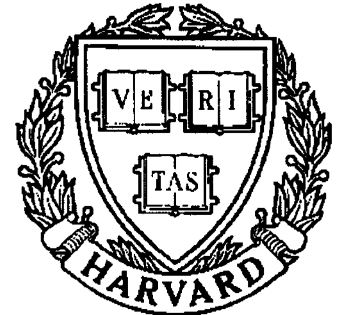
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), the University of Maryland, Harvard University, and Industry

Parallel Algorithms for VLSI Layout

by J. JáJá and S. Krishnamurthy

TR 89-51

Parallel Algorithms for VLSI Layout [1]

Joseph Já Já
Department of Electrical Engineering
Institute For Advanced Computer Studies
and
Systems Research Center
University of Maryland
College Park, MD. 20742
(301)454-1987
E-mail: joseph@wafer.src.umd.edu

and

Sridhar Krishnamurthy
Department of Electrical Engineering
and
Systems Research Center
University of Maryland
College Park, MD. 20742
(301)454-4836
E-mail: ksri@wafer.src.umd.edu

[1] Supported in part by NSF grant number DCR86-00375 and by the Systems Research Center contract number OIR8500108

Abstract

Efficient automatic layout tools are clearly essential for designing complex VLSI systems. Recent efforts have been directed toward developing parallel algorithms to handle the different subproblems involved in the layout phase. Some of these algorithms have been shown to offer significant speed-ups over the sequential ones. In this paper, basic parallel techniques that have been found to be effective for handling problems arising in the layout phase are reviewed. In particular, parallel algorithms for problems arising in partitioning, placement and routing are discussed. The algorithms used to handle these problems can be classified into two broad categories: iterative or direct. The iterative techniques such as simulated annealing and the Kernighan-Lin algorithm are very effective for partitioning and placement. The direct methods seem to be dominant in routing. These methods and some new methods are discussed in the general context of parallel processing. Efficient algorithms for the shared-memory model and for distributed-memory multiprocessors such as the hypercube are described. In addition, several special-purpose hardware for placement and routing are also outlined and their merits discussed.

Keywords: VLSI, partitioning, placement, routing, channel routing, line packing, detailed routing, maze routing, global routing, one-layer routing, shared-memory, distributed-memory, hypercube, prefix sum, knock-knee, line search, wave propagation, river routing, routability testing, constructive improvement, iterative improvement, Kernighan-Lin algorithm, simulated annealing, move decomposition, parallel moves, slicing structure, standard cells, macro cells.

1 Introduction

The importance of efficient automation tools for designing complex VLSI systems cannot be overemphasized. They are clearly essential to any cost effective successful design. A tremendous effort has recently been put into developing high quality tools to assist circuit designers at various stages of the design process. The physical layout stage is one of the crucial stages which is usually time-consuming and critical to the overall performance. The VLSI layout problem can be stated as the problem of determining the positions of the different components and their interconnections subject to various sets of constraints such as design rules, size and performance. This problem is in general very difficult. A typical strategy is to decompose the problem into several independent subproblems such as partitioning, placement, global and detailed routing. However each of these subproblems remains combinatorially intractable. This has led researchers to develop heuristics to handle these problems separately. But most of these heuristics are inefficient for moderate to large size designs.

Recent efforts have been directed toward developing parallel algorithms to handle the different subproblems involved in the layout phase. Some of these algorithms have been shown to offer significant speed-ups over the sequentail ones. In this paper, we will review the basic parallel techniques that have been found to be effective in handling problems arising during the layout phase. These techniques depend to a large degree on the class of subproblems involved. Three main classes of problems will be discussed: partitioning, placement, and routing. Briefly, these problems can be defined as follows. The problem of *partitioning* is to decompose a given circuit specified by its modules and their interconnections into various components subject to certain constraints and optimization criteria. A typical objective is to reduce the interconnection between the various components. The *placement* problem is the problem of specifying the locations of a set of modules so as to optimize a certain objective function subject to certain constraints. A special case is the standard cell placement where the problem is to arrange a set of cells with the same height into rows such that the total density is minimized. *Routing* is the problem of determining the detailed interconnections of a set of nets defined within a region containing already placed modules so as to optimize an objective function subject to a set of constraints.

The algorithms used to handle the three classes of problems introduced above can be broadly classified in two categories: *iterative* or *direct*. The iterative techniques are very effective for partitioning and placement. We will discuss two such methods: simulated annealing and the Kernighan-Lin algorithm. Neither of these methods seem to be easily amenable to parallel processing. However significant progress in this regard has been recently reported, especially in parallel simulated annealing. The direct methods seem to be dominant in routing. In particular, algorithms based on graph-theoretic algorithmic techniques are widely used. Corresponding parallel versions have been recently developed. In particular, efficient parallel algorithms exist for the important special case of *channel routing*. These algorithms are based on a parallel version of the well-known *left edge* (or *line packing*) algorithm and

several ideas from the greedy strategy for channel routing. Many hardware routers have been proposed for the general *detailed routing* problem, where *Lee's maze routing* algorithms are mapped directly into hardware. Efficient parallel algorithms have been reported for several important *one-layer* routing problems such as river routing, routing within a rectangle, and connecting module pins to frame pads.

The development of parallel algorithms has received a considerable attention in the literature. A popular model is the *shared-memory* model, where a number of processors have access to a large memory unit. This model has many interesting features including the fact that it allows the exploration of inherent parallelism without worrying about the communication problem between the different processors, and the fact that it is ideal for writing parallel algorithms. Moreover, efficient step by step simulations of such algorithms on more realistic parallel models have been recently developed. Several variations of the model exist. If concurrent reads from the same memory location and concurrent writes into the same memory location are disallowed, we obtain the EREW (Exclusive Read Exclusive Write) PRAM (Parallel Random Access Machine) model. If only concurrent reads are allowed, we get the CREW (Concurrent Read Exclusive Write) PRAM model. If both types of access are allowed, then it is referred to as the CRCW (Concurrent Read Concurrent Write) PRAM model. In the last case, there are several ways of allowing concurrent writes. A powerful set of techniques for the shared memory model have emerged. As a matter of fact, many nontrivial problems can be solved optimally on this model. For a recent survey, see [41]. For our purposes, whenever we refer to the shared-memory model we mean the CREW PRAM model, eventhough most of the statements hold true for the EREW PRAM model as well.

The analysis of the shared-memory algorithms do not necessarily predict the performance of these algorithms on existing systems. A more reasonable model is a constant-degree, fixed-connection network, in which the memory is distributed among the various processors. Such a model reflects some of the basic features of recent multiprocessors and allows to capture several important issues in parallel computation such as communication, granularity and synchronization. In this paper, we will use such a model based on the hypercube topology. However in most cases we will refer to experimentations performed on existing parallel machines and report the actual performance observed.

The rest of this paper is organized as follows. We start in section 2 by outlining the basic strategies for partitioning and their parallel implementations. Placement algorithms for various layout styles and their parallel implementations will be discussed in section 3. Section 4 will be devoted to various issues in routing. In particular, we discuss parallel strategies to handle channel routing, detailed routing, global routing, and one-layer routing. The last section is devoted to the conclusions.

2 Partitioning

The problem of partitioning is to decompose a given circuit into various components so as to optimize a given cost function. One objective is to reduce the number of signal nets that interconnect the various components. This problem arises in various aspects of design automation and it reduces the complexity of the placement phase [4]. During the partitioning stage, logic gates and functional modules are grouped together and assigned to blocks (components). Then, the placement stage determines the position of these gates and functional modules within each block as well as the positions of the blocks. It should be mentioned that the problem of partitioning arises in all layout styles. For example, in standard cell design, the cells are first partitioned into rows, and then the position of the cells within these rows are determined during the placement phase.

The general partitioning problem can be formulated as follows: Let M be a set of n cells (modules) $M = \{m_1, m_2, \cdots, m_n\}$, connected by a set of k nets, $N = \{n_1, n_2, \cdots, n_k\}$, where a net is defined as a set consisting of at least two cells, and each cell is contained in at least one net. The problem is to partition the set M into l classes, $P = \{p_1, p_2, \cdots, p_l\}$, such that for all $i \neq j$, $p_i \cap p_j = \emptyset$, and $\bigcup_{i=1}^{l} p_i = M$, subject to capacity constraints, (i.e., $|p_i| \leq s_i$, where s_i denotes the maximum number of cells that can be placed in partition p_i) so as to minimize

$$C(P) = \sum_{m_i \in p_h, m_j \in p_k, h \neq k} c_{ij}$$

where c_{ij} is the number of nets that are common to m_i and m_j. The problem can be further generalized by assigning weights to nets, so that the cost function is in terms of weights of nets rather than number of nets. The general partitioning problem is NP-hard [11] and hence efficient heuristics are needed to obtain near-optimal solutions.

Partitioning algorithms can be divided into two categories: algorithms that construct a partition from a description of the network, called *constructive algorithms*; and algorithms that improve upon an existing partition, called *iterative improvement algorithms.* A practical approach would be a combination of the two methods. In fact constructive algorithms are mostly used as starting points for iterative improvement techniques. In this discussion we shall concentrate on the iterative improvement strategies. The two well known iterative improvement strategies for partitioning are the Kernighan-Lin algorithm and simulated annealing. Both of these algorithms have performed consistently well on various examples [20]. We will consider without loss of generality the *bi-partition* problem i.e $|l| = 2$. We further assume that the desired partitions are balanced, i.e., The number of cells in the two partitions are roughly the same.

