Cost Function Error in Asynchronous Parallel Simulated Annealing Algorithms

M. D. Durand
Columbia University
June, 1989

CUCS-423-89

Abstract

Reducing synchronization constraints in parallel simulated annealing algorithms can improve performance. However, this introduces error in the global cost function. Previous work in parallel simulated annealing suggests that if the amount of error in the cost function is controlled, good quality results can still be obtained. In this paper, we present a model of error in asynchronous parallel simulated annealing algorithms to partition graphs and use it to predict the behavior of three different synchronization strategies. These three approaches were implemented on a 20-processor Encore, a shared memory, MIMD multiprocessor, and tested on a variety of graphs.

As predicted, the strategy which allows controlled error yields solutions comparable to those of the serial algorithm with greatly improved running time. Speedups from 5 to 11 (depending on the density of the graphs) using 16 processors were obtained. In contrast, the more synchronized version exhibited unacceptably high running times, whereas the version characterized by uncontrolled error yielded significantly poorer results. This confirms behavior seen in parallel simulated annealing algorithms to perform placement in VLSI circuit layout systems.

This work is supported in part by NSF grant # NSF-ASC-8808327 and DOE grant # DOE-W-31-109-Eng-38.

1. Introduction

In the past, parallelism has been used to reduce the runtime of VLSI placement by simulated annealing. To further reduce the runtime, a variety of methods for relaxing synchronization constraints at the expense of allowing error in the global cost function have been attempted with mixed results. The goal of the research presented in this paper is twofold: first, to attempt to predict which synchronization strategies will be tolerant of error given a model of the problem and the data; and, second, to see if the behavior of parallel simulated annealing reported in the placement literature is also seen with another application problem: graph partitioning. We study the effect of synchronization on runtime and final solution quality using three different parallel simulated annealing algorithms. These were run with differing numbers of processors on four different graphs. A model of error based on a stochastic model of the problem is presented and compared with the results. The results show that an algorithm for graph partitioning with reduced synchronization runs from 5 to 11 times faster (depending on the density of the graph) than the completely synchronized version and gives equally good solutions.

2. Background

Simulated annealing [11] is an approximate search method, based on a physical analogy to cooling solids, for solving combinatorial problems which are too complex to solve exactly. It uses a probabilistic technique for escaping from local minima in the search space, yielding better solutions than iterative improvement in many cases. It has been used to solve many combinatorial optimization problems including image processing, the design of codes and many problems which occur in the computer-aided design of VLSI circuits. A simulated annealing algorithm to place VLSI circuits has been incorporated into the highly successful TimberWolf layout package [16]. Unfortunately, although simulated annealing often gives better results than other algorithms, it is also very slow. One of the approaches used to decrease its running time is parallelism. The problem is that, for many problems, simulated annealing depends on a global state description in order to compute the cost function used to evaluate the quality of the current solution. In parallel implementations, several processors may need to manipulate this global state simultaneously. To guarantee that the computed cost function is correct requires synchronization which reduces the speedup gained from parallelism.

This synchronization cost can be reduced by allowing error in the cost function evaluation. Each processor evaluates the cost of the state as though it were the only processor manipulating the system. The assumption is that if the error is not allowed to become too large, an asynchronous parallel simulated annealing algorithm will still yield good results. Several different approaches to asynchronous parallel annealing have been applied to the problem of VLSI placement [4]. VLSI circuits are usually broken down into subcircuits which are designed separately. In the placement problem, the completed subcircuits must be placed on the chip in such a way as to allow all necessary connections to be routed in minimum area. In their implementation of parallel placement, Kravitz and Rutenbar [12] prevent error from occuring by scheduling simultaneously only those moves which will not interact. Casotto et al. [1, 2] allowed error to occur, but used scheduling heuristics to reduce the likelihood that interacting moves would be considered simultaneously. Rose [14, 15] used a different algorithm at high temperatures and switched to simulated annealing in the low temperature regime, where the probability of two processors interacting is greatly reduced. Darema's approach [3] used special data structures designed to allow fast cost computations. These were duplicated on every processor. Darema reported that the error in her algorithm depended on the frequency with which these local copies were updated. Similarly, in several message passing implementations [6, 8, 14], each processor used a private copy of the global state information to compute the cost function. Changes in cell position were periodically communicated to the other processors which then updated their local data structures. Generally, it was found that the parallel placement algorithms which used shared data structures were tolerant of error. Most yielded good solutions with significant speedup. However, implementations which used local copies were much more sensitive to error. When local data structures were allowed to become sufficiently inaccurate, erratic convergence and poor solutions resulted.

