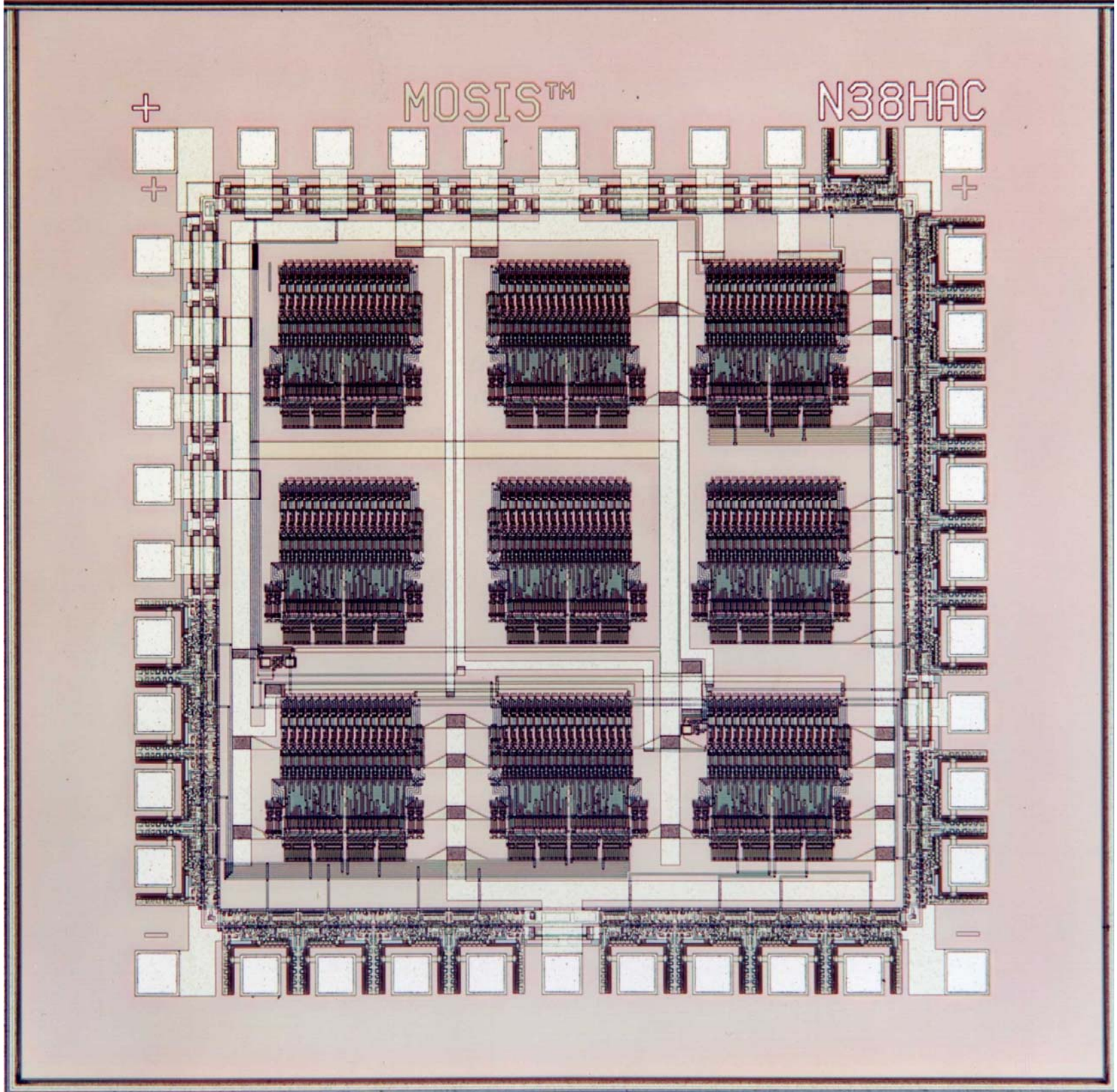
MOSIS™
N38HAC

CHiRPS: a general-area parallel multilayer routing system

R. Venkateswaran
P. Mazumder

Indexing terms: Parallel routing, Cycle-elimination, Grid-graphs, Coterie array

Abstract: A new highly parallel model for concurrent multilayer routing, called CHiRPS, is presented. The nucleus of CHiRPS is a very flexible pathfinder that can be easily configured, even in the presence of obstacles, to generate various commonly used pattern-based routes, such as Steiner trees with single trunk, comb trees, contour-based routes, etc., that span multiple layers simultaneously. The authors employ the concept of a total grid-graph to capture the state of the routing region. The main steps of the pathfinder are based on new parallel algorithms for cycle detection, cycle elimination and tree reduction. The proposed algorithms scale well with increased problem sizes since they require only $O(\log(N))$ time when given a grid-graph with up to N^2 nodes. As such they are good candidates for massively data-parallel machines.