2.1 Kernighan-Lin algorithm and its Parallel Implementation

This algorithm [14] is not only fast, but it also performs very well in practice. The basic idea of the algorithm is to reduce the cost function by repeatedly swapping cells from two equal

sized partitions p_1 and p_2. Define the internal cost of a cell m_i with respect to a partition, as

$$I(m_i) = \sum_{m_i \in p_1, m_j \in p_1} c_{ij}$$

and the external cost to be

$$E(m_i) = \sum_{m_i \in p_1, m_j \notin p_1} c_{ij},$$

then the cost function is

$$C(m_i) = E(m_i) - I(m_i).$$

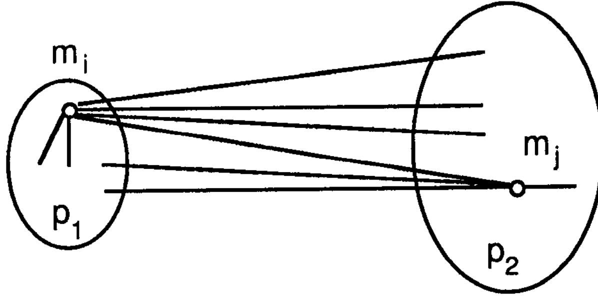
The *gain* $D(m_i, m_j)$ in the cost of the partition if two modules $m_i \in p_1$ and $m_j \in p_2$, are swapped, is given by

$$D(m_i, m_j) = C(m_i) + C(m_j) - 2c_{ij}.$$

In fact the gain measures the difference in cost between the solution before the interchange and the one after the interchange. Notice that the gain in general may assume both positive and negative values. To avoid *thrashing*, cells which are interchanged are *locked* in their new locations. The cost and gain computations are shown in Figure 1. Using these definitions we can state the Kernighan-Lin algorithm as follows. Locate two unlocked cells $m_i \in p_1$ and $m_j \in p_2$, that would produce the largest gain if swapped. Update the cost functions of the other cells in both the partitions. Repeat the above procedure, until all the modules have been interchanged. The result is a sequence of cell pairs (m_i, m_j), and their associated gains $D(m_i, m_j)$. Note that when the procedure stops the net gain is zero. The cost of the partition is tracked by computing $G_k = \sum_{i=1}^{k} D_i$, where D_i is the gain for the i-th interchange. Finally, the actual set of pairs, that are swapped, correspond, to the smallest k that maximizes $G = \sum_{i=1}^{k} D_i$. If $G > 0$, we repeat the above steps, starting with the configuration obtained after the swap, otherwise if $G = 0$ we stop.

An important feature of the above algorithm is that it does not terminate upon encountering a negative gain, for it may be possible that the effect of swapping a pair of cells with a negative gain, may eventually result in a better solution. A naive implementation of the above algorithm would require $O(n^2)$ time for locating a pair with maximum gain, thus resulting in $O(n^3)$ time for one iteration of the algorithm. However, a novel implementation by means of efficient data structures in [10] and [16], reduces the complexity to linear in the number of pins. The data structures help in avoiding unnecessary searching for the best cell to be moved, and also minimize unnecessary updating of cells affected by each move.

A parallel version of the Kernighan-Lin algorithm has been implemented by [12]. They consider the problem of partitioning the cells of the network into p roughly equal sets where p is a power of 2, on a message passing multiprocessor using p processors with each set residing on its own processor. The algorithm begins by partitioning the cells among the p processors.



$E(m_i)=4$; $E(m_j)=3$; $I(m_i)=2$; $I(m_j)=1$; $c_{ij}=1$; $D(m_i, m_j)=2$;

Figure 1: Cost and Gain Computation

The p processors are then divided into two roughly equal sets P_1 and P_2. The initial goal is to reduce the number of nets crossing P_1 to P_2. This is similar to the bi-partition problem described above. If P_1 or P_2 is $\emptyset$, then the algorithm stops. Otherwise, a processor from each set is selected as the *leader* of the set, say $l_1 \in P_1$ and $l_2 \in P_2$. The leaders execute the simplified Kernighan-Lin algorithm as follows. Each processor in $P_1 \cup P_2$, computes the C values (cost function) of all its cells. Each leader then selects an unlocked cell with the maximum C value. An assumption made here is that swapping the two selected cells between the two groups will automatically result in the largest gain. However, this assumption need not necessarily be true. The two cells are then locked and the C values of the remaining cells are then updated by the leaders. The leaders then repeat this process of locking cells and updating C values until all the cells in P_1 and P_2 are locked. The leaders then decide which cells have to be actually swapped by maximizing the total gain G as before. They then inform the processors of the decision and the individual processor contents are updated. The processors then repeat the algorithm until the total gain G cannot be decreased. Then, in parallel, P_1 and P_2 each apply this algorithm recursively. The leader stores the C values in the form of a heap, and constructs the heap in $O(n)$ time. Updating the C values, involves updating the heap by the leader and this takes $O(\log n)$ time for each modification. Since there are a total of P pins, the total time for finding the maximum C value and updating the remaining ones is $O(P \log n)$. Thus ignoring the message passing time, and assuming that the number of iterations per partition is a constant, the complexity of the above algorithm is $O(P \log n)$ time per iteration, where P is the number of pins. Hence, the entire algorithm takes $O(P \log n \log p)$ time, where p is the total number of cuts or partitions.

2.2 Parallel Algorithms Based on Simulated Annealing

Simulated annealing is a general purpose technique for combinatorial optimization, as proposed by [15]. It is an extension of the Monte Carlo method used to determine the equilibrium state of a collection of atoms at any given temperature T. The annealing process consists of melting a solid, and then slowly lowering the temperature until a crystal (state of minimum

energy) is formed. The rate of decrease in temperature has to be very low in the region where the crystal starts to appear. The Monte Carlo method can be used to simulate the above annealing process.

The simulated annealing technique has been used widely for a number of problems in CAD for VLSI such as partitioning, placement, floorplanning and global routing. Most of the heuristic algorithms for the above problems are greedy in nature and thus have a tendency to get stuck at *local minima.* However, it is possible to escape out of local minimas if *hill climbing* moves are allowed. The simulated annealing technique as proposed by [15], allows hill climbing moves. The basic idea of the technique is to first generate a random move and check if the cost of the new configuration resulted by the move satisfies the move acceptance condition. If the cost decreases, the move is accepted. However, if the cost increases, then the move is accepted with a probability $f(\Delta c_{ij}, T) = e^{\frac{-\Delta c_{ij}}{T}}$ where Δc_{ij} reflects the change in cost obtained by moving from configuration i to j and T is the temperature which acts as the controlling parameter. The above step is implemented by generating a random number between 0 and 1, whose value is then compared with the function f. If the number is larger than f then the move is accepted otherwise rejected. A specific number of moves are generated and checked at each temperature before it is decreased. The stopping criteria for the above algorithm usually depends on the rate of improvement of the cost function. If no change in the cost function is seen for a number of temperature update cycles, then the algorithm is stopped. Notice that a hill climbing move is more readily accepted at higher temperatures, and less likely to be accepted as the temperature is decreased.

Simulated annealing as used in the partitioning problem, ([15]) is simple to implement. Each move consists of randomly moving a cell from one partition to the other. The cost function is the same as the one used for the Kernighan-Lin algorithm. The annealing schedule was chosen to start at a high temperature T_0, then cooled exponentially $T_n = (\frac{T_1}{T_0}^n)T_0$, with the ratio $\frac{T_1}{T_0} = 0.9$. At each temperature a large number of moves are attempted (50,000 per temperature) before it is lowered. When the stopping criterion is satisfied the annealing stops.

Several experiments have been tried to compare simulated annealing to the Kernighan-Lin algorithm for partitioning ([19,20]). The quality of the solutions obtained by the two methods are very similar though in some examples the Kernighan-Lin algorithm does outperform simulated annealing. In addition the Kernighan-Lin algorithm takes considerably lesser time to run than simulated annealing.

Although the results obtained by simulated annealing are nearly optimal, simulated annealing is typically very slow. A number of multi-processor based simulated annealing algorithms have appeared in the literature ([1,2,3,5,6,9,13,17,18]). There are basically three approaches to converting a conventional simulated annealing (SA from now on) algorithm into a parallel algorithm. In [17] an attempt is made to break each step (move) into smaller tasks, some of which can be carried out in parallel. This decomposition is also referred to as *move decomposition.* However the experimental results reported in [17] show only negligible increases in speedup by this method. This is probably due to the fact that in annealing algorithms the typical moves are very simple in nature and only a small amount of work is

needed per move. Another approach to achieving parallelism is to make the processors work on different copies of the data [1]. The speedup is obtained by spending less time at each temperature, and by carefully selecting the initial conditions at each stage of the cooling schedule. However, the efficiency in processor utilization drops as the number of processors increases [1].

The third approach and the most promising is the *parallel moves* strategy which has been effectively used in a number of parallel SA algorithms ([5,6,7,8,17,18]). All the parallel moves work on the same data with each parallel move resulting in a new configuration. The possible relations between parallel moves can be *contradictory* or *orthogonal* or *independent.* Contradictory moves are ones that cannot be executed in parallel. For example, in partitioning, two moves are contradictory if they attempt to swap the same cell in one partition with two different cells in the other partition. In such cases only one of the contradictory moves is selected. Two moves are orthogonal, if they are not contradictory, and do not attempt to move the same cell. On the other hand, independent moves are orthogonal moves which also satisfy the requirement that the cost of each move computed in parallel is the same as the cost of each move computed sequentially. Thus independent moves also preserve the statistical properties of the SA algorithm but unfortunately in parallel annealing algorithms independent moves are rare. Orthogonal moves on the other hand occur in large numbers, and are easy to execute. However, the final cost of the configuration at the end of each iteration, after applying orthogonal moves, is not necessarily the total sum of the initial cost of the configuration and the individual gains of each move. This discrepancy may be explained due to the random error affecting the cost evaluation process [6]. Thus the convergence properties of the parallel SA algorithm is not the same as that of the conventional SA algorithm and they need to be further investigated.

Recently a distributed implementation of simulated annealing based on the parallel move strategy has been proposed in [2]. In this approach, functions are characterized so as to lend themselves well to a distributed stochastic search. A dedicated processor is associated with each variable of the given problem, in such a way that each processor need only communicate with a relatively small and well delimited subset of the remaining processors, and that together they be able to approximate a global minimum of the characterized function. The achievable degree of concurrency is strongly dependent of the interrelations between the different variables.

Parallel simulated annealing algorithms for partitioning using the parallel move strategy have been reported in [7]. The cost function is the same as before, reflecting the number of net crossings between the two partitions, as well as the balance criterion. Moves involve swapping cells from one partition to the other. An initial assignment of cells to the two partitions is made randomly. A set of moves $\pi_1, \pi_2, \cdots, \pi_p$ are generated in parallel. Each move is rejected or accepted in parallel as if it were the only move taking place. Each move π_i is accepted with a probability

$$p_i = min[1, e^{\frac{-g_i}{T}}]$$

where g_i is the gain change in the cost of the configuration if move π_i were to be implemented

on the original configuration at the beginning of the current iteration. The temperature schedule is of the form of

$$T_k = (T_{k-1}).\alpha_k$$

where T_k is the temperature at the kth iteration and α_k is a function of the total number of iterations to be done. Out of the p moves, let t ($0 \leq t \leq p$) moves be accepted. Then, the new configuration can be obtained by implementing these accepted moves. The parallel SA (PSA) algorithm as implemented in [7] performs better than the conventional SA (CSA) algorithm in the sense that for the same number of iterations the solution quality obtained by the PSA algorithm is better than that of the CSA algorithm. Besides for the same CPU time, the sequential execution of the PSA algorithm performed better than the CSA algorithm.