3. Sequential Simulated Annealing to Partition Graphs

Simulated annealing solves combinatorial problems by modeling them as ensembles of molecules. A solid which is heated to high temperature and then cooled slowly will be transformed into a crystal, representing a low energy state of the physical system. By considering the cost of the combinatorial state to be analogous to the energy of the solid and simulating cooling, a low cost solution is generated. In the simulation, the "temperature", T, is a parameter which controls the rate at which the search through the solution space progresses.

Every simulated annealing algorithm has a move generator, a cost or "energy" function and an annealing schedule. An initial solution is created using another algorithm. This initial solution is then perturbed by the move generator and the energy of the new solution is computed. If the change in energy, ΔE, is negative then the new solution is better than the current one and is accepted as the new current solution. If ΔE is greater than zero then the new solution is accepted with probability $e^{-\Delta E/T}$. This probabilistic acceptance of moves to poorer quality states allows the algorithm to escape local minima in the state space, yielding better results. The temperature is decreased as the algorithm progresses, reducing the size and number of the uphill moves which will be accepted. When the rate at which moves are accepted becomes very small, the system is considered to be "frozen" in the final solution. The annealing schedule is a set of parameters which determine how the system is "cooled". This includes the initial temperature, the rate at which the temperature is decreased, the number of moves performed at each temperature and the freezing criterion. The choice of annealing schedule can have a significant effect on the quality of the solutions generated by the algorithm.

Let us consider how simulated annealing is applied to graph partitioning. Graph partitioning was chosen as an application because it is simple to implement, typical of the types of problems which simulated annealing is designed to solve and well understood. Heuristic algorithms for partitioning graphs other than simulated annealing have been developed [10, 5], providing a method for checking the results of the simulated annealing algorithm. Johnson [7] has presented a serial simulated annealing algorithm for graph partitioning and performed extensive experiments on its behavior to determine good values for the annealing schedule parameters. We followed his approach.

The graph partitioning problem is to separate the N vertices of an undirected graph $G=(V,E)$ into two subsets of equal size, A and B, in such a way as to minimize the number of edges crossing the boundary between the two subsets (the cutset.) The simulated annealing algorithm to solve this problem is shown in Figure 1. The initial solution is generated by assigning half of the vertices to each subset at random. The array *member* gives the current subset for each vertex. (In Figure 1, the vertex being moved is assumed to be in subset A. If the current vertex is in subset B, then the correct behavior can be obtained by exchanging all As for Bs and vice versa.) A move consists of moving a vertex from its current subset to the other. Since moves of this sort allow the partition to become unbalanced, the energy is defined to be the sum of the cutset size and a term which discourages unbalanced partitions:

$$E = |\{(a,b) \mid a \in A \wedge b \in B \wedge (a,b) \in E\}| + \alpha \cdot (|A| - |B|)^2,$$

where the *imbalance factor*, α, is a constant which controls the relative importance of the cutset and the partition imbalance. In the algorithm, the sizes of the subsets are stored in the variables *maxA* and *maxB*. If the two subsets are not of equal size in the final solution, a greedy algorithm is used to balance the partition after annealing has completed.

The reader may wonder why a move set that results in unbalanced partitions was chosen, rather than a move set which selects one vertex from each subset and swaps them, thereby maintaining a balanced partition. Moving a single vertex at a time has two benefits. First, it allows the annealing algorithm to explore additional states which would not be legal if two vertices were always swapped. These additional states provide bridges which allow the algorithm to escape from local minima. Second, moving one vertex at a time can improve the efficiency of the algorithm. Each move set defines a neighborhood size: the number of states that can be reached from any given state in one move. The neighborhood size for swapping vertices is $N^2/4$ whereas for moving a single vertex the neighborhood size is N. In order to obtain good results, the amount of time spent at each temperature should be proportional to the neighborhood size [7]. Thus moving one vertex at a time, reduces the time spent at each temperature from $O(N^2)$ to $O(N)$.