1 Introduction

An efficient routing algorithm is a crucial part of the automated layout design. At the same time, it is desirable that the router be flexible enough to accommodate variations in the nature of the routing regions and the number of routing layers used. Unfortunately, most methods, such as maze routing [1], which are capable of supporting general concurrent layer search operations, are of quadratic time complexity and therefore quite slow for large multilayer problems. Hence, an alternative solution to speed this task appears to be of considerable interest. This paper describes the CHiRPS (configurable highly routable parallel system) model for rapid multilayer routing. As shown in Fig. 1, the underlying hardware (also called coterie hardware) consists of a processor array of simple bit-serial processors operating in a SIMD fashion. This is appropriate since, as has been observed in previous studies [2, 3], at the detailed routing level the problem is inherently fine-grained and therefore not very suitable to parallelisation using a network of complex processors. The key distinguishing feature of the coterie hardware is a dynamically reconfigurable mesh connecting the processors, permitting simultaneous multiple connected groupings. This is achieved simply by opening or closing switches that are provided at each node. Each switch is independently controlled, by the associated processor, based on its local data and the array instructions. Once the switches are set, consensus (SOME/NONE) computations via a wired-OR operation can be independently made in parallel in each group. Often, the PEs in each group share certain common properties; consequently, the groups are also called coteries, giving the hardware its name. Further details of the hardware are given in Reference 4. From a routing point of view, a very useful feature of the coterie-array is that it provides a constant-time hardware solution to the connectivity determination problem, i.e. determining if a given k-pin net can be connected or not. In contrast, most software algorithms only determine this as a consequence of the actual pathfinding operation. For instance, the maze router will declare a connection failure only if at some point it finds no more cells to visit prior to visiting all pins. Hence, the complexity of connection checking is in general the same as that of net routing.

The main contributions of this paper are the development of a new type of parallel detailed routing model called chord routing, in conjunction with the definition and mapping of a planar grid-graph structure called total-graph that has the additional advantage of being a flexible tool for efficient mapping and execution of concurrent multilayer search on the coterie hardware. The chord router is based on new algorithms for cycle-detection, cycle-elimination and tree-reduction on total graphs. It can be easily tailored to generate shortest-length paths (like maze routing), shortest bend paths (like line-probe) or pattern-based routes (like Steiner trees with single trunk, comb trees, contour-based routes), etc. The algorithm runs in time logarithmic to the problem size N, and so is quite fast, especially for large N. In the rest of this paper we assume a gridded routing model, but with no restrictions on the shape or locations of terminals. Nets are routed one-at-a-time and net ordering strategies have been developed based on any prior knowledge of the routing region, such as constraint graphs for regular channel-routing problems.

2 Problem formulation

2.1 Total grid-graph

Assume each routing layer is specified by a layer-graph, denoted as $G_i(V, E_i)$ for the ith layer. Each layer-graph is simply a grid-graph for a particular routing layer with one node for each grid cell and an edge connecting two adjacent cells that are both free, as shown in Fig. 2*a* and

Paper 1619E (C1, C2), first received 24th March and in revised form 19th September 1994

The authors are with the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109-2122, USA

The first author was supported, in part, by an IBM Fellowship.

b. Informally, the total graph for layers 1, ..., k and denoted by TG ($\langle 1, 2, \ldots, k\rangle$), is formed by constructing a union of the edges belonging to the individual layer grid-graphs subject to via-availability constraints, i.e. if any two edges from different layers, say l_1 and $l_2 (l_1 < l_2)$ are incident at any node in the resulting total graph, then the grid cell corresponding to that node must be free on all intermediate layers l, such that $l_1 < l < l_2$. Since edges of the total graph will be tagged by layer numbers, any path embedded on the total graph connecting two points can be automatically mapped into a route for a net with vias implied at those nodes where there is a jump in

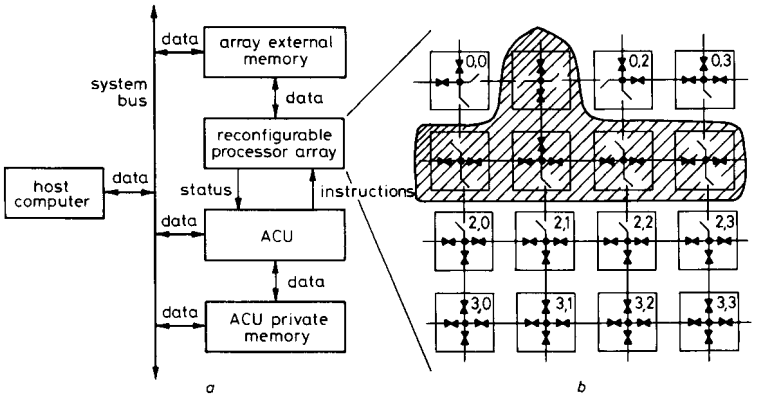

Fig. 1

a System overview *b* Illustration of hardware connection check on coterie mesh

labels. In fact, the chief advantages of the total graph concept are its implicit ability to capture via locations (nodes with multiple associated edge-labels) and also the fact that it can be directly mapped onto a two-dimensional array processor such as the CAAPP. The algorithm for total graph construction is as follows.

