TECHNICAL
RESEARCH
REPORT

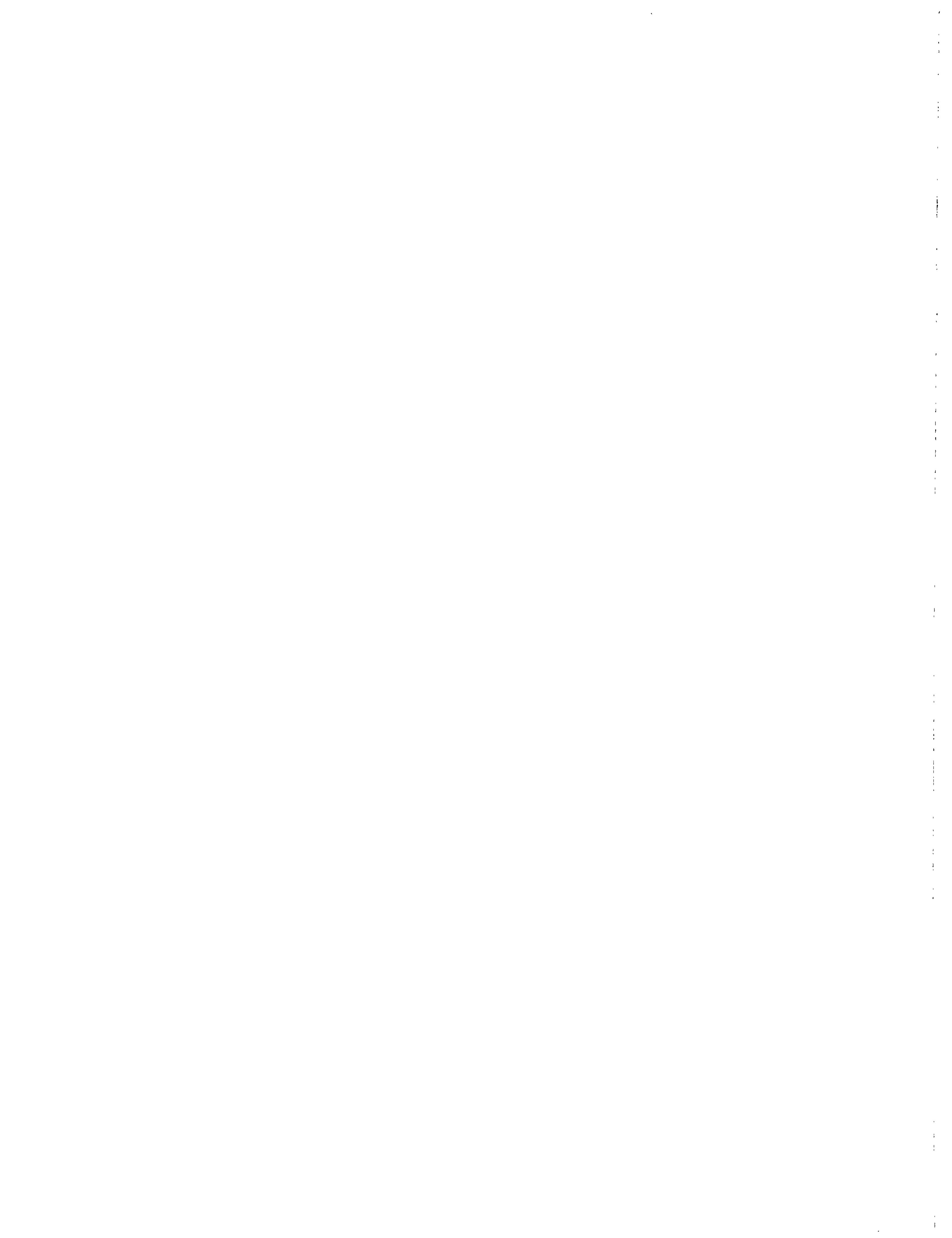
SRC TR 88-62

# Parallel Algorithms for Several VLSI Routing Problems

by

S.-C. Chang

SYSTEMS RESEARCH CENTER
UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND 20742



# PARALLEL ALGORITHMS FOR SEVERAL VLSI ROUTING PROBLEMS

by
Shing-Chong Chang

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1988

Advisory Committee:

Professor Joseph JáJá
Professor Hung C Lin
Professor P.S. Krishnaprasad
Associate Professor A. Yavuz Oruc
Professor Azriel Rosenfeld

# Abstract

Title of Dissertation: Parallel Algorithms for Several VLSI Routing Problems

Shing-Chong Chang, Doctor of Philosophy, 1988

Dissertation directed by: Joseph JáJá, Professor, Electrical Engineering

We consider several basic problems in VLSI routing such as river routing between rectangles, routing within a rectilinear polygon, wiring module pins to frame pads and channel routing in the knock-knee model. The known strategies to handle these problems seem to be inherently sequential. We develop new techniques that lead to efficient parallel algorithms. Our basic model of parallel processing is the CREW-PRAM. All our algorithms use $p$ processors, where $1 \leq p \leq n$, $n$ is the input length, and run in time $O(\frac{n}{p} + \log n)$ or $O(\frac{n \log n}{p} + \log^2 n)$. Our algorithms have fast implementations on other parallel models such as the mesh and the hypercube.

