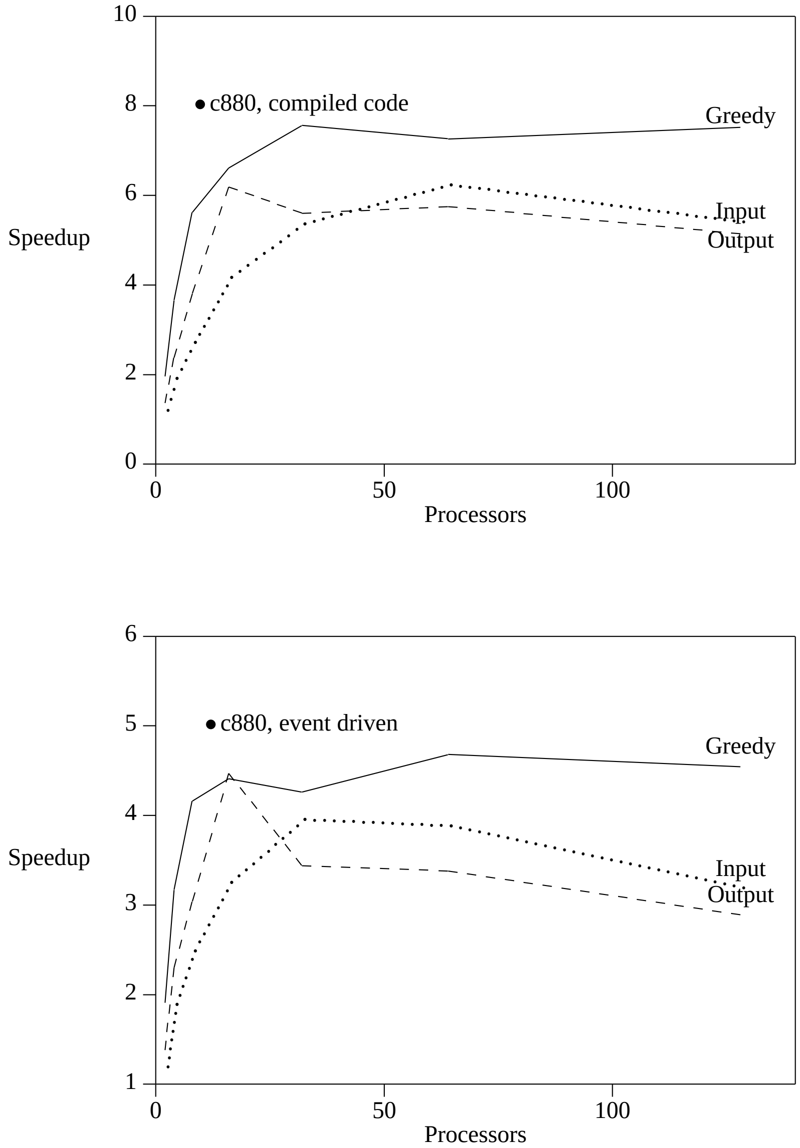

Figure 7. Speedup curves for c880

Table I. Detailed results for 3 circuits using compiled code simulation, 16 processors

Circuit	Ratio	Random		Input		Output		Greedy		Annealing	
		S_p	Time	S_p	Time	S_p	Time	S_p	Time	S_p	Time
c432	0.25	5.62	0.12	3.25	0.06	4.65	0.06	8.66	2.68	7.80	278.42
	1.00	3.36	0.06	2.83	0.06	3.88	0.06	5.74	2.32	4.89	273.90
	2.50	1.62	0.06	2.22	0.08	2.80	0.06	3.83	2.88	2.68	278.28
	5.00	0.98	0.04	1.61	0.06	1.85	0.06	2.71	2.32	1.57	259.72
	7.50	0.62	0.06	1.24	0.06	1.37	0.06	2.11	2.42	1.05	277.08
	10.00	0.46	0.06	0.99	0.06	1.07	0.06	1.85	2.36	0.84	271.74
c499	0.25	6.86	0.06	5.06	0.06	6.23	0.04	12.55	2.84	9.80	318.64
	1.00	5.52	0.04	4.63	0.06	5.65	0.06	9.16	2.90	6.74	306.40
	2.50	3.06	0.06	3.85	0.06	4.58	0.06	6.11	2.80	3.43	304.16
	5.00	1.66	0.06	2.73	0.06	3.27	0.06	3.88	2.76	2.35	305.84
	7.50	1.12	0.06	2.07	0.06	2.51	0.06	3.38	2.74	1.79	306.14
	10.00	0.91	0.06	1.66	0.06	2.04	0.06	2.54	2.76	1.11	305.46
c880	0.25	7.74	0.12	4.83	0.12	7.90	0.10	12.34	5.72	10.79	1014.34
	1.00	5.09	0.08	4.63	0.08	7.26	0.10	8.72	4.46	7.45	829.48
	2.50	2.77	0.10	4.14	0.10	6.19	0.10	6.61	4.16	3.86	815.38
	5.00	1.51	0.08	3.24	0.12	4.51	0.12	4.25	0.12	2.64	816.90
	7.50	1.08	0.10	2.56	0.08	3.50	0.10	3.33	4.14	1.83	818.34
	10.00	0.83	0.08	2.11	0.10	2.86	0.10	2.86	4.20	1.22	817.10