Given a labelled graph G(V, E), let L(G, e) denote the label assigned to an edge $e \in E$. If we assume that the edge-label denotes the layer to which it is assigned physically, we have $L(G_i, e) = i$ for all $e \in E_i$. Also, let $v_k = 1$ when cell v is unoccupied in layer k, and assume that a via is always possible between adjacent layers whenever the corresponding grid cells are free. Then, the initial via-permission function for each of the layer graphs is given by

$$VP(G_i, u) = \{j \mid (j \leqslant i) \wedge (\forall k, \text{such that } j \leqslant k \leqslant i, v_k = 1\} \cup \{j \mid (j > i) \wedge (\vee k, \text{such that } i \leqslant k \leqslant j, v_k = 1)\}$$

For instance, if cell v is unoccupied in layers 1, 3, 4, 5 and blocked in cell 2, then $VP(G_1, v) = \{1\}$, $VP(G_2, v) = \{\emptyset\}$ while $VP(G_i, v) = \{3, 4, 5\}$ for $i = 3, 4, 5$. Intuitively, for routing, a path can enter or leave a node v on any layer in VP (G, v). Next, given two labelled graphs, G_x: (V, E_x) and G_y: (V, E_y), define the constrained-prefix sum $G_x + {}_pG_y$ of the two graphs as the graph G′: (V, E′), with

$$E' = E_x \cup \{e(u, v) \mid e \in E_y \wedge L(G_y, e) \in \{VP(G_x, u) \cap VP(G_x, v)\}\}$$

$$L(G', e) = \begin{cases} L(G_x, e) & e \in E_x \quad VP(G', v) = VP(G_x, v) \\ L(G_y, e) & \text{otherwise} \end{cases}$$

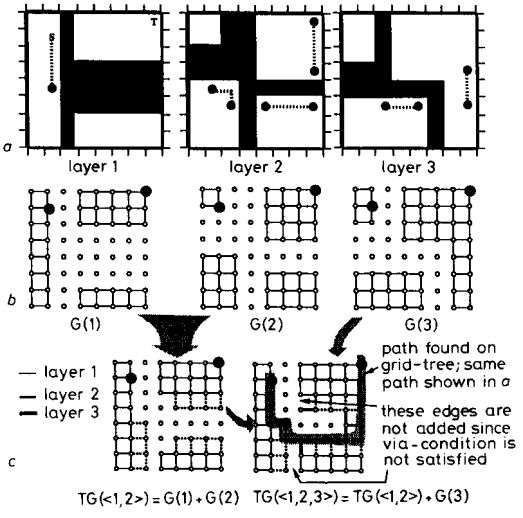

Fig. 2

a 3-layer maze problem
b Individual grid graphs
c Total grid graph

Mathematically, now, the total grid graph TG for k layers is given by

$$TG(\langle 1, 2, \ldots, k\rangle) = G_1 + {}_pG_2 + {}_pG_3 + \cdots + {}_pG_k \quad (1)$$

For actual problems, the total graph construction can be constrained in various useful ways as follows:

1. *Restricted direction model:* For such models, it is sufficient to consider only horizontal or vertical edges from a certain layer during the graph construction. This is a popular approach for detailed routing using three or more layers.

2. *Restricted bend model:* It is also possible to restrict edge addition only to rows and columns containing pins. For channel routing, the effect of such restrictions is to limit doglegs only to columns where terminals for the net occur.

3. *Preferred layer model:* If it is desired to route a certain net entirely in a certain layer(s), it is possible to do so by simply restricting the total-graph construction to use edges from those layer(s).

3 Details of parallel chord router

3.1 Overview

Given a total grid-graph G, the chord router first computes in parallel a minimal set of edges, called a chord-set, to remove from G so as to remove all cycles from G. In other words it computes a spanning tree for G. The name for the router comes from the fact that the edges removed become chords for the tree so constructed. A top-level description of the chord router shows three main steps:

1. Check to see if any cycles exist in G. If yes, go to step 2, else let T = G and go to step 3.
2. Eliminate all cycles in G yielding a tree T.
3. Reduce T to form a required route R, i.e. one in which all leaves are required pins.

We now describe the parallel algorithms behind each of these steps.

3.2 Parallel cycle-detection

An efficient sequential version to determine if cycles exist within a partial-grid G is to keep removing the edges incident on the leaf nodes in G iteratively until no more leaf nodes exist. If this process results in a null graph, then it can be concluded that the original graph was acyclic; if not we can conclude that the graph contains cycles. The parallel version of the algorithm, FrondContract, given in Fig. 3 is similar in nature, except for a

```
proc FrondContract (G: (E, V), V') ≡
begin
  Initially, coterie network 𝒩 = E;
  A = {leaf nodes in 𝒩}\{V'}; /*These nodes broadcast a 1 in step
    2 of the while loop*/
  while (A ≠ ∅) do
      (1) 𝒩' = 𝒩 − {L_u | ((degree (u) ≥ 3) ∨ (u ∈ V'))};
        /*L_u denotes all links attached to node u at that time*/
      (2) VAL = A.RegBroadcast (N')
        /*This step marks nodes in coteries with one or more leaf
          nodes ∉V'*/
      (3) 𝒩 = 𝒩' − {L_u | VAL = = 1)}
        /*All nodes that set VAL to 1 in the above step
          disconnect*/
      (4) A = {leaf nodes in 𝒩}\{V'};
  od end
```