For the river routing problem between rectangles, we have derived $O(\frac{n}{p} + \log n)$ algorithms to determine the characteristic bendpoints and the minimum separation, and $O(\frac{n \log n}{p} + \log^2 n)$ algorithm to solve the offset problem. To solve the routing problem within a rectilinear polygon, we convert the layout problem into a geometric problem of finding the union of rectilinear polygons and develop $O(\frac{n}{p} + \log n)$ algorithms for both the deatiled routing and routability testing problems.

For the problem of wiring module pins to frame pads, we develop a new strategy that leads to an $O(\frac{n}{p} + \log n)$ algorithm. We have also developed $O(\frac{n \log n}{p} + \log^2 n)$ algorithms to detect routability and to minimize the total wire length.

To solve the channel routing problem in the knock-knee model, we have developed an $O(\frac{n}{p} + \log n)$ parallel algorithm to partition the $n$ input nets into $d$ chains, where $d$ is the density of the problem. This new technique could replace the left edge or the greedy strategy for channel routing. Then we have developed $O(\frac{n}{p} + \log n)$ algorithms to modify the above chains and to generate the optimal wiring. For

the three layer assignment problem, we have developed $O(\frac{n}{p} + \log n)$ algorithms to modify the above layout and do the layer assignment such that the layout is three layer wirable.

# PARALLEL ALGORITHMS FOR SEVERAL VLSI ROUTING PROBLEMS

by
Shing-Chong Chang

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1988

Advisory Committee:

Professor Joseph JáJá
Professor Hung C Lin
Professor P.S. Krishnaprasad
Associate Professor A. Yavuz Oruc
Professor Azriel Rosenfeld

© Copyright by
Shing-Chong Chang
1988

To My Parents

# ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my advisor, Dr. Joseph JáJá, for introducing me to this interesting and challenging research area, for his guidance and encouragement during this work, and for his financial support through the years of my graduate education.

I would like to thank the other members of my committee, Dr. Hung C Lin, Dr. P.S. Krishnaprasad, Dr. A. Yavuz Oruc, and Dr. Azriel Rosenfeld, for their constructive criticism. Special thanks to Dr. Kazuo Nakajima, Dr. Dave Mount, and Dr. Chuck Silio for their encouragement and valuable instructions in all phase of my doctor program.

I am also grateful to the Electrical Engineering Department, Systems Research Center and Institute for Advanced Computer Studies (UMIACS), for the resources provided. Particular thanks go to Dr. György Fekete who help me a lot while managing the facilities in the VLSI System Lab. Acknowledgements are also due to all my friends who have made my stay at Maryland pleasant and enjoyable.

Finally, I must thank my wife and son, for their unfailing encouragement, support, patience and love. I also owe a great deal of gratitude to my parents in Taiwan, Republic of China and would like to dedicate this dissertation to them for their encouragement in my pursuing higher education.