4. Parallel Simulated Annealing

One way to parallelize simulated annealing is to accelerate individual moves by parallelizing the move generator, the energy computation and/or the modify routine. This method is effective if the individual moves are sufficiently complex to be appropriate for parallelism. A second method is to allow several processors to propose, evaluate and accept moves simultaneously. This method is more general. An algorithm which uses this technique can be used for any application. Where appropriate, these methods can be combined.

In order to get good running time in the second method, where several processors are manipulating the state simultaneously, it may be necessary to allow processors to access shared data structures without synchronizing in order to benefit from parallelism. As stated in section 2, this will result in error in the cost function used to evaluate the current state. Previous work has shown that for placement parallel simulated annealing is relatively tolerant of error in implementations which use shared global data structures [3, 1], but intolerant of error in implementations in which each processor has a private copy of the global state information [3, 14, 15, 9]. The reason is that in the latter case, error is allowed to accumulate. When all processors are working with a single shared global data structure, error only occurs if two processors are actually manipulating two related cells at the same time. However, if all processors are stopped, the shared global data will be correct and consistent. We call this temporary error. However, in implementations which use local copies this is not the case. As the algorithm progresses, the error accumulates in the local copies until they are explicitly updated. We call this cumulative error.

The first method cannot be used for graph partitioning, since the inner loop of the serial algorithm is not sufficiently complex to be parallelized. The move generator (selecting the current vertex in line 5), the modify operation (lines 13 through 15) and the balance term in the change in energy (line 11) have constant computational cost. The computation of the cutset term in ΔE (lines 7 through 10) requires $O(d)$

integer operations, where d is the average degree of the graph. Since we are considering graphs with average degree between 10 and 80, this computation does not contain enough parallelism. When this method was implemented, it was no surprise when the runtime increased with the number of processors.

Instead, our parallel algorithm uses simultaneous moves. Each processor is assigned a different vertex in line 5 and procedes to execute lines 7 through 15 using that vertex. The minimal synchronization requirement is that each processor be assigned a unique vertex in line 5. Beyond that, let us consider the consequences of using reduced synchronization to accelerate our graph partitioning implementation. There are two sources of potential error in our energy function: error in the *member* array during the computation of the change in cutset size and error in *maxA* and *maxB* during the computation of the balance term.

In order to guarantee that the cutset term is correct, we must require that no two processors consider neighboring vertices simultaneously. If we do not, then a processor may perceive a neighboring vertex to be in subset A when another processor has already moved it across the partition to subset B, resulting in an error in the cutset term. To prevent this error, not only the current vertex, but also its neighbors must be locked. However, locking the neighbors both reduces the amount of parallelism which can be gained from a given graph and increases the time spent executing synchronization primitives. This cost can be reduced by locking only the current vertex and not its neighbors. In this case, two processors may be assigned to neighboring vertices simultaneously, causing the cutset term in the energy to be inaccurate. This error is temporary since, although a vertex may temporarily appear to be in the wrong subset, the *member* array is always correct. The degree of error depends on the number of neighbors moving simultaneously. The greater the number of neighbors that may have moved since *member* was last read, the greater the error in the cutset. The number of neighbors moving depends on the size of the graph (N), the average degree of the graph (d), and the number of processors (P). If P processors are partitioning a graph with N vertices, the probability that any one vertex will currently be assigned to some processor is P/N. The expectation of the number of neighbors of the current vertex simultaneously assigned to processors is directly proportional to P and the total number of neighbors of the vertex and inversely proportional to N. Hence, for a fixed graph size, we expect the temporary error to increase as the number of processors increases and as the average degree increases.

The second potential source of error comes from the values of *maxA* and *maxB*. To guarantee that these are always correct, *maxA* and *maxB* would have to be locked in lines 13 through 15 to prevent any other processor from reading or modifying them. This option is not possible, since locking *maxA* and *maxB* every time the state is modified would serialize the algorithm at high temperatures when most moves are accepted. Instead, synchronization can be reduced by allowing processors to read *maxA* and *maxB* while they are being modified by another processor, resulting in temporary error in the values of *maxA* and *maxB*. We can further relax synchronization by allowing more than one processor to modify *maxA* and *maxB* simultaneously. In this case the error is cumulative since the actual values of *maxA* and *maxB* are wrong. Furthermore this error will grow because errors in *maxA* and *maxB* always make the partition appear to be more balanced than it actually is, encouraging moves which will unbalance the partition further still. To see this, consider that the change in the imbalance term depends on the difference in *maxA* and *maxB*, ΔAB. When a move is accepted, the value of *maxA* is incremented by one and the value of *maxB* is decremented by one (or vice versa.) If one of these operations gets lost in a collision, the result will be to reduce ΔAB. This in turn will reduce the perceived penalty for increasing the imbalance and also reduce the bonus for improving the balance of the partition. Thus what starts out as random error in *maxA* and *maxB* leads to divergent behavior as the algorithm progresses.