3 Floorplanning and Placement

The problem of floorplan design and placement are related to each other. In fact the floorplanning problem is a generalization of the placement problem. The placement problem is the problem of placing a set of circuit modules of fixed geometries such that a certain objective function (weighted sum of area, timing, power consumption of the chip) are optimized. There may be other additional constraints such as the constraints on the aspect ratios and area of the modules, and constraints on the module positions and orientations, making placement a difficult task. The floorplanning problem is less structured and is even more difficult. In this phase, the exact shape of the modules may be unknown and the designer needs to select areas and aspects ratios of the modules so as to optimize the design quality measured in terms of area of the chip, timing and power dissipation. Another degree of freedom may be the position of the pins of some of the modules. As a result the floorplanning optimization problem is considerably difficult.

When the relative positions of the modules in a layout are given, one problem may be the determination of the final positions of the modules so as to minimize the total area of the chip. If the modules are allowed to change their orientation or their aspect ratios are not fixed, then these new degrees of freedom need to be taken into consideration. Even these relatively simple problems turn out to be NP-complete in general. However, it turns out that for a special class of initial placement structures, called *slicing structures*, elegant sequential algorithms exist for the problem of determining the aspect ratios and orientations of the modules so as to obtain an optimal area [37]. We discuss some of these algorithms and their parallel implementations in section 3.1.

Semi-custom design methodologies such as standard-cells are used to reduce the design time at the expense of chip area. Here the designer constructs a circuit by selecting modules from a library of modules that implement pre-defined logic functions. In general, standard cells are of the same height but varying width. Thus they typically fit together in rows. Each

cell has terminals at the top and bottom. The terminals are connected to each other by wires which run in the channels between the rows. Feedthrough cells are used to connect terminals of cells on non adjacent rows. The standard cell problem is that of arranging a set of cells in horizontal rows. Efficient automatic techniques have been proposed for the standard cell placement problem ([25,26,34]). However a major limitation of the above algorithms is that they are extremely slow. Recently, a number of parallel algorithms for the placement of standard cells have been proposed ([3,5,6,17,18,21,32]) which have produced good solutions fast. Some of these algorithms will be outlined in section 3.2.

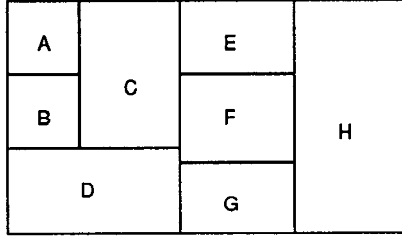
The macro/custom placement problem is that of placing circuit modules of different sizes and shapes on the chip. This problem is more difficult than the gate array or standard cell placement because of the difficulty in estimating the routing area as the modules are irregular in size. In section 3.3, we present some of the sequential and parallel algorithms that have been proposed for solving this problem. Finally a number of hardware accelerators for placement have been proposed ([23,27,31,38]) and we shall enumerate some of them in section 3.4.

3.1 Algorithms for Slicing Structures

Of particular interest in floorplanning and placement is a special class of rectangular dissections called *slicing structures* [30]. These structures exhibit several advantages over the general rectangular dissections. They can be represented in a compact form and are more suited for hierarchical layout. In addition they are computationally easier to handle ([4,30,37,39]). Sequential algorithms for finding the optimal shape and orientation of the cells in sliced floorplans have been proposed in [37]. Efficient algorithms for obtaining the layouts that can be represented by slicing structures have been proposed in ([4,39]).

Any given layout consists of an enclosing rectangle subdivided by horizontal and vertical line segments into nonoverlapping rectangles. A rectangle that is not subdivided by a line segment is called a *basic rectangle* (cell). A floorplan is said to be a slicing structure if either it is a basic rectangle or there is a line segment (slice) that divides the enclosing rectangle into two pieces such that each piece is a slicing. Slicing structures can be compactly represented by series-parallel graphs or by *slicing trees* [30]. Slicing trees specify the natural hierarchical structure of the floorplan. For simplicity it is assumed that slicing trees are oriented rooted binary trees. However, the algorithms described below are valid for nonbinary trees since it is easy to convert a nonbinary slicing tree into a binary slicing tree. Each nonleaf node of the tree is labeled $+$ or $*$, specifying whether this is a vertical or a horizontal slice. Each leaf corresponds to a basic rectangle (cell). A slicing floorplan and its corresponding slicing tree is shown in Figure 2.

The problem of determining the optimal shape and orientation of the cells given their relative positions to each other can be solved in polynomial time with an elegant algorithm proposed in [37]. A floorplan associates with each basic rectangle R two positive integers a_R and b_R which are the dimensions of the cell that must fit in R. Each cell has two possible orientations. Given a slicing tree describing the relative positions of the cells to each other,



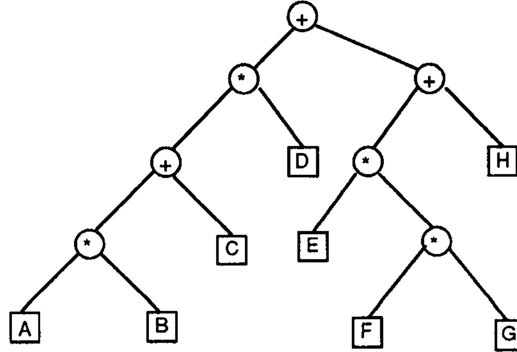


Figure 2: A slicing floorplan and its slicing tree

let ρ be an orientation of the cells resulting in a floorplan F. Let $h_F(\rho)$ and $w_F(\rho)$ be the height and width respectively of the resulting floorplan F corresponding to the orientation ρ. Define $\psi(h, w)$ to be a nondecreasing function in both its arguments (i.e., if $h < h'$ and $w < w'$ then $\psi(h, w) < \psi(h', w')$) and computable in constant time (for example, perimeter ($\psi(h, w) = 2h + 2w$) or area ($\psi(h, w) = hw$)). The objective is to minimize $\psi(h_F(\rho), w_F(\rho))$ over all orientations ρ. The sequential algorithm [37] achieves this by performing a bottom up traversal of the slicing tree. If u is a node of the tree T, let $F(u)$denote the floorplan described by the subtree rooted at u and let $L(u)$ denote the set of leaves in that subtree. For each node u, the algorithm constructs a list of pairs $(h_1, w_1), (h_2, w_2), \cdots (h_m, w_m)$ satisfying the following properties:

1. $m \leq |L(u)| + 1$,
2. $h_i > h_{i+1}$ and $w_i < w_{i+1}$ for $1 \leq i < m$.

In addition due to the nondecreasing nature of ψ, the algorithm does not keep $(h(\rho), w(\rho))$ in the list if there is another orientation ρ' that is strictly better than ρ in the h or w dimension (or both) and is not worse than ρ in either dimension. Finally, the algorithm minimizes ψ, by minimizing $\psi(h_i, w_i)$ over all pairs (h_i, w_i) in the list constructed for the root of T. In order to construct the orientation that minimizes ψ, the algorithm also maintains a pair of pointers for each pair in the list. The algorithm begins by constructing a list for each leaf of T. If the leaf cell has dimensions a and b with $a > b$, the list is $(a, b), (b, a)$. Note that this

list satisfies the above mentioned properties. The algorithm now works its way up the tree. In general, let u be a nonleaf node of T with children v and v', and say that u represents a vertical slice, and let

$$(h_1, w_1), \cdots, (h_k, w_k),$$
$$(h'_1, w'_1), \cdots, (h'_m, w'_m),$$

be the lists constructed for v and v', respectively, where

$$k \leq |L(v)| + 1, m \leq |L(v'| + 1.$$

A + operation at node u can combine any pair (h_i, w_i) from the list of v with a pair (h'_j, w'_j) from the list of v' as follows:

$$(h_i, w_i) + (h'_j, w'_j) = (max(h_i, h'_j), w_i + w'_j)$$

and create a new pair at node u. However, observe that one need not consider all the km such pairs, since a number of them may be suboptimal. For example, if $h_i > h'_j$, one need not consider combining pairs (h_i, w_i) with (h'_z, w'_z) for any $z > j$ as the resulting pair will be suboptimal.

The procedure for merging the two lists at node u is given as follows:

Algorithm Serial Merge

1. $i = 1, j = 1$.
2. If $i > k$ or $j > m$ then stop.
3. Add $(h_i, w_i) + (h'_j, w'_j)$ to the list for u, with pointers to (h_i, w_i) and (h'_j, w'_j).
4. If $h_i > h'_j$ then $i = i + 1$ and go to step 2.
5. If $h_i < h'_j$ then $j = j + 1$ and go to step 2.
6. If $h_i = h'_j$ then $i = i + 1, j = j + 1$ and go to step 2.

Notice that the running time of the above merging process is $O(k + m)$ and the resulting list at u satisfies the required properties. It is thus easy to see that the total running time is $O(nd)$ where d is the depth of the slicing tree and n is the total number of cells. Thus for a fully balanced tree the above algorithm runs in $O(n \log n)$ time.

A parallel version of the above algorithm can now be described. The basic method is again the same. We start at the leaves of the slicing tree and attempt to merge (in parallel) two basic rectangles into a new rectangle. The minimum ψ value is obtained by minimizing ψ over all the pairs in the list at the root. The procedure for parallel merge of two lists at a node u labeled + can be described as follows:

Algorithm Parallel Merge

1. Merge the two sorted lists $(h_1, w_1), \cdots, (h_k, w_k)$ and $(h'_1, w'_1), \cdots, (h'_m, w'_m)$ into new sorted list $(\hat{h}_1, \hat{w}_1), (\hat{h}_2, \hat{w}_2), \cdots, (\hat{h}_{k+m}, \hat{w}_{k+m})$ such that $\hat{h}_i \geq \hat{h}_{i+1}$ for $1 \leq i < k+m$.

2. Label a pair $(\hat{h}_i, \hat{w}_i)$ as 0 if the pair is originally from the list of node v, or by a 1 if the pair is originally from the list of node v'. This labeling operation results in a block of consecutive 0's and 1's, say $B_1, B_2, \cdots, B_p$. Notice also that within each such block $\hat{h}_i > \hat{h}_j$ and $\hat{w}_i < \hat{w}_j$ for $i < j$.