# Table of Contents


**Chapter 1 Introduction** . . . . . 1

1.1. Motivation . . . . . 1

1.2. Preliminary Definitions . . . . . 2

1.3. Parallel Computation Models . . . . . 9

1.3.1. PRAM Model . . . . . 9

1.3.2. Mesh-Connected Model . . . . . 10

1.3.3. Hypercube Model . . . . . 11

1.3.4. Perfect Shuffle Model . . . . . 12

1.3.5. Shuffle-Exchange Network . . . . . 12

1.3.6. Cube-Connected-Cycles . . . . . 13

1.4. Basic Parallel Techniques . . . . . 14

1.5. Summary of Results . . . . . 21

**Chapter 2 VLSI Routing** . . . . . 25

2.1. Introduction . . . . . 25

2.2. River Routing Between Rectangles . . . . . 27

2.3. Routing Within a Rectilinear Polygon . . . . . 29

2.4. Wiring Module Pins to Frame Pads . . . . . 30

2.5. Channel Routing in the Knock-Knee Model . . . . . 31

**Chapter 3 River Routing Between Rectangles** . . . . . 37

3.1. Introduction . . . . . 37

3.2. Definitions . . . . . 38

3.3. The Separation Problem . . . . . 38

3.4. The Offset Problem . . . . . 42

**Chapter 4 Routing Within a Rectilinear Polygon** . . . . . 46

4.1. Introduction . . . . . 46

4.2. Definitions . . . . . . . . . . . . . . . . . . . . . . . 47

4.3. Detailed Routing . . . . . . . . . . . . . . . . . . . . 47

4.4. Routability Testing . . . . . . . . . . . . . . . . . . . 55

**Chapter 5 Wiring Module Pins to Frame Pads** . . . . . . . . 63

5.1. Introduction . . . . . . . . . . . . . . . . . . . . . . 63

5.2. Definitions . . . . . . . . . . . . . . . . . . . . . . . 64

5.3. Routability Testing . . . . . . . . . . . . . . . . . . . 65

5.4. Minimizing Wire Length . . . . . . . . . . . . . . . . . 76

**Chapter 6 Channel Routing in the Knock-Knee Model** . . . . 80

6.1. Introduction . . . . . . . . . . . . . . . . . . . . . . 80

6.2. Definitions . . . . . . . . . . . . . . . . . . . . . . . 81

6.3. Channel Routing . . . . . . . . . . . . . . . . . . . . 83

6.4. Layer Assignment . . . . . . . . . . . . . . . . . . . . 93

**Chapter 7 Conclusion** . . . . . . . . . . . . . . . . . . . . 106

7.1. Implementation . . . . . . . . . . . . . . . . . . . . . 106

7.1.1. The Connection Machine System . . . . . . . . . . 106

7.1.2. Programming Model and *Lisp . . . . . . . . . . 108

7.1.3. Discussion . . . . . . . . . . . . . . . . . . . . . 109

7.2. Concluding Remarks . . . . . . . . . . . . . . . . . . . 110

**Bibliography** . . . . . . . . . . . . . . . . . . . . . . . . . 117

# List of Tables


Table 7.1 The Running Time of Channel Routing Program . . . . . 110

Table 7.2 Summary of Results on the CREW-PRAM Model . . . . . 115

Table 7.3 Summary of Results on the Other Models . . . . . . . . 115

# List of Figures


Figure 1.1 Basic River Routing Problem between Rectangles . . . . . . . 3

Figure 1.2 Routing within a Rectilinear Polygon . . . . . . . . . . . . 4

Figure 1.3 Routing Between Module Pins and Frame Pads . . . . . . . . 5

Figure 1.4 Types of Shared Grid Points in the Knock-knee Model . . . . 6

Figure 1.5 Channel Routing in the Knock-knee Model . . . . . . . . . . 6

Figure 1.6 The Structure of the PRAM Model . . . . . . . . . . . . 10

Figure 1.7 The Structure of a Mesh . . . . . . . . . . . . . . . . . . 11

Figure 1.8 The Structure of a Hypercube . . . . . . . . . . . . . . . 11

Figure 1.9 The Structure of a Perfect Shuffle Model . . . . . . . . . . 12

Figure 1.10 The Structure of a Shuffle-Exchange Network . . . . . . . . 13

Figure 1.11 The Structure of a Cube-Connected-Cycles . . . . . . . . 14

Figure 1.12 Parallel Prefix Computation by Using Path Doubling . . . . 17

Figure 2.1 GI and GO Routing Strategies . . . . . . . . . . . . . . . 32

Figure 2.2 Possible Wire Routed by Sequential CRP Algorithm . . . . 33

Figure 2.3 The Layout of a Channel Routing Problem . . . . . . . . . 33

Figure 2.4 Mapping between Knock-knee and Diagonal Diagram . . . . 34

Figure 2.5 Forbidden Patterns . . . . . . . . . . . . . . . . . . . . . 34

Figure 2.6 Changing in the Wiring of Two Crossing Nets . . . . . . . 36

Figure 2.7 Diagonal Diagram and Legal Partition . . . . . . . . . . . 36

Figure 3.1 Basic River Routing Problem . . . . . . . . . . . . . . . . 39

Figure 3.2 Minimum Separation with Offset 7 . . . . . . . . . . . . . 43

| | | |
|---|---|---|
| Figure 4.1 | Routing Around a Rectangle Boundary | 49 |
| Figure 4.2 | The Union of Bounding Perimeters | 53 |
| Figure 4.3 | Intersection Between Rays and Block Contours | 60 |
| Figure 5.1 | Illustration of Different Types of Nets | 67 |
| Figure 5.2 | Determination of Intermediate Terminals | 69 |
| Figure 5.3 | Case 1 Intersection | 72 |
| Figure 5.4 | Case 2 Intersection | 73 |
| Figure 5.5 | Case 3 Intersection | 73 |
| Figure 5.6 | Intersections Determined by Test Pairs | 75 |
| Figure 5.7 | Left and Right Wells | 77 |
| Figure 5.8 | Transformation of Well with no Obstacles | 78 |
| Figure 5.9 | Transformation of Well with an Obstacle | 79 |
| Figure 6.1 | Type of Shared Grid Points | 82 |
| Figure 6.2 | Forbidden Patterns | 83 |
| Figure 6.3 | A Channel Routing Problem | 85 |
| Figure 6.4 | The Chains Created by *Create Chains* | 85 |
| Figure 6.5 | Possible Successors of Two Nets | 88 |
| Figure 6.6 | New Chains Generated by *Modify Chains* | 89 |
| Figure 6.7 | Forms of Groups in the Proof of Lemma 6.2 | 90 |
| Figure 6.8 | Possible Detours of Nets | 92 |
| Figure 6.9 | Layout, Diagonal Diagram and Constraint Graph | 92 |
| Figure 6.10 | Configurations for Forbidden Columns | 95 |
| Figure 6.11 | Trivial Reference Lines | 97 |