5. Experimental Implementation

To evaluate the tradeoff between speed and solution quality, three different synchronization strategies were compared experimentally: a synchronous version, which is correct except for the temporary error in *maxA* and *maxB*; an asynchronous version, which exhibits temporary error in both terms in the energy function; and a corrupt version which exhibits cumulative error in *maxA* and *maxB*. We expected to see no error in the synchronous case, little or no error in the asynchronous case and a major effect due to error in the corrupt case.

All three methods require each processor to lock the vertex it is currently considering to prevent it from being modified by other processors. In the **synchronous version**, no other processor may read the current vertex either. Other processors may read the neighbors of the vertex but no other processor may modify them. This is implemented by locking the neighbors with a read/write monitor which allows either multiple readers or a single writer, giving the writer priority. In order to prevent deadlock from occuring between processors trying to lock the same vertex, the neighbors of the current vertex are always locked in increasing numerical order and unlocked in decreasing order. In addition, *maxA* and *maxB* are locked in the modify routine, ensuring that no two processors can attempt to change *maxA* and *maxB* simultaneously. It is nevertheless possible that one processor will try to read the value of *maxA* or *maxB* while another one is trying to modify it. In this case, the reading processor may read wrong (and possibly inconsistent) values.

In the **asynchronous version**, no other processor may modify the current vertex, but other processors may read its subset in the *member* array. The neighbors of the current vertex are not locked with the result that the cutset term in the energy may be wrong. As above, *maxA* and *maxB* are locked for writing but not reading. In the **corrupt version**, like the asynchronous version, the current vertex is locked but the neighbors are not. The variables *maxA* and *maxB* are not locked, allowing more than one processor to modify them simultaneously. The values of *maxA* and *maxB* are not corrected during the course of the algorithm and therefore become increasingly wrong. When annealing completes, *maxA* and *maxB* are recomputed from *member*, giving the correct value for the final energy.

These parallel algorithms were written in the C language using the monitor macro package developed at Argonne National Laboratory [13] to implement the parallel constructs. They were executed on a 20-processor Encore Multimax, a shared memory architecture based on National Semiconductor NS32032 chips connected by a high speed bus. Four random 250-vertex graphs with average degree 10, 20, 40 and 80, respectively, were used as test data. The graphs were generated by creating an edge between each of the $\frac{N(N-1)}{2}$ pairs of vertices with probability $p = \frac{d}{(N-1)}$, where d is the desired average density.

The three parallel versions were executed on each of the four graphs using 1, 2, 4, 8 and 16 processors. Each experiment was run 30 times and running time and final energy were measured. For comparison, the serial version was also run 30 times on each of the four graphs. To check that the simulated annealing algorithm for graph partitioning is correct, the graphs were also partitioned using the Kernighan-Lin algorithm [10], a different heuristic algorithm. The results obtained using simulated annealing were close to but slightly better (within 10 percent) than those obtained using Kernighan-Lin.

An annealing schedule similar to that proposed by Johnson was used. Annealing began at a different initial temperature for each of the four graphs. The initial temperatures were chosen to yield an initial acceptance ratio of approximately 40 percent. Johnson's experiments with the serial algorithm show that an initial acceptance ratio of 40 yielded good results in a reasonable amount of time. The number of moves performed at each temperature was $60N$, after which the temperature was reduced by a factor of 0.95. The system was considered frozen after 40 temperature reductions by which time the acceptance ratio had dropped below 2 percent and no further progress had been made for at least three consecutive temperatures.