3. Mark the first element pair in each block B_i for $1 < i \leq p$. Let this pair be $(\tilde{h}_i, \tilde{w}_i)$.

4. For each element $(\hat{h}_j, \hat{w}_j)$ in block B_i, $1 < i \leq p$, add the pair $(\hat{h}_j, \hat{w}_j) + (\tilde{h}_{i+1}, \tilde{w}_{i+1})$ in the list for node u, with pointers to $(\hat{h}_j, \hat{w}_j)$ and $(\tilde{h}_{i+1}, \tilde{w}_{i+1})$, where $(\tilde{h}_{i+1}, \tilde{w}_{i+1})$ is the first element in block B_{i+1}.

This completes the description of the merging process. Notice that the list created at node u, satisfies the properties mentioned earlier. On a shared memory system, the parallel merge procedure can be implemented using $O(\frac{n}{\log n})$ processors, in $O(\log n)$ time ([22]). Thus we have an optimal parallel algorithm which runs in $O(d \log n)$ time using $O(\frac{n}{\log n})$ processors, where d is the depth of the slicing tree. Thus for balanced slicing trees the algorithms takes $O(\log^2 n)$ time, using $O(\frac{n}{\log n})$ procesors.

3.2 Algorithms for Standard-Cell Placement

In the standard cell design environment, the components of the circuit are *standard cells* or *polycells.* They are typically designed to fit together in rows. Once the cells are placed in rows the connections among the cells are made in the area between the rows, which is called a *channel.* Connections between cells, which are lying on non-adjacent rows, are made using *feedthrough cells.* Figure 3 shows a standard cell placement example. The total design time using standard cells is smaller than custom design. In addition, the layout is straightforward. However, a high fraction of the total area is used for routing and hence the need arises to develop efficient placement and routing algorithms for this design methodology. Most of the algorithms for standard cell placement make use of the iterative improvement strategy and have proven to be quite successful. In particular, the Min-Cut strategy and its variations ([4,25,33]) and simulated annealing ([34,35]) have proven to be quite effective in practice.

The basic idea of the min-cut approach is based on the recursive application of the Kernighan-Lin bi-partitioning algorithm (or its modified version). At first the circuit is partitioned into two parts either with a vertical slice or with a horizontal slice. The number of cells in each part is approximately the same. The area of each half is proportional to the area of the cells included in it. The next step of partitioning is to divide each of these groups in two, this time on the other axis. The process continues until there are only a few cells in each group. At this stage, the cells in each group are assigned to rows, keeping those cells that are in the same group, close together in the same row or in adjacent rows. Dunlop and Kernighan [25] have also taken into account the interconnectivity information between cells of two different groups, by means of *terminal propagation.* Stiener trees are

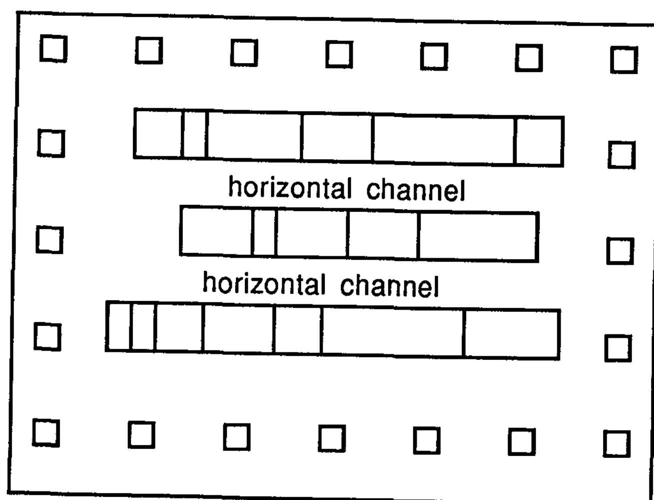


Figure 3: Standard cell placement

used to approximate the effect of external connections. The terminal propagation step first identifies the positions along the boundary where the interconnections between the modules to be partitioned and the modules external to the partition cut. The positions of the modules within the partition are then decided depending upon these intersection points. The final step is to create rows from the final partition, and the position of the modules within these rows are decided. The divide and conquer technique used in the min-cut approach makes it very attractive for its implementation on parallel computers. Some parallel algorithms for standard cell placement such as in [32], use min-cut based heuristics.

The TimberWolf3.2 standard cell placement and routing package uses the technique of simulated annealing [34]. The placement and routing is done in three distinct phases. In the first stage the cells are placed such that the total interconnect routing area is minimized. Each iteration in this stage involves the generation of a new configuration by making a weighted random selection from one of the following moves: (1) displacement of a single cell to a new location possibly on a different row, (2) exchange of the positions of two cells, or (3) change in the orientation of a cell. The acceptance criteria for the new configuration is governed by the function f given by:

$$f(\triangle c, T) = min[1, e^{\frac{-\triangle c}{T}}]$$

The cost function is a weighted sum of the total net lengths, a penalty function for reducing module overlaps and a penalty function for controlling the length of the rows. The controlling parameter T is updated as follows:

$$T_{new} = \alpha(T_{old}) * T_{old}, 0 < \alpha < 1.$$

During the second stage, the positions of the cells within each row are determined. The cells do not change rows during this stage. New configurations are generated by pairwise

exchange of cells within rows or by a change in the orientation of the cell. A global routing step is then performed, in which the number of wiring tracks is accurately estimated. The total number of wiring tracks is approximated to be equal to the sum of the densities of all the channels. Local changes are then made to refine the standard cell placement during the third and the final stage, such that the total channel density is minimized. New states are generated by randomly selecting cells and interchanging them with their neighbors or reorienting them and the effect on the total channel density is noted. If the change in the total channel density is less than or equal to 0 the new state is retained. The TimberWolf3.2 standard cell placement and routing package has proven to be quite successful in providing substantial chip area savings.

Various parallel algorithms for standard cell placement have been designed ([3,5,6,17,18, 21]) based on the TimberWolf standard cell placement package. Although the TimberWolf algorithm produces good results, a major limitation of the algorithm is that it is extremely slow. For example, placement using simulated annealing for circuits having 15,000 cells may require 100 to 360 hours on a conventional computer, while on a 64K-processor Connection Machine, the time required reduces to less than two hours ([6,40]). Two multiprocessor-based simulated annealing algorithms called Move-Decomposition and Parallel Moves have been reported by Rutenbar and Kravitz ([17,18]). Experimental results reported in [17] show that only a little speedup can be achieved by the move decomposition technique.

Parallel algorithms for standard cell placement based on the parallel move strategy and its variations, have been implemented on the hypercube ([3,13,21]), Connection machine ([6,40]), and MIMD multiprocessors [32]. The intuitive idea of the parallel move strategy is to carry out many moves or swaps separately and in parallel and to accept or reject each such move or swap separately and in parallel. A key to the success of such a strategy is the control of error which occurs due to the fact that the cost changes for parallel moves or swaps are in general not independent. The cells are first evenly distributed by area among the different processors. Each processor then repeatedly interacts with its neighboring processors (as in ([3,21])) or with other randomly selected processors (as in ([6,40])). For each such interaction either an exchange or a displacement is attempted. For best results the ratio of displacement moves to exchange moves is kept at 5 to 1 [34]. Other moves such as row interchange and overlap elimination are also attempted [6]. As in TimberWolf, in order to enhance the convergence of the algorithm during the later stages a range limiting mechanism is incorporated. The range limiter is a decreasing function of the temperature and a processor will not be able to interact with another processor outside its range. The cost function to be minimized has two components, one is the total estimated wire length, while the other is a linear function of the overlap among the cells. The cooling schedule of the parallel algorithms is similar to that of TimberWolf.

The parallel annealing algorithms have resulted in substantial speedups over the TimberWolf placement package without any significant loss in the quality of the placement. The error in the evaluation of the cost of the moves tends to disappear as the temperature decreases. Kravitz and Rutenbar ([17,18]) deal with the problem of error in cost evaluation by identifying sets of moves that do not interact. They refer to such moves as a *serializable set*

of moves. At high temperatures, the size of the serializable set of moves is small resulting in little or no parallelism. On the other hand at low temperatures, the size of the serializable set is close to the number of processors. To solve the problem of different performance at high and low temperatures, the authors used a strategy which combines move decomposition at high temperatures and parallel moves at low temperatures. The speedups obtained by this combined strategy when implemented on a VAX 11/784 (which consists of 4 VAX 11/780 processors), on a standard-cell design with approximately 100 cells, was around 2.5.

Using different algorithms at high and low temperatures (so as to control the error in the cost evaluation at high temperatures), has also been attempted in [32]. Here, the authors use min-cut based heuristics to replace the high temperature portion of the simulated annealing, while at low temperatures they use the parallel move strategy. Parallel algorithms based on the TimberWolf standard cell placer have also been implemented on the Connection Machine (CM) ([6,40]). The algorithms attempt more than 500 moves simultaneously, and obtain placements within 2% of the TimberWolf placement. A single, global description of the state is distributed across all the processors using two data structures, one for the cells and one for the nets. Since the algorithm stores a single description of the state across the machine, it ensures that after each iteration the state description is correct. This is in contrast to implementations of the parallel simulated annealing algorithm on machines with local memory, where periodic synchronization and global updates are needed after every few iterations, so as to ensure a correct state description ([13,32]). However, during each iteration the simultaneous movement of cells does introduce errors, especially at high temperatures, which they control by limiting the number of moves simultaneously. A more detailed review of parallel algorithms for placement using simulated annealing appears in [24].

3.3 Algorithms for Macro/Custom Placement

Macro/Custom cells are cells of any rectilinear shape. In addition, these cells may have fixed geometries including pin locations or they may have fixed area with a given aspect ratio range and with pins that need to be placed. As a result, the corresponding placement problem is considerably more difficult that the placement of sliced structures. Figure 4 shows a macro cell placement example.