Figure 6.12 Overlap Reference Lines . . . . . . . . . . . . . . . . . 97

Figure 6.13 Disjoint Reference Lines . . . . . . . . . . . . . . . . . 98

Figure 6.14 Inclusion Reference Lines . . . . . . . . . . . . . . . . . 98

Figure 6.15 Transformation of $I_2$ Reference Lines . . . . . . . . . . . . 100

Figure 6.16 Transformation of $I_4$ Reference Lines . . . . . . . . . . . . 101

Figure 6.17 Transformation of $D_1$ Reference Lines . . . . . . . . . . . 101

Figure 6.18 Maximal Chain of $D_1$'s . . . . . . . . . . . . . . . . . . 102

Figure 6.19 Possible Wiring for Case 2 of Lemma 6.6 . . . . . . . . . . 103

Figure 6.20 Possible Diagonals between $L_i$ and $L_{i-1}$ . . . . . . . . . . 104

Figure 6.21 Changes in the Wiring of $N_{16}$ and $N_{21}$ . . . . . . . . . . . 105

Figure 6.22 Final Layout, Diagonal and Constraint Graph . . . . . . . 105

Figure 7.1 A Program Segment of *Create Chains* . . . . . . . . . . . . 111

Figure 7.2 Layout Generated by Routing Program . . . . . . . . . . . 112

Figure 7.3 Variation of Running Time vs. Net Number . . . . . . . . . 113

CHAPTER
ONE

# INTRODUCTION

## 1.1 Motivation

The recent advances in semiconductor technology allows the fabrication of highly complex systems on single chips ([MeC],[WeE]). As the complexity of these electronic circuits increases, sophisticated software tools are needed to successfully design such systems ([Rub]). Although Computer-Aided Design (CAD) systems have existed for a long time, many of them are inadequate for the current task and new concepts and systems are needed to cope with the increasing complexity ([Tho],[Ull],[Rub]). In particular, the routing phase is a critical and a time-consuming part of the overall design process ([Oht]). Unfortunately, it turns out that most routing problems are NP-complete and hence no efficient solutions seem to be likely. There are few exceptions, however. For example, various river routing (one-layer) problems, the two-layer channel routing with no constraints, and few routing problems in the knock-knee model are known to have efficient solutions ([DKS],[MeP],[Oht],[Pin],[PrL]).

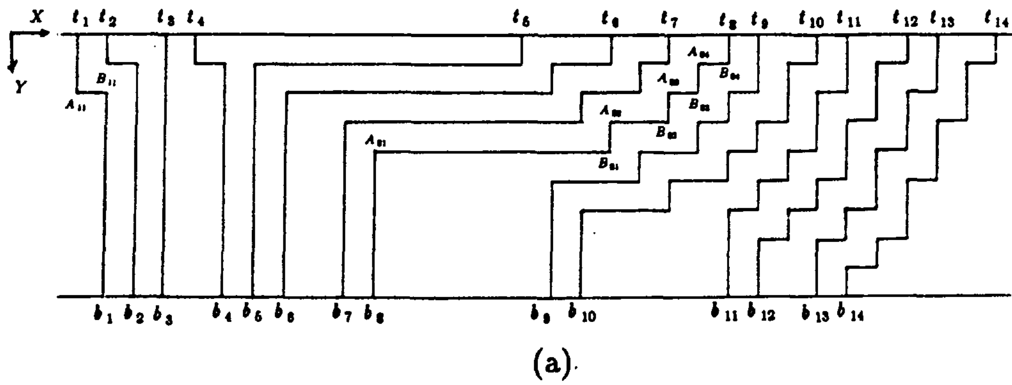
Since the routing phase takes a major portion of the design process, there is an increasing number of problems to be overcome in producing usable VLSI circuits with a much shorter design time. The major thrust of this thesis is

the development of a good set of techniques to obtain fast and efficient parallel routing algorithms. In particular, we consider several problems in VLSI routing such as river routing between rectangles, routing within a rectilinear polygon, wiring module pins to frame pads and channel routing. Our results here are of both theoretical and practical interest. We develop new techniques that lead to efficient parallel routing algorithms. These techniques are also useful for other routing problems ([KrJ]). A number of our algorithms have been implemented on the Connection Machine to form the central part of the Parallel CAD tool currently being developed at the University of Maryland VLSI Design Laboratory.