6. Results

6.1. Solution Quality

As expected, the synchronous version showed no error. The average final energies were independent of the number of processors and the graph density. The asynchronous method also showed no significant error, consistent with the hypothesis that simulated annealing is tolerant of temporary error. In contrast, the corrupt method did show error. The final energy increased as the number of processors increased. Figure 2 shows final energies as a function of the number of processors for a 250-node graph with average degree 80 annealed using the three different synchronization strategies. Each point is the average of 30 runs. The final energies for both the synchronous and the asynchronous versions are independent of the number of processors, P. As the figure shows, final energies obtained by the corrupt version increase with P. The results obtained for other graphs showed similar behavior.

Figure 3 shows a plot of average energy as a function of temperature for each of the three parallel algorithms. These curves show the behavior of simulated annealing as the algorithm progresses. Each point is the average of 30 runs executed with 16 processors on a 80-degree graph. Notice that in this figure, since temperature increases from left to right, the progress of the algorithm with time is from right to left. The synchronous and asynchronous algorithms, depicted by squares and circles, respectively, show similar behavior during the entire algorithm. The corrupt algorithm is represented by stars and crosses. The stars show the actual energy computed with correct values of *maxA* and *maxB* (recomputed for this purpose at the end of every temperature.) This energy is greater than the energy obtained by the synchronous algorithm and diverges more as the algorithm progresses. The crosses show the perceived value of the energy computed using the corrupt values of *maxA* and *maxB* at the end of every temperature. Similarly, the perceived value becomes increasingly lower than the synchronous curve. This illustrates how cumulative error in *maxA* and *maxB* causes the imbalance cost to be underestimated, leading to increasingly poor results.

6.2. Running time

As expected, the synchronous algorithm ran very slowly. This can be seen in Figure 4 which shows the performance of all three algorithms on the 80-degree graph. For this case, the synchronous algorithm ran slower than the serial algorithm for all experiments. On one processor, it ran from 3 to 10 times slower than the asynchronous algorithm, depending on the degree of the graph being partitioned. The asynchronous algorithm did much better than the serial algorithm. As the figure shows, the corrupt algorithm performed no better than the asynchronous algorithm. This suggests that locking *maxA* and *maxB* for writing does not constitute a major bottleneck. Given the poor final energies obtained with the corrupt algorithm, this approach is clearly not the one to use.

The speedup derived in the synchronous algorithm was limited by chunking and by the locking of neighbors. In order to minimize the cost due to scheduling, tasks were only scheduled once per temperature. The total number of moves to be performed at each temperature ($60N$) was divided by the number of processors and each processor was allocated its share of $\frac{60N}{P}$ moves. One ramification of this method, called chunking, is that as the number of processors grows the task size decreases, thereby increasing the ratio of the scheduling overhead to the amount of work performed by each processor. Furthermore, the average number of vertices that are locked simultaneously increases with the number of processors and the average degree of the graph. As $P \cdot d$ approaches N, the additional speedup which can be obtained dwindles. This is illustrated in Figure 5, which shows that the 80-degree graph approaches saturation much faster than the 10-degree graph does for the synchronous algorithm.

In contrast, the speedup of the asynchronous algorithm is limited because the task size is not large enough to amortize the scheduling costs. The running times required for this algorithm for all four graphs are shown in Figure 6. As in the synchronous case, the task size decreases as the number of processors increases because of chunking. Since task size depends on the average degree of the graph, more speedup can be obtained for denser graphs. Figure 6 shows that the 10-degree case saturated at 8 processors, whereas a significant improvement in the running time was made in the 80-degree case by increasing the number of processors from 8 to 16. Speedup is also limited by Amdahl's law. As can be seen in the figure, whereas the running times were quite different for the four graphs when one processor was used, they are very close for 16 processors. This is because with 16 processors the cost of inner loop has been greatly reduced and a major portion of the running time is due to the sequential portions of the program.

7. Conclusion

In this paper, we defined the concepts of temporary and cumulative error in parallel simulated annealing algorithms. We used these in a model of error for graph partitioning and argued that cumulative error would have a major effect on the final solutions of parallel simulated annealing algorithms to partition graphs, whereas temporary error would not. This model was tested experimentally on three parallel algorithms with different synchronization strategies. The results showed that graph partitioning is sensitive to cumulative but not to temporary error. Our asynchronous algorithm yielded good final energies with much improved running time, exhibiting speedups from 5 to 11 depending on the density of the graph being partitioned.

In this paper, a model of error based on characteristics of the problem and the data was used to predict the types of situations in which error would cause divergence. In the future, we need more quantitative analysis which can predict the extent of the divergence. More general models which can be applied to entire classes of problems must also be developed.