A number of efficient sequential algorithms for this problem have been proposed ([28,29, 35,36]). As in other placement algorithms, the technique of simulated annealing has been effectively applied for the placement of macro/custom cells ([28,35]). Unlike the macro cell placement algorithm in [28] where only rectangular blocks of varying sizes are considered, the TimberWolf macro/custom cell placement package [35] is more general in nature, in the sense that it allows more flexibility on the shape of the modules. Each module is allowed to have an aspect ratio chosen within a particular range. In addition, the position of some of the pin terminals of the modules are not fixed. Each such uncommitted pin may be assigned to any one of a certain number of possible locations. The types of move employed by the algorithm include module displacement, module exchange, orientation change, aspect ratio change for flexible modules and pin-site assignment change for uncommitted pins. As in the

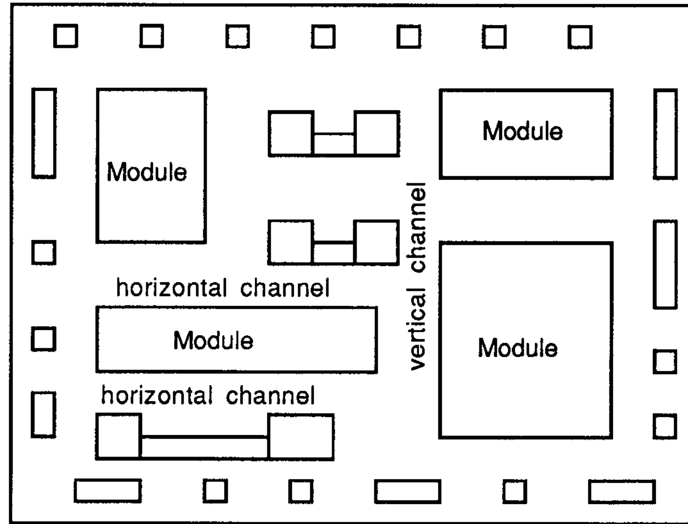


Figure 4: Macro cell placement

TimberWolf standard cell placement package a range limiter is used to control the scope of the moves. The cost function is a weighted sum of the total estimated wire length, a penalty function for module overlap and a penalty function for capacity overflows at pin sites. The TimberWolf macro/custom cell placement package has been used successfully on real world circuits.

Parallel algorithms for simplified macro cell placement based on the TimberWolf macro cell placement package have been implemented on multiprocessor systems [5]. The parallel move strategy is employed. The cells are initially partitioned, equally among the various processors. Each processor owns the cells in its partition. The processors run asynchronously, and the cost of each move is evaluated assuming that the remaining cells are fixed. Hence, this may result in errors in the cost evaluation. But this error tends to decrease as the temperature decreases. The algorithm attempts to assign cells to the processors in a way that minimizes their interaction and thus reduces errors. The is done by having a dynamic algorithm (a second simulated annealing algorithm running simultaneously with the placement algorithm) to keep cells that are physically close assigned to the same processor. The additional computational cost incurred by this algorithm is small as compared to the placement algorithm. The move set consists of single cell displacements and pairwise interchange of two cells. Each processor p selects a cell a in its partition and attempts to move it to a new position by making one of the above moves. In the case of a pairwise interchange move, processor p randomly selects another cell b, $b \neq a$, and the two cells are exchanged. Two cases may result: (1) b belongs to the partition in p in which case the exchange is easy, (2) b belongs to a partition of another processor q, in which case the two processors p and q must

interact to ensure that processor q is not moving cell b, while processor p is exchanging cell a with b. The set of possible moves is changed as the temperature decreases. At high values of temperature, the moves are unlimited but at low values of temperature the moves are limited using the range limiter [35]. The cost function to be minimized is a weighted sum of the estimated wire length, total area of the chip and a penalty function for cell overlaps. The parallel algorithm was implemented on the Sequent Balance 8000 and produced results comparable to the standard sequential algorithm. The authors ran the algorithm on a 30-cell design connected by 200 nets, and obtained a speedup of around 6 using eight processors.

3.4 Special Purpose Hardware for Placement

Another parallel processing approach to VLSI layout involves the design of special purpose hardware for specific layout problems, as opposed to the use of general purpose parallel computers. Several special purpose machines have been proposed for placement ([23,27,31, 38]).

It is interesting to observe that in most of the above placement schemes, the basic approach is to obtain an approximate solution to the problem using the *local search* method ([23,27,38]). The essential idea of these systems involves the execution of the following two main steps:

- Parallel application of local transformations (exchanges) to the large placement problem.
- Parallel evaluation of the resulting change in the cost function.

In the Module Interchange Placement Machine proposed by [27], use of special purpose hardware for speeding up the computation of the cost increment for the new placement resulting from the interchange of two modules is proposed. This aims to speed up the most compute bound step of the iterative improvement algorithm, namely the cost evaluation process. A 4-stage arithmetic pipeline is used to compute the incremental change in the cost function after each local transformation. The results indicate that a speedup of one order of magnitude or better can be achieved in comparison to the implementation of the iterative improvement algorithms through software.

An SIMD architecture with a two dimensional array of processors has been proposed in ([23,38]) for placement. The main steps of the proposed approach are:

1. Perturb the placement (in parallel) using adjacent pairwise interchange. The interchange is performed alternately between X-dimension neighbors and then with Y-dimension neighbors.
2. Compute (in parallel) the net gain resulted by the above interchange and decide whether to accept or reject the interchange.

3. If the interchange is accepted then (in parallel) swap the data between the two adjacent processors.

4. Each processor then broadcasts its new location (sequentially) either by means of a global bus or a control processor to the remaining processors.

One of the main drawbacks of the above schemes, is that erroneous decisions are possible when numerous pairs of modules are considered simultaneously for exchange. This is due to the fact that the algorithm performs the interchange of the position of two modules while assuming that the position of the remaining modules is fixed, which is not always true. This may result in the introduction of an error in the cost computation. Several methods have been proposed in [38] and [23] so as to reduce these errors.

A recent scheme proposed in [31], performs the hardware acceleration of the placement algorithm based on the *Divide and Conquer* technique. Following are the main steps of the proposed scheme:

1. Randomly partition the N logic modules in P clusters of size m.

2. Each processor now works on a placement problem of size m. The processor uses techniques such as enumeration or Branch and Bound to arrive at an optimal local placement.

3. A control processor then proceeds to optimally place the clusters again using enumerative techniques.

4. The resulting placement is then either accepted or rejected using some heuristic technique such as iterative improvement or the Metropolis procedure.

The authors claim that since m and P are small compared to N, enumerative techniques can be used for executing steps 2 and 3. The authors propose a reduced array architecture consisting of a one dimensional array of processors and a two dimensional array of memory elements. The processors operate in the SIMD mode supervised by the control processor. Best results are obtained when $m = N^{2/3}$ and $P = N^{1/3}$.

4 Routing

The most general version of VLSI routing consists of determining the detailed routing of a set of nets defined in a routing region containing a fixed set of modules subject to certain constraints and to certain optimization criteria. Some of the typical constraints include the number of routing layers available and the routing model, and the wire width and separation. In this section, we will restrict ourselves to the case where there is an underlying grid such that all terminals lie on grid points and the routing (or wiring) consists of gridline segments. One or two layers of interconnection will be assumed. In the case of two layers, we will use

the *standard* two-layer model where the horizontal wires run on one layer and the vertical wires run on the other. As for the optimization criteria, several parameters such as total area, total wire length, maximum wire length, number of vias, can be used.

Except for very few special subproblems, this general problem and its many variations and specializations are very difficult in the sense that the corresponding optimization problems are NP-complete. However efficient sequential and parallel heuristics exist for the important case of channel routing, where the routing region is restricted to the area between two parallel rows with no obstacles. We will start by discussing this important case and provide the basic techniques for obtaining efficient solutions in parallel. We will next outline the basic strategies to handle the detailed routing problem, especially in the case of a single net, and examine the work that has been done to accomplish this using parallelism. Global routing issues will be briefly discussed in section 4.3. This is the least understood problem in the parallel framework. The last section is devoted to one-layer routing, a class of problems for which efficient optimal solutions can be found in many cases.

4.1 Channel Routing

Channel routing is the problem of determining the wiring of a set of nets within a rectangular region with no obstacles and with the terminals located at opposite sides. The main optimization criterion is to minimize the number of tracks used. This is the most desirable routing subproblem. It is relatively well-understood and efficient sequential and parallel techniques for obtaining optimal and near-optimal solutions exist. An example of such a routing problem and a solution are given in Figure 5. Before presenting an efficient parallel strategy to solve this problem, we introduce few definitions.

As stated before we assume that there is an underlying grid such that the terminals lie on grid points and the routing consists of gridline segments. The *local density* d_c at column c is defined to be the number of nets whose intervals contain c. The *density* d is given by $d = \max_c\{d_c\}$. Clearly the number of tracks required for the routing must be at least d. The *vertical constraint graph* $G_{vc} = (V, A)$ gives also additional constraints that every solution has to satisfy. The set of vertices V consists of all the nets $\{n_i\}$ such that there is an arc $(n_i, n_j) \in A$ for each column c in which n_i has a top terminal and n_j has a bottom terminal. Such an arc indicates that n_i has to be connected to a higher track than n_j in column c. Our routing model will be the standard two-layer model where horizontal wires run on one layer and vertical wires run on the other layer.

The case when G_{vc} has no arcs will serve as the basis for a solution to the general case. This special case can be formulated as the problem of packing a set of (say horizontal) intervals into a minimal set of rows such that each row contains nonoverlapping intervals. A simple optimal algorithm starts by selecting an interval I_1 with the smallest left terminal, and then will select I_2, the interval with the smallest left terminal greater than the right terminal of I_1, and so on. This algorithm is called the *line packing* or *left edge* algorithm. An efficient parallel version appeared in ([45,50,66]) and is given below. Without loss of generality, we

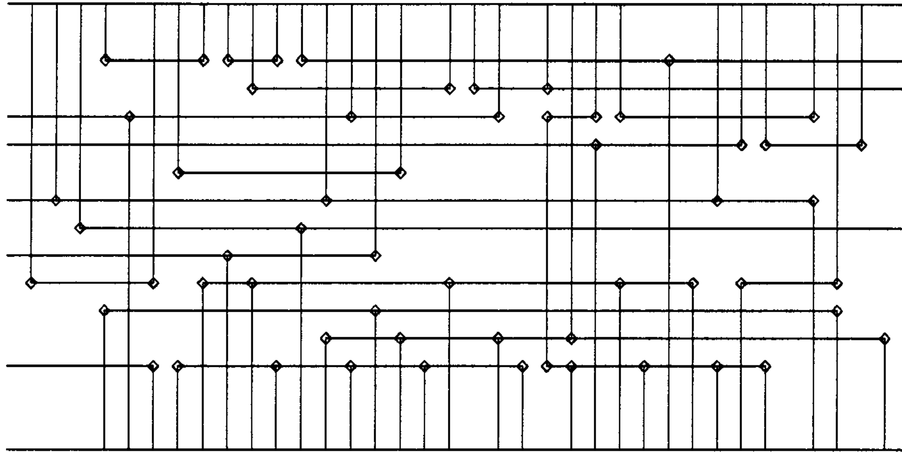


Figure 5: A channel routing example and its solution

will assume that each net consists of two terminals. Otherwise we can replace a multi-terminal net N with a two-terminal net N' consisting of the leftmost and the rightmost terminals of N.