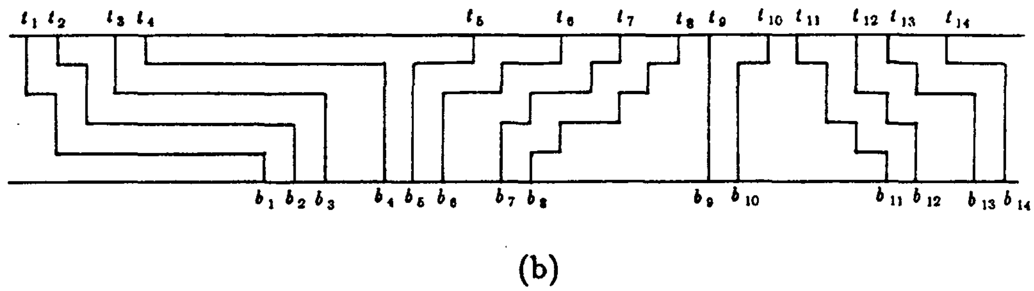
## 1.2 Preliminary Definitions

The class of general *river routing problems* involves routing between ordered sequences of terminals such that the final layout is planar. One such problem is the wiring of two ordered sets of terminals $\{b_1, b_2, \ldots, b_n\}$ and $\{t_1, t_2, \ldots, t_n\}$ across a channel between the parallel boundaries of two rectangles. The width of the channel is the vertical distance between the two lines forming the 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 unit distance. The *offset problem* is to find the distance to slide one of the rectangles so as to achieve the minimum possible separation between the rectangles. Figure 1.1.(a) shows an example of river routing problem and a wiring achieving the minimum separation. Figure 1.1.(b) shows the wiring obtained by sliding the bottom terminals to the right of 7 unit distance. In this case the separation is optimal.

A more general version of the river routing problem, *routing within rectilinear polygon*, which is known to have an efficient serial algorithm, is to per-




(a).




(b)

Figure 1.1: (a). Basic river routing problem, (b) Minimum separation with offset 7

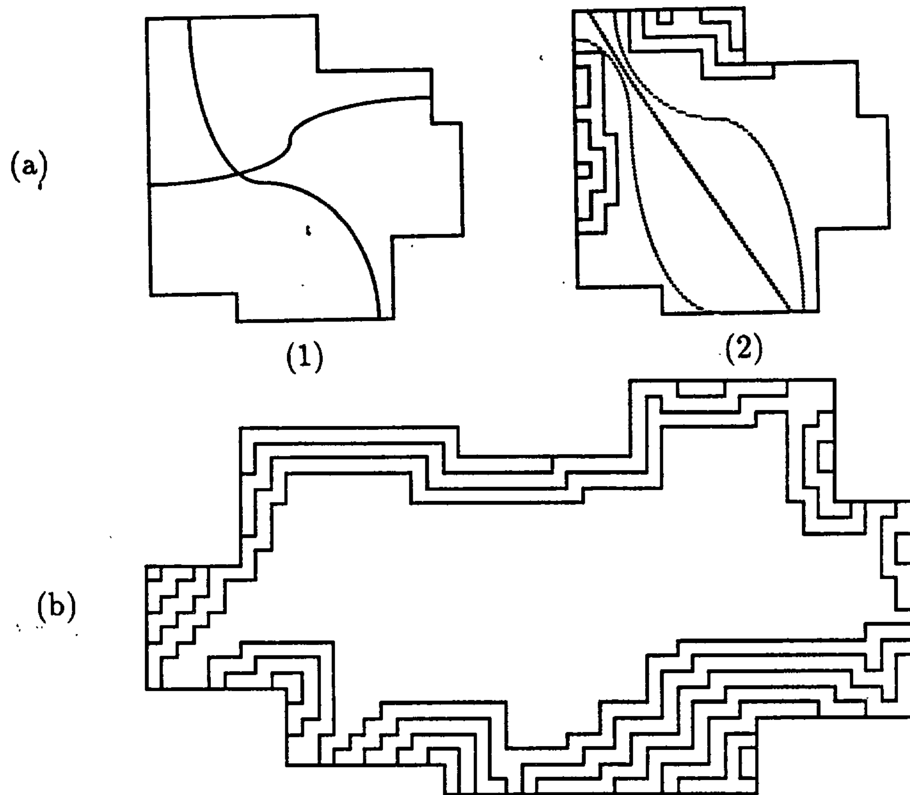



Figure 1.2: Routing within a rectilinear polygon : (a) Routability testing, (1) Planarity testing, (2) Area testing, (b) Detailed routing

form planar routing where the ports lie on the boundary of a simple rectilinear polygon. In this case, we are interested in whether the routing is possible or not(*routability testing*) and, if it is possible, we have to provide the *detailed routing*. Figure 1.2.(a) shows the two possible reasons that the problem is unroutable: (1) the connection is nonplanar and (2) the area is not large enough for the routing. Figure 1.2.(b) is an example of the detailed routing of this problem. Several interesting subproblems such as finding the contour of the union of a set of rectilinear polygons or determining whether a set of nets can be wired within a set "passages" are also tackled.