The success with which asynchronous parallelism has been applied to simulated annealing suggests that the class of problems that this technique can be used to solve can be expanded. In our algorithm, the maximum number of processors which can be used is limited by the task size. For random graphs of the type we considered, only 16 processors could be used. Graph partitioning is typical of problems which simulated annealing is used to solve in that it has a fast move set and the change in energy can be computed quickly. With the advent of larger multiprocessors, we can now consider using simulated annealing to solve problems with more expensive move generators and energy functions.

8. Acknowledgments

I wish to thank the staff at the Mathematics and Computer Science Division at Argonne National Laboratory for hosting the Summer Internship in Parallel Computing where the experimentation was performed. Thanks are also due to IBM who supports my graduate studies through the Research Student Associate program. Finally, I wish to thank Fredericka Darema and Greg Pfister for their thoughtful comments on this paper.

References

[1] A. Casotto, F. Romeo and A. Sangiovanni-Vincentelli.
A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells.
In *International Conference on Computer-Aided Design*, pages 30-33. IEEE, 1986.

[2] A. Casotto, F. Romeo and A. Sangiovanni-Vincentelli.
A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells.
IEEE Transactions on Computer-Aided Design CAD-6(5):838-847, September, 1987.

[3] F. Darema-Rogers, S. Kirkpatrick and V.A. Norton.
Simulated Annealing on Shared Memory Parallel Systems.
Technical Report RC 12195 (\#54812), IBM, T. J. Watson Research Center, Yorktown Heights, NY, October, 1986.

[4] M. D. Durand.
Parallel Simulated Annealing: Accuracy vs. Speed in Placement.
IEEE Design and Test of Computers :8-34, June, 1989.

[5] C. M. Fiduccia and R. M. Mattheyses.
A Linear-Time Heuristic for Improving Network Partitions.
In *19th Design Automation Conference*, pages 175-181. IEEE, 1982.

[6] R. Jayaraman and R. A. Rutenbar.
Floorplanning by Annealing on a Hypercube Multiprocessor.
In *International Conference on Computer-Aided Design*, pages 346-349. IEEE, November, 1987.

[7] D. S. Johnson, C. R. Aragon, L. A. McGeogh and C. Schevon.
Optimization by Simulated Annealing: An Experimental Evaluation (Part I).
1987.
Draft: ATT Bell Laboratories, Murray Hill, NJ.

[8] M. H. Jones and P. Banerjee.
An Improved Simulated Annealing Algorithm for Standard Cell Placement.
In *International Conference on Computer Design*, pages 83-86. IEEE, 1987.

[9] M. Jones and P. Banerjee.
Performance of a Parallel Algorithm for Standard Cell Placement on the Intel Hypercube.
In *24th Design Automation Conference*, pages 807-813. 1987.

[10] B. W. Kernighan and S. Lin.
An Efficient Heuristic Procedure for Partitioning Graphs.
Bell System Technical Journal , February, 1970.

[11] Kirkpatrick, S. and Gelatt, Jr., C. D. and Vecchi, M. P.
Optimization by Simulated Annealing.
Science 220(4598):45-54, May, 1983.

[12] S. A. Kravitz and R. A. Rutenbar.
Placement by Simulated Annealing on a Multiprocessor.
IEEE Transactions on Computer-Aided Design CAD-6(4):534-549, July, 1987.

[13] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson and R. Stevens.
Portable Programs for Parallel Processors.
Holt, Rinehart and Winston, Inc., 1987.

[14] J. S. Rose, D. R. Blythe, W. M. Snelgrove and Z. G. Vranesic.
Fast, High Quality VLSI Placement on an MIMD Multiprocessor.
In *International Conference on Computer-Aided Design*, pages 42-45. IEEE, 1986.

[15] J. S. Rose, W. M. Snelgrove and Z. G. Vranesic.
Parallel Standard Cell Placement Algorithms with Quality Equivalent to Simulated Annealing.
IEEE Transactions on Computer-Aided Design CAD-7(3):387-396, March, 1988.

[16] C. Sechen and A. Sangiovanni-Vincentelli.
The TimberWolf Placement and Routing Package.
IEEE Journal of Solid-State Circuits SC-20(2):510-522, April, 1985.

Table of Contents