Algorithm Parallel Line Packing

Input: terminals of all the nets
Output: d chains of nets, where d is the density, and each chain defines a set of nonoverlapping intervals.

1. Sort all terminals from left to right. If a right and a left terminal are equal, put the right terminal before the left.

2. Assign $+1$ to each left terminal and -1 to each right terminal. Compute the prefix sums of all the terminals.

3. For each right terminal r_i whose prefix sum is p, find the closest left terminal l_j to the right whose prefix sum is greater than p. Set $succ(N_i) = N_j$ if j exists. Otherwise set $succ(N_i) = nil$.

Theorem: The above algorithm provides an optimal solution to the channel routing problem in the case when no vertical constraints exist. It can be implemented on a shared-memory model in time $O(\frac{n}{p} + \log p)$ with p processors, where n is the number of nets. It can also be implemented on a hypercube with p processors in time $O(\frac{n}{p})$ whenever $n = \Omega(p^{1+\epsilon})$, for any $\epsilon > 0$.

It is shown in ([45]) that the above algorithm can be extended to handle routing in the knock-knee model in an optimal fashion. A routing in the *knock-knee* model consists of a

set of edge-disjoint paths. It is shown in [60] that three layers suffice to realize the channel routing in the knock-knee model. In the rest of this subsection, we will outline the extension of the parallel line packing algorithm to the standard two-layer model and report on some experimental results. This extension has appeared in [54].

The general algorithm is based on a combination of the algorithm parallel line packing and ways to resolve the corresponding vertical constraints. A sketch of the general algorithm is given below.

Algorithm Parallel Channel Routing

1. Split long nets into smaller subnets so as to reduce the number of vertical constraints. The split points of a net define portions which are *rising, falling* or *steady* [62].

2. Apply the parallel line packing algorithm. Modify so as reduce the vertical constraint violations.

3. Determine the wiring in each column which has no vertical constraint.

4. For all columns with vertical constraint violations, try to determine legal detour rows and columns.

5. If the above steps are not sufficient to complete the routing, the channel is made larger. This is done by adding a suitable number of rows at precomputed positions.

The sequential complexity of the above algorithm is $O(n^2)$, where n is the number of terminals of all the nets. All the known algorithms for the general case seem to have running times of at least of this order. Moreover this algorithm can be implemented on a shared-memory model with p processors, $1 \leq p \leq n^2$, in time $O(\frac{n^2}{p} + \log^2 n)$.

This algorithm has been implemented on a Connection Machine with 16K processors using C/PARIS. The input to the router is a standard file containing information about the nets, and the positions of the terminals. The output is a file which can be displayed on a SUN 3/160 through the interface of a graphics package. Table 1 shows the performance of this algorithm run on a set of benchmark examples from the literature. It also includes the best known reported running times on a VAX 11/780 of YACR2 ([63]) for comparison purposes. A significant speed-up is observed in most of the cases. We expect the performance to be much superior for larger problems.

4.2 Detailed Routing

Given a grid with a set of previously placed modules and a set of nets, the detailed routing problem is to specify the routing of all nets such that certain desired optimization criteria are met. Wires can run on a grid in any of several layers with the restriction that wires in a given layer have to be separated by a certain minimum distance. A wire may change layers by using

Name	Size	Density	Rows	Seq-Time (secs)	CM-time (secs)
				(YACR2) (on VAX 11/780)	(CM with 16K Procs)
YK3a	75	15	15	0.6	0.64
YK3b	119	17	17	1.1	0.84
YK3c	133	18	18	1.5	0.715
diff. example	241	19	20	2.8	1.03
rand1	216	21	23	3.2	0.849
rand2	194	20	21	2.0	1.17
rand3	201	16	17	1.9	1.02
rand4	224	15	18	3.0	0.819
rand5	262	18	18	2.6	1.128
HUGHES1	652	16	17	7.1	1.007
HUGHES2	703	15	17	7.6	1.044
HUGHES3	712	11	13	6.2	0.98
HUGHES4	691	19	21	11.0	1.006

Table 1: Performance of Parallel Channel Routing algorithm on the Connection Machine

4	3	2	3	4	5	6	7	8	9	10	11
3	2	1	2	3	4	5	6	7	8	9	10
2	1	A	1		5	6	7	8			
3	2	1	2		6	7	8	9	10	11	12
4	3	2	3							12	13
5	4	3	4		14				B	13	14
6	5				13	14				14	
7	6	7		11	12	13	14				
8	7	8	9	10	11	12	13	14			
9	8	9	10	11	12	13	14				

Figure 6: An example of maze routing

a *contact cut* or *via*. This problem has been studied extensively in the literature. A typical solution consists of a heuristic to order the nets and a method to route each net separately while avoiding the initial obstacles (modules) and the wires of the previously routed nets. It seems that no good heuristics exist to perform the ordering satisfactorily. Another alternative is the *rip-up* and *reroute* strategy, where a designer could interactively change the ordering of the nets whenever blockages occur. As for routing a single net while avoiding a set of obstacles, two basic strategies are well-known. The maze algorithms initially introduced by Lee [55] are used extensively. Any such algorithm guarantees an optimal solution, if it exists. However their time and space requirements are extremely high. The *line search* algorithms introduced by Hightower [51] use much less memory but do not necessarily produce a wiring even if one exists. In [44] the linear search technique was combined with some algorithms from computational geometry to obtain a fast algorithm that always guarantees a solution with a minimum length or minimum number of bends (vias), whenever such a solution exists. The running time of this algorithm depends only on the size of the obstacles, unlike that of a maze router whose running time will typically depend on the size of the overall grid and the set of obstacles.

A typical maze router works in three stages. See Figure 6 for an example. The first stage, *wave propagation*, consists of starting a wave from a terminal of a net called source and propagating the wave to the neighboring cells while avoiding the obstacles. This strategy is similar to breadth-first search in graphs. The second stage, *backtrace*, consists of tracing back a path from the destination terminal to the source terminal. The third stage, *label clearance*, consists of reclaiming all the cells marked during the first stage but not used by the path determined during the second stage. It is clear that the amount of time and space needed to route a single net consisting of two terminals A and B is $O(L^2)$, where L is the

shortest rectilinear distance between A and B. In the worst case, $L = N$, where the grid is of size $N \times N$. Moreover the algorithm requires a nontrivial amount of temporary storage for the wavefront. Many variations of this algorithm have appeared in the literature, but they all suffer from $O(N^2)$ worst case bound on the amount of time and space needed.

Maze routers are very popular but as indicated above very slow in general. With the new advances in VLSI technology, especially in packing densities, designs have become quite complicated and the maze routers will be too slow to handle the detailed routing. As a result, several researchers have proposed special-purpose hardware to handle maze routing ([42,43, 52,57]). The generality of these architectures vary significantly. Most of them are based around two-dimensional meshes of processors, where the routing grid is mapped directly onto the mesh. Most of the proposed architectures are supposed to reduce the complexity to $O(L)$ if there are $N \times N$ processors, where the grid is of size $N \times N$. In this case, Lee's algorithm can be mapped directly into the array. In the case where the number of processors is smaller, the most commonly used method is to partition the grid into frames, and each frame is processed sequentially by the array. Another method is to partition into subarrays, each of which will be handled by a processor of the mesh. An example is the *doubly twisted torus* of [57] that can route in $O(L)$ time using only $O(N)$ processors. In this case the grid is partitioned into a set of N subarrays (not consecutive) such that each subarray is handled by a processor. In the rest of this subsection, we will outline the general parallel algorithm developed in [58] and that can be implemented on a mesh with any number of processors.

This algorithm has three major parts. The first part is to decompose the grid evenly into subarrays, each of which is handled by a separate processor. The second part consists of performing global routing by viewing each subgrid as a single cell with a set of terminals and by associating a *capacity* with each edge connecting two cells. Therefore each processor is responsible for one cell and keeps track of the crossings at the boundary of the corresponding cell. Each source processor initiates an expansion message containing various information such as the net number, the source, the destination, and route bit vector. Upon receiving such a message, a processor updates the fields and advances the expansion message. Once a target processor is reached, the backtrace phase begins through the least cost expansion. Notice that each processor is responsible for the global routing of nets whose source terminals are contained in the corresponding cell. Once the global routing part is completed, the third phase is initiated. This phase consists of determining the placement of the crossings on the common boundaries of the cells. The procedure used is iterative and is based on calculating the position as a weighted average of current position and positions of the crossings as projected on the crossing border of strands connected to the crossing. The last phase is a local operation consisting of performing a detailed routing of each cell by its processor. This algorithm was implemented on the hypercube machine NCUBE/six with 64 processors. It was also implemented on a large mainframe. The hypercube execution time was about three times as fast.

4.3 Global Routing

The global routing phase is the process of defining routing regions and assigning nets to particular routing regions such that the routing of all nets could be completed during the detailed routing phase. The routing regions could be channels, switchboxes, L-shaped, etc. Only channels appear in the gate array or standard cell layout styles. These are the most desirable routing regions because efficient channel routing techniques exist as we have seen before. Typically the global routing problem is viewed as a graph problem, where the vertices represent the routing regions and the edges represent the boundaries between the routing regions. Moreover a weight $w(e)$ is associated with each edge e which represents the number of tracks crossing the boundary associated with e. Given a multi-terminal net N, a basic problem is to find a shortest path in the graph connecting the terminals of N. Notice that this is the Steiner tree problem which is well-known to be NP-complete. A solution to this problem will provide an assignment of the routing regions of N. The problem of handling all the nets simultaneously can be tackled in two different ways. One way, which is the more popular method, is to route the nets sequentially. The other method, which is more suitable for parallel implementation as we will shortly see, is to route all the nets simultaneously, and then rip-up and reroute the nets which cause congestion.

There are relatively very few reported results on parallel algorithms for global routing. The known methods do not seem to be easily amenable to parallel processing techniques. We will outline below two parallel methods for global routing.

One obvious approach would be to route all the nets in parallel. Each of the nets could in turn be handled by a set of processors say by subdividing the net into two-terminal subnets and have each processor handle each two-terminal subnet separately. The combination of the subsolutions may not generate a valid routing because of congestion in certain regions. A subset of the nets will be rerouted by using the same method. After few iterations we will obtain the desired solution. This is essentially the method used in [61] for determining a global routing for standard cells. The algorithm used to handle the two-terminal subnet problem consists of evaluating the cost of a subset of all possible two-bend solutions, and choosing the one with the lowest cost. Even this step can be executed in parallel. The algorithm was implemented on the Encore MULTIMAX with eight processors. The resulting speed-ups were in the order of 5-7.