The problem of *wiring module pins to frame pads* is to connect a set of pins on a module to a set of pads lying on the boundary of a chip. This problem

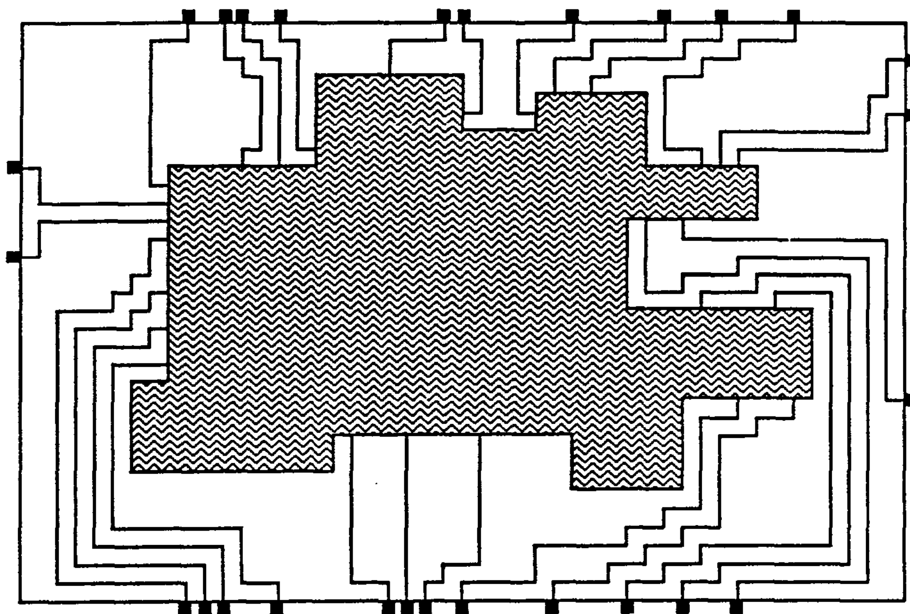


Figure 1.3: Routing between module pins and frame pads

has been addressed in the sequential context by Baker and Pinter ([BaP]). An instance is given by a triplet $< \mathcal{M}, \mathcal{F}, \mathcal{N} >$, where $\mathcal{M}$ is an arbitrary rectilinear polygon representing a module, $\mathcal{F}$ is a rectangle representing a frame (assuming a horizontal bottom edge), and $\mathcal{N}$ is a set of two-terminal nets such that one terminal is on $\mathcal{M}$ and the other is on $\mathcal{F}$. We assume that $\mathcal{F}$ contains $\mathcal{M}$ and that each boundary segment of $\mathcal{M}$ is parallel to a frame edge. Our goal is to find out whether a *one-layer* routing exists within the given area (*routability testing*) and if it does to determine such a wiring (*detailed routing*). We also address the problem of changing a given such wiring so that the resulting wiring is of minimum length (*minimizing wire length*). Figure 1.3 shows an example of such a routing problem.

The next problem we consider is the *channel routing problem* of two terminal nets in the knock-knee model. An instance of the channel routing problem (CRP) is a channel consisting of a rectangular grid and a set of nets whose terminals lie on the grid points of the (horizontal) parallel boundaries. A routing

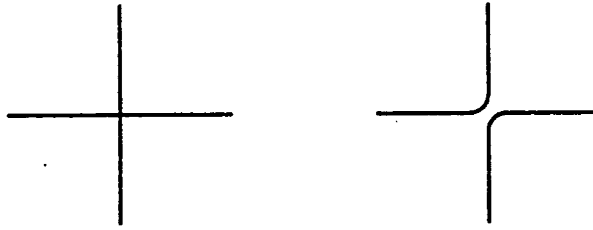


Figure 1.4: Types of shared grid points in the knock-knee model

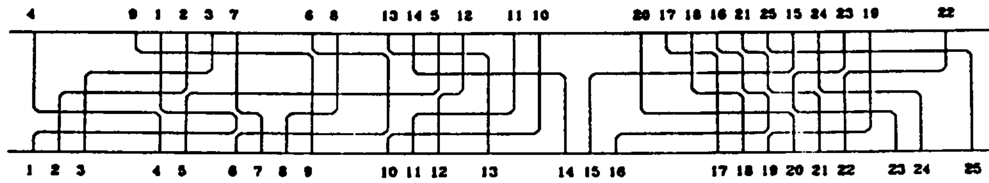



Figure 1.5: Channel routing in the knock-knee model

in the knock-knee model consists of a set of edge-disjoint paths (made up of gridline segments) connecting the terminals of each net. Hence a shared grid point could be one of two types: crossing and knock-knee.(as shown in Figure 1.4) Our goal is to determine the wiring of all the nets with the minimum number of tracks (*layout problem*). In addition, the resulting layout should be realizable in a minimum number of layers (*layer assignment problem*). Figure 1.5 is an example of channel routing problem in the knock-knee model.