Table II. Summary results for 7 other circuits using compiled code simulation, 16 processors

Circuit	Random		Input		Output		Greedy	
	S_p	Time	S_p	Time	S_p	Time	S_p	Time
c1355	3.78	0.18	4.48	0.18	5.19	0.18	7.62	8.18
c1908	3.60	0.26	3.74	0.24	6.16	0.26	8.24	10.96
c2670	4.81	0.32	5.42	0.32	5.58	0.32	10.37	15.76
c3540	5.07	0.48	4.01	0.56	5.31	0.46	10.28	20.30
c5315	6.71	0.68	6.75	0.68	6.16	0.64	12.10	30.32
c6288	3.52	0.92	6.97	1.00	5.46	0.80	7.98	37.56
c7552	8.69	0.90	7.03	0.88	6.88	0.88	13.83	41.42

Table III. Detailed results for 3 circuits using event driven simulation, 16 processors

Circuit	Ratio	Random		Input		Output		Greedy		Annealing	
		S_p	Time	S_p	Time	S_p	Time	S_p	Time	S_p	Time
c432	0.25	4.18	0.06	3.18	0.06	4.48	0.08	7.26	2.32	5.79	256.38
	1.00	1.84	0.04	2.45	0.06	3.06	0.06	4.26	2.34	2.46	276.00
	2.50	0.86	0.10	1.62	0.06	1.81	0.06	2.65	2.38	1.61	262.78
	5.00	0.45	0.04	0.98	0.06	1.03	0.06	1.80	2.36	0.69	262.02
	7.50	0.32	0.10	0.69	0.06	0.72	0.06	1.31	2.36	0.57	269.94
	10.00	0.26	0.08	0.53	0.06	0.55	0.06	1.10	2.48	0.46	271.92
c499	0.25	6.97	0.08	5.10	0.08	5.91	0.06	11.47	3.08	9.31	348.54
	1.00	3.64	0.06	4.26	0.04	4.79	0.06	7.19	2.76	4.97	304.32
	2.50	1.62	0.06	2.91	0.06	3.26	0.06	4.28	2.80	2.42	305.82
	5.00	1.00	0.06	1.77	0.06	2.08	0.08	2.81	2.76	1.43	309.00
	7.50	0.66	0.04	1.26	0.06	1.52	0.06	2.19	2.76	1.06	306.94
	10.00	0.51	0.06	0.98	0.04	1.20	0.06	1.67	2.74	0.68	309.24
c880	0.25	6.50	0.12	4.93	0.10	7.97	0.12	10.90	4.22	10.05	811.58
	1.00	3.37	0.08	4.34	0.10	6.72	0.10	7.28	4.14	5.10	814.78
	2.50	1.61	0.10	3.24	0.10	4.47	0.10	4.41	4.14	2.71	820.90
	5.00	0.89	0.10	2.12	0.10	2.85	0.12	2.87	4.14	1.24	817.60
	7.50	0.59	0.08	1.57	0.10	2.09	0.10	2.29	4.22	0.97	816.44
	10.00	0.45	0.10	1.25	0.10	1.64	0.10	1.82	4.20	0.73	829.98

Table IV. Summary results for 7 other circuits using event driven simulation, 16 processors

Circuit	Random		Input		Output		Greedy	
	S_p	Time	S_p	Time	S_p	Time	S_p	Time
c1355	1.87	0.18	3.28	0.16	3.69	0.20	5.28	7.54
c1908	2.04	0.26	3.34	0.22	4.83	0.24	5.61	10.54
c2670	2.54	0.32	5.13	0.32	4.69	0.30	8.09	15.48
c3540	2.92	0.44	3.44	0.46	4.38	0.44	7.05	20.74
c5315	4.57	0.64	6.34	0.64	5.33	0.64	10.57	30.74
c6288	1.98	0.78	6.29	0.84	4.14	0.80	5.77	35.88
c7552	6.29	0.86	6.88	0.86	6.36	0.86	11.84	41.30