The other method is to use a parallel version of the simulated annealing algorithm for global routing. Simulated annealing can be used efficiently for global routing [67] and hence the parallelization methods mentioned in the previous sections for simulated annealing are applicable here.

4.4 One-Layer Routing

As we have seen above, almost all the optimization problems arising in VLSI routing are NP-complete. One notable exception is the class of one-layer routing problems such as those arising with a hierarchical layout strategy ([53]). Efficient serial algorithms have already

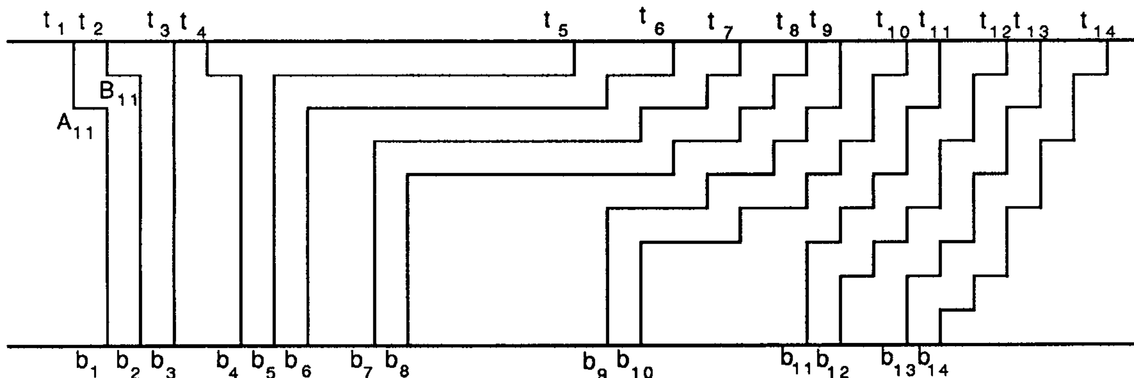


Figure 7: An example of a river routing problem

appeared for most of these problems. Corresponding parallel algorithms have appeared in [46], [47], and [48]. Below we give a sketch of some of the main results.

The class of general one-layer routing problems involves routing between ordered sequences of terminals such that the final layout is planar. *River routing* is the problem of wiring two ordered sets of terminals across a channel. The *separation problem* is to find the minimum width of the channel necessary to wire all nets such that any two wires are separated by a minimum unit distance. A more general version is to perform planar routing where the ports lie on the boundary of a simple rectilinear polygon. Another related important problem is to connect a set of pins on a module to a set of pads lying on the boundary of a chip.

River routing is by far the easiest of the three problems mentioned above. We now present the solution to the separation problem. Let $\{N_i = (b_i, t_i) | 1 \leq i \leq n\}$ be an instance of the channel separation problem. A net N_i is a right net if $b_i < t_i$; it is a left net if $b_i > t_i$. Otherwise it is a vertical net. We can partition the nets into *right* blocks, *left* blocks, and *vertical* blocks. A set of right nets $N_i, N_{i+1}, \ldots, N_k$ is a right block if it is maximal with the property $b_j < b_{j+1} \leq t_j$, for $i \leq j < k$. We can similarly define left and vertical blocks. The wiring problem is reduced to wiring each block separately. We will concentrate on the wiring of right blocks. An optimal strategy for right blocks consists of wiring the nets from left to right such that for each net we move from the bottom terminal upward and try to stay as close to the upper row as possible ([49,59,65]). The wiring of a net can be specified by the coordinates of its bendpoints. For example, net N_1 of Figure 7 has the bendpoints A_{11} and B_{11}. For each net N_i, we have $2p$ bendpoints for some p. Not all of these bendpoints are needed to determine the overall wiring. We call the bendpoints closest to the bottom row the *characteristic* bendpoints of a net. The algorithm to find the minimum separation is based on the following fact.

Fact: Let N_i be a net in a right block and let $\hat{j}$ be the minimum $j \leq i$ such that $t_j + (i - j - 1) \geq b_i$. Then the coordinates of the characteristic bendpoints of N_i are $A_{i1} = (b_i, i - \hat{j} + 1)$ and $B_{i1} = (t_{\hat{j}} + i - \hat{j}, i - \hat{j} + 1)$.

The algorithm to find the minimum separation and the characteristic bendpoints is as follows.

Algorithm Separation

1. Partition the nets into blocks.

2. Compute $\hat{j}(i)$ for each net N_i. Use above fact to obtain all the characteristic bendpoints.

3. Let the characteristic bendpoints be $B_{i1} = (x_{i1}, y_{i1})$. Then the minimum separation is $\max\{y_{11}, \ldots, y_{n1}\} + 1$.

The following result is shown in [46].

Theorem: The above algorithm finds the characteristic bendpoints of the n input nets and the minimum channel separation in time $O(\frac{n}{p} + \log p)$ time on a shared-memory model with p processors. The same algorithm can be implemented to run on the hypercube model in time $O(\frac{n}{p})$ whenever $n = \Omega(p^{1+\epsilon})$, for any positive constant ϵ.

The problem of routing a set of nets within a rectangle (or polygon) can be done efficiently in parallel. However the algorithm is considerably more involved. We give here a brief sketch of the overall strategy. We only consider the problem of testing whether or not it is possible to route a set of nets in a given rectangle R.

Given a set of nets $\{N_i\}$ in R, these nets may not be routable because of one of the following reasons: (i) the graph determined by the nets in the rectangle is not planar, or (ii) the wiring of all the nets requires more area. The first case can be handled easily by sorting and computing prefix sums essentially. Actually, the graph is not planar if and only if there is a net $N_i = (a_i, b_i)$ such that the prefix sum value of a_i is not equal to that of $b_i + 1$. A *side* net is a net whose terminals lie on the same side of the rectangle. A *side group* is a group whose representative net (a net not covered by any other net in the group) is a side net. A *corner* net (group) and a *cross* net (group) can be defined similarly. In Figure 8, the side group is $\{N_{14}, N_{15}\}$, the corner group is $\{N_1, N_2, N_3, N_4, N_5\}$, and the cross group is $\{N_6, N_7, N_8, N_9, N_{10}, N_{11}, N_{12}, N_{13}\}$. The overall strategy of the routability testing is described next.

Algorithm Routability Testing

1. Partition the input nets into groups.

2. Ignoring corner and cross groups, determine the contours of the side groups and test if the combined side groups are routable.

3. For each corner group, test whether the nets in the group are routable within the rectangle ignoring the remaining groups. Notice that there are at most four corner groups.

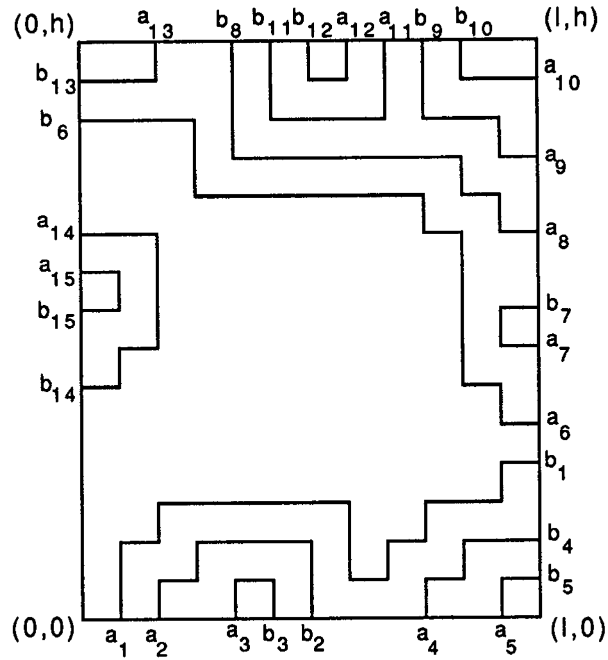


Figure 8: Basic river routing around a rectangle boundary

4. For each cross group, test whether the nets in the group are routable within the rectangle ignoring the remaining groups.

5. Test whether any of the contours generated at steps 2,3,4 intersect. The problem is routable if and only if no two contours intersect.

A complexity result similar to the one stated above for the minimum separation problem is shown for routability testing and detailed routing in [48].

5 Conclusions

In this paper, parallel algorithms for the automatic layout of integrated circuits have been reviewed. Particular emphasis has been placed on partitioning, floorplanning and placement, and routing. Several special-purpose architectures for placement and routing have also been outlined and their merits discussed. Attention has been given to new algorithms and approaches. Whenever possible and meaningful, a comparison among the various approaches have been given. Some of these algorithms have been shown to offer significant speed-ups over the sequential ones. Moreover, problems which have been previously impractical to handle due to their size, can now be handled effectively in a reasonable amount of time. However, while much has been accomplished and significant results have been obtained, several problems are still open either because the solutions obtained thus far are not satisfactory, or the known methods are sequential in nature and are not easily amenable to parallel processing

techniques. In addition, new technologies such as the availability of multiple layers for routing, pose new challenges and require the development of new algorithms and approaches. In summary, we expect that the development of parallel algorithms and architectures for the automatic layout of integrated circuits will continue to be an exciting area of research for several years to come.

References

[1] E.H.L. Aarts, F.M.J de Bont, J.H.A. Habers, and P.J.M. van Laarhoven, "A Parallel Statistical Cooling Algorithm," *Symposium on the Theoritical Aspects of Computer Science(STACS),* Paris, Jan 1986.

[2] V. Barbosa and E. Gafni, "A Distributed Implementation of Simulated Annealing," *Journal of Parallel and Distributed Computing,* vol. 6, pp. 411-434, 1989.

[3] M. Jones and P. Banerjee, "Performance of a Parallel Algorithm for Standard Cell Placement on the Intel Hypercube," *Proceedings of the 24th DA Conference*, pp. 807-813, 1987.

[4] M.A. Breuer, *Design Automation of Digital Systems*, Prentice Hall 1972.

[5] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentalli, "A Parallel Simulated Annealing Algorithm for Placement of Macro-cells," *IEEE Trans. on CAD*, pp. vol. CAD-6, No. 5, pp. 838-847, Sept. 1987.