Fig. 3 *Frond contraction*

cut-through feature that allows entire segments (maximal fronds) to be deleted simultaneously. A frond is defined as a path starting from any nonleaf node or a node in V' of the tree and terminates at a leaf node $\notin$V' and has no other fronds emanating from any intermediate node on the path. A frond that cannot be extended any further is called a maximal frond. Each iteration of the while loop can then be viewed as identifying and deleting all maximal fronds in the graph in parallel. An example is given in Fig. 4*a–e*. The above problem can be generalised as follows:

Problem FrondContract (G, V'): 'Given a graph G: (V, E) and a node-set V' $\subset$ V, find a maximal subgraph G_s: (V_s, E_s) of G such that all leaves of G_s are in V''. By maximal, we mean that any edge added to G_s will lead to a leaf-node that is not in V'.

From this formulation, it follows that cycle-detection can be done using V' = $\emptyset$. Furthermore, when V' $\neq \emptyset$ and G is a tree, then G_s is the unique path, if any, in G connecting all nodes in V'. Thus, the same algorithm serves the role of both parallel cycle detection (step 1) and fast parallel tree-reduction (step 3).

Complexity analysis: Note that each iteration of the while loop takes constant time on the coterie-network. Therefore, to analyse the time complexity, it is sufficient to

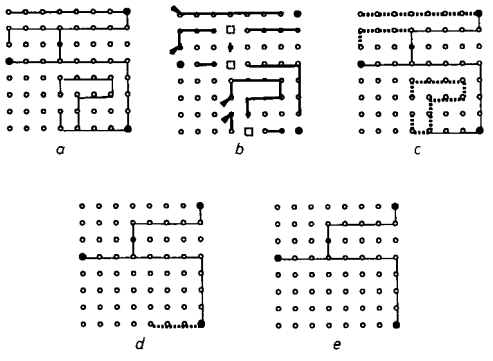

Fig. 4 *Application of frond-deletion for pathfinding on a grid-graph*

prove an upper bound on the total number of iterations that are required for frond-contraction in the worst case. This is stated in the following theorem.

Theorem 1: On the coterie-network model, at most $I = 2 \log (N + 1) - 1$ iterations are needed for graph-contraction on an $N \times N$ grid-graph G.

Proof: Please refer to Reference 5 for a proof. Further, it can be shown that the above bound is tight for a complete binary tree of height $2 \log (N) - 1$. □

3.3 Parallel cycle-elimination

In a sense, what we are seeking to do is to find a spanning tree for any given total-graph. Since these graphs are quite sparse, algorithms for spanning trees, that require the construction of adjacency matrices, are not very useful owing to the size of the grid-graphs. Other parallel algorithms [6, 7] given in the literature for parallel spanning tree generation on general graphs also suffer from the same. These algorithms are also area-inefficient, in the sense that they require N^4 processors for an N^2-node graph problem. On the other hand, the TreeGen algorithm, presented in this Section, incorporates knowledge of the total-graph structure; hence, for an $N \times N$ grid graph, it runs in logarithmic time, using only N^2 processors and constant space per processor. First, we define some terms.

Define a *trivial cycle* as a cycle formed by edges connecting any four neighbouring nodes in the grid with coordinates, say, $\langle i, j \rangle$, $\langle i + 1, j \rangle$, $\langle i + 1, j + 1 \rangle$ and $\langle i, j + 1 \rangle$. Node $\langle i, j \rangle$ is said to *anchor* this trivial cycle. All other cycles are termed *nontrivial* (see Fig. 5). Suppose G: (V, E) is a given total grid-graph and G'(V, E') is a subgraph of G with only nontrivial cycles. Let T: (V, E^T) be a spanning tree for G'. Then each edge *e* in the set C: {E' − E^T} can be called a *chord* since adding it to E^T will introduce a cycle in T. The set C itself will be called a *chord-set*. Every spanning tree has a different but unique chord-set. Consequently, identifying the chord-set is the same as constructing the spanning-tree. This is the premise on which the algorithm presented below is based. Assume variables *D*, *U*, *R* and *L* are initially set to 1(0) at

a node, if an edge is present (absent) between the node and its immediate Down, Up, Right or Left neighbour, respectively. The notation R.East( ) is used to refer to the R value stored at the East neighbour. The same applies for other directions as well.

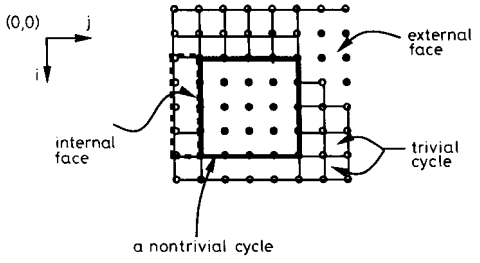

Fig. 5 *Illustration of cycle and region definitions*

Procedure TreeGen: (G):
Input: A total grid graph G (V, E)
Output: A spanning tree T (V, E^T) for G.