For all of the above VLSI routing problems, we will restrict ourselves to the case where the wires are rectilinear, i.e., there is a grid structure such that each wire consists of a set of grid line segments ([Lei],[Ull]). Our methods generalize for all the other known variations ([SeD],[Tom]).

Next, we give some definitions of algorithm analysis. To develop and analyze algorithms, we need to establish a formal computing model. A widely used abstract computation model is the unit-cost RAM model ([AHU]) which has a standard set of operations, each taking unit time, and a random access memory of unlimited size with unit access time. The complexity of an algo-

rithm is determined by the amount of computing resources needed to solve the problem. Resources may be the overall execution time or the amount of the memory space required. The size of a problem is a measure of the quantity of input data needed to describe the problem. A complexity function is a function in terms of the size of the problem expressing the amount of resources that are needed by the algorithm in order to solve the problem of that size. The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. Analogous definitions can be made for the *space complexity.* When discussing the complexity functions, we usually use the following notations for asymptotic analysis:

- A function $f(n) = O(g(n))$ if and only if there exist positive constants $c$ and $n_0$ such that, for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.
- A function $f(n) = \Omega(g(n))$ if and only if there exist positive constants $c$ and $n_0$ such that, for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$.
- A function $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

For parallel algorithms, the *Parallel Random Access Machine* (PRAM) is a widely used computation model ([FoW],[Sch]), which has an unbounded number of processors sharing a common memory. There are many variants of this model. Our basic model of parallel processing is the CREW-PRAM (*Concurrent Read Exclusive Write*) model in which different processors may read data from the same memory location but concurrent writes are not allowed. All our parallel algorithms are based on this model and can be mapped into other more realistic models such as mesh, hypercube, cube connected cycle etc. ([Bar],[Prv],[Ree],[Sto], [Stu],[Wis],[Ull]). One important resource of parallel processing systems is the number of processors required to solve the problem by an algorithm.

We now introduce several complexity classes and relate them to our work. In general, two problems are in the same class if they are equally as difficult to solve. For serial computation, there are two main classes: $P$ and $NP$ ([HoU],[GaJ]). The class $P$ consists of all problems solvable on a deterministic Turing Machine in polynomial time. $NP$ is the class of all problems solvable by a non-deterministic Turing Machine in polynomial time. Therefore all problems in $P$ are also in $NP$. We define another class, $NP$-complete, to be the problems that are the "hardest" in $NP$, i.e., any problem in $NP$ can be reduced to any one of these problems. Informally, we can think of $NP$-complete problems to be those that are the most unlikely problems in $NP$ to posses efficient solutions. For parallel computation, we will say that a problem of size $n$ is easy if it can be done in poly-log time when using a polynomial number of processors ([Pip]). This class of problem is called $NC$. Clearly, for any problem in $NC$, it must be in $P$. Similarly, we can define the class $P$-complete, which are the most unlikely problems in $P$ to posses efficient parallel solutions. Any algorithm found to solve a $P$-complete problem in poly-log time would allow all problems in $P$ to be solvable in poly-log time. Theoretical research on parallel algorithms has focused on $NC$ theory. This motivates the development of parallel algorithms that are extremely fast, but possibly wasteful in their use of processors ([KR2],[ViS]). Such algorithms seem to be of limited interest for real applications. We are aiming for *efficient parallel algorithms* that run in $O(\frac{T(n)}{p} + \log^k p)$, where $p$ is the number of processors ($1 \leq p \leq n$), $T(n)$ is the running time of the best known sequential algorithm with input length $n$, and $k$ is a fixed positive constant. In this case, we can achieve a linear speedup. Most parallel computers have only a limited number of processors instead of unbounded number of processors as defined in the PRAM model.

## 1.3 Parallel Computation Models

To develop and analyze algorithms, we need to establish a formal computation model. The model should be abstract enough to make performance analysis tractable and realistic enough to reflect existing features of parallel computer architectures. The diversity of parallel computer architectures makes it impossible to define a unique model which is representative of all parallel computers. In this thesis, we will concentrate on the CREW-PRAM model. We will show that all our algorithms can be mapped into many other computation models including those discussed in this chapter ([CJ4]).