[6] A. Casotto and A. Sangiovanni-Vincentalli, "Placement of Standard Cell using Simulated Annealing on the Connection Machine," *Proceedings of the Int. Conf. on CAD*, pp. 350-353, Nov. 1987.

[7] M.J. Chung and K. K. Rao, "Parallel Simulated Annealing for Partitioning and Routing," *Int. Conf. on Computer Design*, pp. 238-242, Oct. 1986.

[8] F. Darema-Rogers, S.Kirkpatrick, and V.A.Norton, "Simulated Annealing on Shared Memory Parallel Systems," *IBM Journal on R & D*, 1987.

[9] S. Devadas and A.R. Newton, "Topological Optimization of Multiple Level Array Logic: On Uni and Multi-processors," *Int. Conf. on CAD*, pp. 38-41, Nov. 1986.

[10] C.M. Fiduccia and R.M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proceeding of the 19th DA Conference*, pp. 175-181, 1982.

[11] M.R. Garey and D.S. Johnson, *Computers and Intractability*, Freeman, pp. 209-210, 1979.

[12] J.R. Gilbert and E. Zmijewski, "A Parallel Graph Partitioning Algorithm for a Message-Passing Multi-processor," *International Journal of Parallel Programming*, pp. 427-449, 1987.

[13] R. Jayaraman and R. Rutenbar, "Floorplanning by Annealing on a Hypercube Multi-processor," *Int. Conf. on CAD*, pp. 346-349, 1987.

[14] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, vol. 49 pp. 291-307, 1970.

[15] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, pp. 671-680, 1983.

[16] B. Krishnamurthy, "An Improved Min-Cut Algorithm for Partitioning VLSI Networks," *IEEE Trans. on CAD*, vol. CAD-3, pp. 436-446, 1984.

[17] S.A. Kravitz and R.A. Rutenbar, "Placement by Simulated Annealing on a Multipro-cessor," *IEEE Trans. on CAD* vol. CAD-6, pp. 534-539, July 1987.

[18] R.A. Rutenbar and S.A. Kravitz, "Layout by Annealing in a Parallel Environment," *Int. Conf. on Computer Design*, pp. 434-437, Oct. 1986.

[19] S. Nahar, S. Sahni, E. Shragowitz, "Simulated Annealing and Combinatorial Optimiza-tion," *Proceedings of the 23rd DA Conference*, pp. 293-299, 1986.

[20] A. Sangiovanni-Vincentalli, "Automatic Layout of Integrated Circuits," 1987.

[21] P. Banerjee and M. Jones, "A Parallel Simulated Annealing Algorithm for Standard Cell Placement on a Hypercube Computer," *Int. Conf. on CAD*, pp. 34-37, 1986.

[22] G. Bilardi and A. Nicolau, "Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-Memory Machines," *Siam Journal of Computing*, vol. 18, no. 2, pp. 216-228, April 1989.

[23] D.J. Chyan and M.A. Breuer, "A Placement Algorithm for Array Processors," *Pro-ceedings of the 20th DA Conference*, pp. 182-188, 1983.

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

[25] A. Dunlop and B. Kernighan, "A Procedure for Placement of Standard Cell VLSI Circuits," *IEEE Transactions on CAD*, vol. CAD-4, pp. 92-98, Jan. 1985.

[26] L.K. Grover, "A New Simulated Annealing Algorithm for Standard Cell Placement," *Int. Conf. on CAD*, pp. 378-380, 1986.

[27] A. Iosupovici, C. King and M.A. Breuer, "A Module Interchange Placement Machine," *Proceedings of the 20th DA Conference*, pp. 171-174, 1983.

[28] D.W. Jepsen and C.D. Gelatt, "Macro Placement by Monte Carlo Annealing," *Int. Conf. on CAD*, pp. 495-498, 1983.

[29] U. Lauther, "A Min-cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation," *Journal of Digital Systems*, vol. 4, pp. 21-34, 1980.

[30] R. Otten, "Automatic Floor-plan Design," *Proceedings of the 19th DA Conference*, pp. 261-267, 1982.

[31] C.P. Ravi Kumar and S. Sastry, "Parallel Placement on Reduced Array Architecture," *Proceedings of the 25th DA Conference*, pp. 121-127, 1988.

[32] J.S. Rose et al, "Fast, High Quality VLSI Placement on an MIMD Multiprocessor," *Int. Conf. on CAD*, pp. 42-45, 1986.

[33] P. Suaris and G. Kedem, "An Algorithm for Quadrisection and Its Application to Standard Cell Placement," *IEEE Transactions on Circuits and Systems*, vol. 35, pp. 294-303, March 1988.

[34] C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf3.2: A New Standard Cell Placement and Global Routing Package," *Proceedings of the 23rd DA Conference*, pp. 432-439, 1986.

[35] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *IEEE Journal of Solid-State Circuits*, vol. 20, pp. 510-522, April 1985.

[36] L. Sha and R.W. Dutton, "An Analytical Algorithm for Placement of Arbitrary Sized Rectangular Blocks," *Proceedings of the 22nd DA Conference*, pp. 602-608, 1985.

[37] L. Stockmeyer, "Optimal Orientations of Cells in Slicing Floor-plan Designs," *Information and Control*, vol. 59, pp. 91-101, 1983.

[38] K. Ueda et al, "A Parallel Processing Approach for Logic Module Placement," *IEEE Trans. on CAD*, vol. CAD-2, pp. 39-47, Jan. 1983.

[39] D.F. Wong and C.L. Liu, "A New Algorithm for Floorplan design," *Proceedings of the 23rd DA Conference*, pp. 101-107, 1986.

[40] C.P. Wong and R.D. Fiebrich, "Simulated Annealing-Based Circuit Placement Algorithm on the Connection Machine System," *Technical Report*, Thinking Machines Corporation, CAD87-2, 1987.

[41] R. Karp and V. Ramachandran, "A Survey of Parallel Algorithms for Shared-Memory Machines," *Technical Report*, no. ucb/csd 88/408, University of California, Berkeley, CA.

[42] M. Breuer and K. Shamsa, "A Hardware Router," *J. of Digital Systems*, vol. 4, pp. 393-408, 1981.

[43] T. Blank, M. Stfik, and W. VanCleempt, "A Parallel Bit Map Architecture for DA Algorithms," *Proceedings of the 18th Design Automation Conference*, pp. 836-845, 1981.

[44] J. Cohoon and D. Richard, "Optimal Two-terminal Wire Routing," *Advanced Research in VLSI, Proc. 4th MIT Conf.*, pp. 259-280, 1986.

[45] S. Chang and J. JáJá, "Parallel Algorithms for Channel Routing in the Knock-knee Model," *Proceedings of the 1988 International Parallel Processing Conference*, pp. 18-25.

[46] S. Chang and J. JàJá, "Parallel Algorithms for River Routing," *Proceedings of the 1988 International Parallel Processing Conference*, pp. 9-13.

[47] S. Chang and J. JáJá, "Parallel Algorithms for Wiring Module Pins to Frame Pads," *to appear in the Proceedings of the 1989 International Parallel Processing Conference.*

[48] S. Chang, J. JáJá, and K. Ryu, " Optimal Parallel Algorithms for One-layer Routing," *Technical Report*, UMIACS, University of Maryland, College Park, 1989.

[49] D. Dolev, K. Karplus, A. Seigal, A. Strong and J. Ullman, "Optimal Wiring Between Rectangles," *ACM STOC*, pp. 312-317, May 1981.

[50] E. Dekel and S. Sahni, "Parallel Scheduling Algorithms," *Operations Research*, vol. 31, no.1, January-February 1983.

[51] D. Hightower, "A Solution to the Line-routing Problem on the Continuous Plane," *Proceedings of the 6th Design Automation Workshop*, (Miami Beach, FL), pp.1-24, 1969.

[52] A. Iosupovici, "A Class of Array Architectures for Hardware Grid Routers, " *IEEE Trans. on Computer-Aided Design*, vol. CAD-5, no.2, pp. 245-255, April 1986.

[53] D. Johannsen, "Bristle Blocks: A Silicon Compiler," *Proceedings of the 16th Design Automation Conference*, pp. 310-313, June 1979.

[54] S. Krishnamurthy and J. JáJá, "Parallel Algorithms for Channel Routing," to appear.

[55] C. Lee, "An Algorithm for Path Connections and its Applications," *IRE Trans. Electron. Comput.*, vol. EC-10, no.3, pp. 346-365, 1961.

[56] T. Ohtsuki, ed., Advances in CAD for VLSI, vol. 4, Layout Design and Verification, New York, Elsevier, 1986.

[57] T. Ohtsuki, M. Tachibana, and K. Suzuki, "A Hardware Maze Router with Rip-up and Reroute Strategies," *Proceedings of the Int. Conf. on CAD*, pp. 220-222, 1985.

[58] O. Olukotun and T. Mudge, "A Preliminary Investigation into Parallel Routing on a Hypercube Computer," *Proceedings of the 24th Design Automation Conference*, pp. 814-820, 1987.

[59] R. Pinter, "River Routing: Methodology and Analysis," *Proceedings of the 3rd CALTECH Conference on Very Large Scale Integration*, pp. 141-163, March 1983.

[60] F. Preparata and W. Lipski, "Optimal Three-layer Channel Routing," *IEEE Transactions on Computers*, C-33, pp. 427-437, 1984.

[61] J. Rose, "LocusRoute:A Parallel Global Router for Standard Cells," *Proceedings of the 25th Design Automation Conference*, pp. 189-195, 1988.

[62] R. Rivest and C. Fiduccia, "A Greedy Channel Router," *Proceedings of the 19th Design Automation Conference*, pp. 418-424, 1982.

[63] J. Reed, A. Sangiovanni-Vincentelli and M. Santomauro, "A New Symbolic Channel Router: YACR2," *IEEE Transactions on CAD*, vol. 4, pp. 208-219, July 1985.

[64] T. Szymanski, "Dogleg Channel Routing is NP-Complete," *IEEE Transactions on CAD*, vol. CAD-4, no. 1, pp. 31-41, Jan. 1983.

[65] A. Seigal and D. Dolev, "Some Geometry for General River Routing," *SICOMP*, 17(3), pp. 583-605, June 1988.

[66] J. Savage and M. Wloka, "A Parallel Algorithm for Channel Routing," *Technical report*, Department of Computer Science, Brown University, January 1989.

[67] M. Vecchi and S. Kirkpatrick, "Global Wiring by Simulated Annealing," *IEEE Transactions on CAD*, vol. CAD-2, no.4, pp. 215-222, Oct. 1983.