Stage 1: The first step is to delete trivial cycles. Each node computes $D \wedge R \wedge D$.East( ) $\wedge R$.South( ), which is 1 only when there exists a trivial cycle anchored at that node. The PEs that anchor a trivial cycle set their value of D to 0. This yields the graph G′: (V, E′) that contains no trivial cycles.

Stage 2: Check to see if G′ has any other cycles. As stated earlier, this can be verified by checking if G″: (V, E″) = FrondContract (G′, V′ = ∅) returns an empty graph. If so, the procedure can terminate and G′ be returned as the required spanning tree. Otherwise, we continue by identifying a chord-set C for G″ such that G^C: (V, E″ − C) is a spanning tree of G.

Stage 3: Clearly a necessary condition for C to be a chord-set of G″ is that it must include at least one edge from each cycle in G″. Since each cycle corresponds to a face, we begin by forming separate connected regions within the coterie mesh, one per face, such that each PE belongs to at most one region. The latter requirement is needed for SIMD operation since otherwise processors can vary in terms of the number of different cases each of them has to consider. The actual code needed to set up the regions turns out to be quite simple and is given below:

1. CONU = ¬U; COND = ¬D;
   CONL = ¬L; CONR = ¬R; /*Initial network is the complement of G″*/
2. $X_1 = U \wedge D$; $X_2 = R \wedge L$;
   $X_3 = \neg R \wedge \neg$R.South( ); $X_4 = \neg D \wedge \neg$D.East( );
3. *if* X_1 *then* CONL = 0; *fi*/*DISJ1*/
4. *if* X_2 *then* CONU = 0; *fi*/*DISJ2*/
5. *if* X_3 *then* COND = 1; *fi*/*JOIN1*/
6. *if* X_2 *then* CONR = 1; *fi*/*JOIN2*/
   EqualizeLinks( );

Let us analyse what each step accomplishes. In step 1, an initial network $\mathcal{N}$ is set up based on four CON plane variables; where CONU is 1 at a node to indicate the presence of the up-connection at that node, and so on (Fig. 6*b*). However, adjacent faces continue to remain connected in $\mathcal{N}$. To isolate them from each other, we apply the DISJ1 and DISJ2 steps (3 and 4). The first disconnects nodes that lie on the right border of the cycle from its interior (Fig. 6*c*) while the second disconnects nodes that lie on the bottom border of a cycle from the interior (Fig. 6*d*). In a similar vein, whenever opposite borders of a cycle lie on adjacent rows (columns), the JOIN step, (5 and 6) add links to $\mathcal{N}$, as specified, to ensure that each face results in exactly one component (Fig. 6*e–f*).

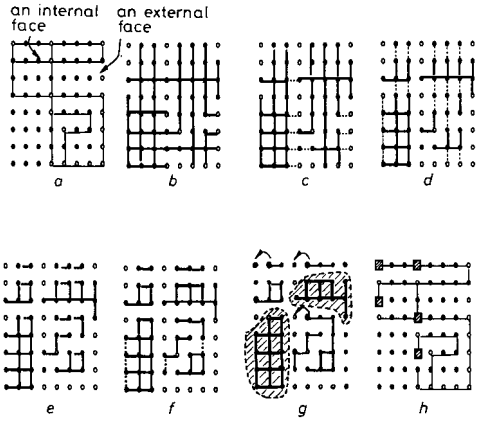

Fig. 6 *Illustration of tree-generation*
a Given grid graph G with 5 internal and 2 external faces
b Initial network N′. Link added wherever there is a missing grid edge in G
c DISJ1: Disconnect West link from (*b*) IF PE has both north & south edges incident in *a*. Edges removed by DISJ1 shown dotted. This isolates nodes on right borders of cycles from its interior
d DISJ2: Disconnect North link from *c* IF PE has both east & west edges incident in *a*. Edges removed by DISJ2 shown dotted. This isolates nodes on bottom borders of cycles from its interior
e JOIN1: Adds East link to *d* IF PE(*x*) & its East neighbour both have missing south edges in *a*. Edges added by JOIN1 shown dotted. This connects horizontally oriented cycles of spread 1
f JOIN2: Adds South link to *e* IF PE(*x*) & its south neighbour both have missing east edges in *d*. Edges added by JOIN2 shown dotted. This connects vertically oriented cycles of spread 1
g A region is deactivated if at least one node in it has a boundary edge (w.r.t. to entire array) missing. Shaded areas show two such regions. Each corresponds to an exterior face. Leftmost nodes within each region selected. If selected PE does not have a south edge in *a*, then the PE to its left is marked (see arrows)
h Acyclic tree T for G. Down edges from all nodes marked in previous step (shown hatched) constitute set CL for this case

Stage 4: Having set up the network, and consequently components, we next mark the activity status for the components as follows:

1. Isolated nodes are marked inactive unless $D = 0 \vee R = 0$, i.e. the corresponding right or down edge is missing in G″.

2. All components that correspond to exterior faces are also deactivated. This is done as follows. First, each cell independently sets a variable $B = 1$ if it determines, based on their addresses and the grid-size, that it lies on the border of the grid. Next, all cells broadcast this value within their component with a RegBroadcast operation. Subsequently, all nodes that receive a 1 in the broadcast step deactivate themselves. Fig. 6*g* shows two regions deactivated since they correspond to external faces.

3. All remaining PEs are presumed to be active. In Fig. 6*g*, only 5 active components remain each of which corresponds to one internal face in G″.

Stage 5: At this stage, it is possible to compute several chord-sets. In particular, chord-set CL (CR) is determined as follows. In parallel, for each active component, select randomly a PE with smallest (largest) column index. For each marked PE $\langle i, j \rangle$; if the PE has a

DOWN edge in G', then add that edge to CL (CR); else add the DOWN edge of the PE to the immediate West (East), namely, $\langle i, j-1\rangle$ to CL ($\langle i, j+1\rangle$ to CR). Return either G^{CL}: (V, E″ − CL) or G^{CR}: (V, E″ − CR) as the required tree.

Complexity analysis: The TreeGen algorithm also takes O(log (*N*)) time for an $N \times N$ mesh. This is not hard to see. Stages 1, 3 and 4 of the algorithm are such that each PE performs only local computations which take constant time. Step 2 takes in the worst case O(log (*N*)) time from theorem 1. Stage 5 requires a RegSelMin operation to select a leftmost (rightmost) PE cell which needs O(log (*N*)) for data that are at most log (*N*) bits long. □

A formal proof of correctness for the above algorithm can be found in Reference 5. In reality, stage 2 is actually not required for the correct operation of the TreeGen algorithm. The operations described in stages 3–5 equally work well when directly applied to G. However, there does not seem to be any easy characterisation of the resulting coterie components and consequently the proof of correctness. However, this does reduce the run time and so is of practical interest.

3.4 Extensions to tree generation

Note that chord-sets {CL} and {CR} were constructed by uniformly choosing the leftmost (rightmost) PE of each coterie. On a complete grid-graph, this leads to so-called comb trees with a single vertical trunk at column 0 ($N-1$) with horizontal row-wise connections to all remaining nodes in G (see Fig. 7*a* and *b*). A generalisation of the tree-generation mechanism is to move the vertical trunk to an intermediate column, say *j*, where $0 < j < N-1$. Similarly, a tree having one horizontal trunk at row *j* with possibly many vertical branches emanating from it can be constructed by first rotating the input graph G by 90°. We call such trees single-trunk Steiner trees (STST). Construction of an STST requires the same TreeGen procedure as before, with the following modifications:

1. Divide the array conceptually into a left half (all nodes with $ColIndex < j$) and a right half (all nodes with $ColIndex > j$).
2. Interchange the roles of R with L, and East( ) with West( ) in Stages 1, 3 and 4 of Procedure TreeGen for the right half of the array. Also in stage 1, for the right half of the array, define the top-right node of a trivial cycle as its anchor. The operations for the left half remain as before.
3. Compute both CL and CR in stage 5. Now classify a member-edge of CL as type-1 if it is on the left border of a cycle that lies entirely to the left of column $j+1$, or as type-2 if it is on the left border of a cycle that lies entirely to the right of column $j-1$, with the rest being termed as type-3. Similarly, a member-edge of CR is termed as type-1 if it is on the right border of a cycle that lies entirely to the right of *j*, or as type-2 if it is on the right border of a cycle that lies entirely to the left of *j*, with the rest being termed again as type-3. The following then yields two new chord-sets, CLSET(*j*) and CRSET(*j*), that allow for location of the vertical trunk at any column *j*:

$$\text{CLSET}(j) = \{\text{all type-1 edges in CL and CR}\} \cup \{\text{all type-3 edges in CL}\}$$

$$\text{CRSET}(j) = \{\text{all type-1 edges in CL and CR}\} \cup \{\text{all type-3 edges in CR}\}$$

Note that the two differ only if there exist cycles (obstacles) that span column *j*. In that case, the resulting trees will contour either along the left border of the obstacle (CLSET) or the right border (CRSET). For instance, the route shown in Fig. 7*b* can be understood as an STST generated with CRSET (0). Consequently, the route contours along the right side of the blocks present along column 0. In general, the idea of STST can be extended to multiple-trunk Steiner trees (MTST) having one primary trunk with one or more secondary trunks perpendicular to it (see Fig. 7*d*). Thus, the chord-set idea appears to be quite general and useful since many of the resulting routes have been found quite relevant to channel, switchbox and area routing problems.

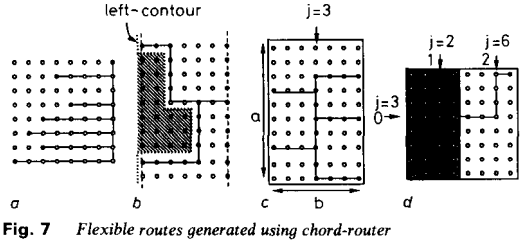

Fig. 7 *Flexible routes generated using chord-router*
a East Comb tree generated using chord-set {CL}
b Contour-based route generated with {CR}
c Single-trunk Steiner tree using CLSET (3) with length 22
d Multitrunk Steiner tree: MTST (3, 2, 6) using two zones with length 18

4 Implementation details

4.1 Overview

The software for our router has currently been coded in C++ and is linked with the IUA Class Library (ICL) [8] that provides all kinds of array data structures and communication functions including region broadcasts. The router is based on the algorithms described in the preceding sections. Various options and heuristic strategies have also been incorporated into our router to account for routing conflicts amongst the nets, thereby enabling it to handle various types of routing problems in a consistent uniform manner. For instance, some deal with the order in which layers are processed during grid-graph construction, while others determine the type of chord set to use in generating the routing tree, or the reservation of territories for pins of unrouted nets, or bounds on wire-lengths, etc. Each such policy can be simply controlled by means of a user-specified parameter which is specified in a separate options file. Because of space limitations only an overview is given in this paper; further details of each of these steps will be a topic of a future publication. Postprocessing options include wire-length minimisation and via minimisation routines.

Another factor which is of great concern for sequential net-at-a-time is the order in which the nets are scheduled for routing. For special routing problems such as channels, we employ a simple scheduling strategy that is derived from the underlying vertical constraint graph (VCG). Note that the VCG is a direct graph with one node per net and a directed edge from node n_i to n_j whenever there exists a column where the terminal for net *i* is above that for net *j*. The idea with the VCG-based ordering is to schedule nodes with indegree 0 first. Subsequently, these nodes are deleted from the VCG. This process can lead to a new set of nodes with indegree 0, and the process is iterated until either all nets are routed

or until all nodes have nonzero indegrees. The latter case occurs when there are cycles in the VCG. In such a case, we choose a node with minimum indegree (primary key) having the maximum outdegree and remove its outgoing edges. This has the effect of ignoring vertical constraints for the net and at the same time minimising the number of such violations. This process is repeated until a breakthrough occurs, which is the case when the indegree of at least one node becomes zero. Ties are resolved in favour of shorter nets. Each net is routed using the CRSET (0) chord set strategy, which has the effect of routing it as close to the left border as possible.* An exception to this rule is that nets which have terminals only along the right border are routed using the CLSET (C_{max}) chord-set instead, where C_{max} is the column index of the rightmost column.

For general problems, the role of a scheduler is less well defined. We have met with reasonable success using a simple-minded heuristic based on the number of pins and Manhattan wire length. Shorter nets and those with fewer pins are routed first. The cut pattern used is either CLSET(c) or CRSET(c) with the value of c set to the median of the column indexes for each pin of the net. In Reference 5, we have proved that the median yields the best route in an expected sense. For these problems, the router also considers rotating the boundary region so as to change the orientation of the primary trunk.

4.2 Experimental results

For these experiments we used a relaxed-HVH layering model wherein the directional restrictions are enforced strictly initially and relaxed later in case a net cannot be routed. Such constraints are easily achieved in our algorithm by restricting edge addition in the total graph construction phase to conform with the layer directionality. For instance, for the HVH constraint, only horizontal edges are added from G_1 and G_3, and only vertical edges added from G_2. Owing to the grid nature of our model, we first determine the expected channel density w. Let d denote the horizontal channel density, i.e. the maximum number of nets crossing any column and v denote the length of the longest path in the vertical constraint graph for the channel. Then $w = \max(d/n_h, v)$, where n_h is the number of H and B layers.† Table 1 gives some of the results of running the chord router on several benchmark channels. Fig. 8 shows the routing solution for one problem. We observe that, due to the relaxed HVH model, some channels can in fact be routed using fewer tracks than the initial estimate w.

Table 2 compares the solution quality of our router with those of five other 3-layer routers. Note that the routers being compared with are all high-quality channel-specific algorithms, whereas our router is based on parallel algorithms developed with the intention of applying it for a wide variety of routing problems, besides just channel routing. For the difficult channel problem, our method needs more tracks than the best known solution; however, it uses relatively less wire-length after accounting for the differences owing to increased wirelengths along the width of the channel. A general observation has been that, for channels with 50 or fewer nets, the solution of our parallel router, in terms of channel width, is better than or the same as existing channel routers; but for larger problems, due to the sequential net routing and general-purpose nature of this router, the results are off by a small number. This seems the price to pay for being able to use the same router on several different problems such as: two-layer (in general, k-layer) channels, switch-boxes, and general area problems; problems with irregular boundaries or ones with pre-wired nets or other obstacles, etc. For instance, a solution to a larger 2-layer 128 × 128 general area routing problem, using the chord router, is shown in Fig. 9. Here, the first layer is blocked at several places as shown, and the second layer is assumed to be entirely available for routing purposes. The netlist used is the same as in Reference 3. The overall wirelength, 781, is very close to the optimal, 759, derived assuming no blockages. In fact, these results are better than the results obtained with a conventional maze router (821) owing mainly to incorporation of pin-reservation and pattern restriction strategies. The latter resulted in shorter routes for two nets marked 'a' and 'b'.

Table 1: Results for some 3-layer benchmark channels

Channel	Nets	d, v	w	#Trks	Tot WL	#Vias
yosh	10	5, 3	3	2	53	8
burs	10	4, 8(cyc)	4	3	72	9
ex1	34	12, 7	7	6	442	50
ex3a	30	15, 4	8	8	781	71
ex3b	47	17, 9	9	9	1238	97
ex3c	54	18, 6	9	10	1652	137
diff	72	19, 9	10	13	4615	332

Table 2: Comparison with other routes

Example	This paper	Cham [9]	C&L [10]	B&S [11]	E&D [12]	Robust [13]
ex3a	8	8	8	8	8	8
ex3b	9	10	10	10	10	9
ex3c	10	10	9	10	9	10
diff	13	11	14	11	13	10

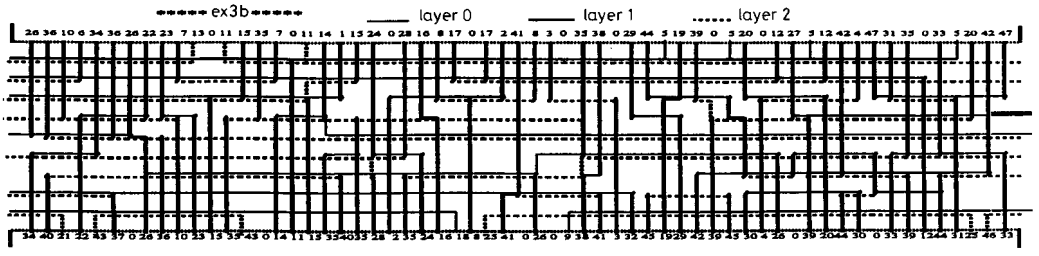

Fig. 8 *9-track 3-layer solution for channel ex3b*

5 Conclusions

In this paper we have developed a new framework for general-area multilayer routing that is both fast and flex-

* Note that, to retain compatibility with the path-router, we orient our channels vertically; consequently left (right) terms are used in place of the usual top (bottom)

† In the layering model used, each layer can be tagged as H (strictly horizontal), or V (strictly vertical) or B (both-direction wiring allowed)

ible. A major contribution of our work has been in the development of very highly parallel algorithms for cycle detection and elimination on a special class of routing graphs called total-graphs. The latter was also shown to

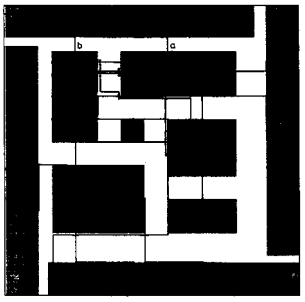

Fig. 9 *Parallel maze routed general-area example*

be an effective and flexible tool for multilayer routing area representation since it not only allows a planar representation of a three-dimensional space, but can also be controlled to account for layering and other strategies. The models are robust in the sense that the same algorithms are used to route different kinds of routing problems, varying in not only the shape of the routing region but also in the number of routing layers.

6 References

1 LEE, C.Y.: 'An algorithm for path connections and its applications', *IRE Trans. Electron. Comput.*, 1961, pp. 346–365
2 HONG, S.J., and NAIR, R.: 'Wire-routing machines — new tools for VLSI physical design', *Proc. IEEE*, 1983, **71**, pp. 57–65
3 VENKATESWARAN, R., and MAZUMDER, P.: 'A hexagonal array machine for multi-layer wire routing', *IEEE Trans.*, 1990, **CAD-9**, pp. 1096–1112
4 WEEMS, C.C., and RANA, D.: 'Reconfiguration in the low and intermediate levels of the image understanding architecture', *in* LI, H., and STOUT, Q. (Eds.): 'Reconfigurable massively parallel computers' (Prentice-Hall, 1991), chap. 4, pp. 88–105
5 VENKATESWARAN, R., and MAZUMDER, P.: 'Highly parallel routing algorithms on a reconfigurable processor array'. Tech. rep., University of Michigan, Department of EECS, 1993
6 SAVAGE, C., and JAJA, J.: 'Fast efficient parallel algorithms for some graph problems', *SIAM J. Comput.*, 1981, **10**, pp. 682–691
7 ATALLAH, M.J., and KOSARAJU, S.R.: 'Graph problems on a mesh-connected processor array', *J. ACM*, 1984, **31**, pp. 649–667
8 BURRILL, J.H.: 'The class library for the IUA' (Amerinex AI, Inc., Amherst, MA, 1992)
9 BRAUN, D., BURNS, J., DEVADAS, S., MA, H.K., MAYARAM, K., ROMEO, F., and SANGIOVANNI-VINCENTELLI, A.: 'Techniques for multilayer channel routing', *IEEE Trans.*, 1988, **CAD-7**, pp. 698–711
10 CONG, J., WONG, D.F., and LIU, C.L.: 'A new approach to three or four layer channel routing', *IEEE Trans.*, 1988, **CAD-7**, pp. 1094–1104
11 BRUELL, P., and SUN, P.: 'A greedy three layer channel router'. Digest of International Conference on *Computer design*, November 1985, pp. 298–300
12 ENBODY, R.J., and DU, H.C.: 'Near optimal n-layer channel routing'. *Design automation* conference, 1986, pp. 708–714
13 YOELI, U.: 'A robust channel router', *IEEE Trans.*, 1991, **CAD-10**, pp. 212–219