### 1.3.1 PRAM Model

As we stated before, the PRAM is a widely used parallel computation model ([FoW][Sch]), which consists of an unbounded number of processors sharing a common memory. It is a SIMD (*Single Instruction Multiple Data*) machine in which all the processors are synchronized by a common clock and execute the same instruction at each clock tick ([HwB]). (Processors may be enabled or disabled such that the common instruction for any given time unit is executed only on enabled processors.) The instructions may include reading from or writing to the shared memory and some basic arithmetic and logic operations. During each instruction cycle any number of processors can simultaneous access the shared memory. There are many variants of this model depending on how they handle simultaneous access to the same memory location. The weakest PRAM model is the EREW (*Exclusive Read Exclusive Write*) PRAM model, which does not allow concurrent access to the same memory location. CREW (*Concurrent Read Exclusive Write*) PRAM is a stronger PRAM model which allows concurrent reads to the same memory location while concurrent writes

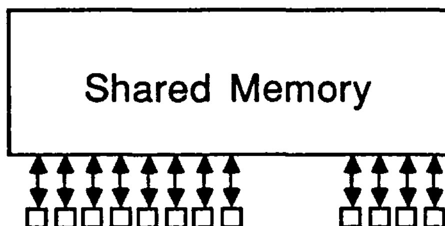



Figure 1.6: The structure of the PRAM model

are not allowed. Note that a processor can modify the content of a memory location only if it has exclusive access to it. This is the model considered in this thesis. CREW (*Concurrent Read Concurrent Write*) PRAM model is the strongest parallel computation model. There exist no constraints on concurrent access to the same memory location. There are various schemes for solving write conflicts ([Kue],[Gol],[DyR]).

Any algorithm that runs on EREW PRAM can be executed on CREW and CRCW PRAM's. Similarly an algorithm developed on CREW PRAM can run on CRCW PRAM. [Vi2] shows that an algorithm runs on CRCW PRAM in $O(T(n))$ can be executed in time $O(T(n) \cdot \log n)$ on CREW PRAM. Similarly a factor of $O(\log n)$ increment is sufficient for transferring an algorithm from CREW PRAM to EREW PRAM.

Figure 1.6 shows the structure of a PRAM model.

## 1.3.2 Mesh-Connected Model

In this model, the processors may be thought of as logically arranged as in a $k$-dimensional array $A(n_{k-1}, n_{k-2}, \ldots, n_0)$, where $n_i$ is the size of the $i$th dimension and $N = n_{k-1} \cdot n_{k-2} \cdot \ldots \cdot n_0$ is the total number of processors. The processor at location $A(i_{k-1}, i_{k-2}, \ldots, i_0)$ is connected to processors at location

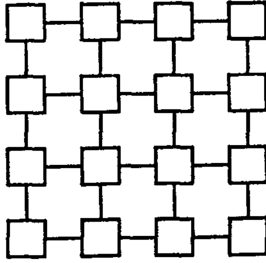


Figure 1.7: The structure of a two dimensional mesh with 16 processors

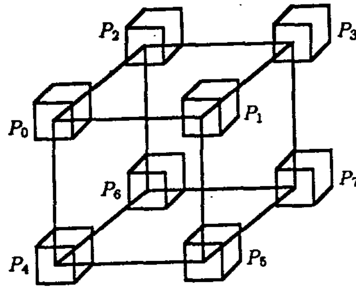



Figure 1.8: The structure of a hypercube with 8 processors

$A(i_{k-1}, \ldots, i_j \pm 1, \ldots, i_0)$, $0 \leq j \leq k-1$, provided they exist. Data may be transmitted from one processor to another only via this interconnection link ([NaS],[Stu],[Ree],[Bar]). Finger 1.7 shows an example of (2-dimensional) $4 \times 4$ mesh.

### 1.3.3 Hypercube Model

A hypercube of dimension $d$ is a set of $N = 2^d$ processors, connected in the form of a $d$-dimensional cube. That is, each node has a unique number $i$, for $0 \leq i < 2^d$, and two processors are connected if and only if the binary representations of their numbers differ in exactly one bit position. Figure 1.8 is an example of $2^3$ hypercube.

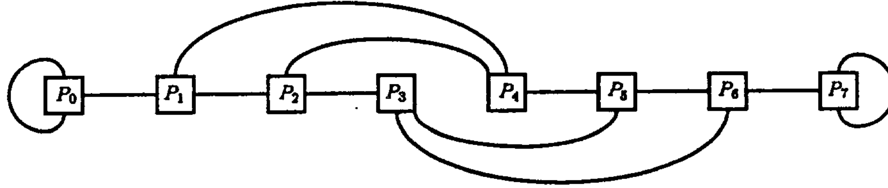



Figure 1.9: The structure of a perfect shuffle model with 8 processors

### 1.3.4 Perfect Shuffle Model

Assume that there are $N = 2^p$ processors and let $i_{p-1}i_{p-2}\ldots i_0$ be the binary representation of $i$, for $0 \leq i < N$. Let $i^{(b)}$ be the the number whose binary representation is $i_{p-1}\ldots i_{b+1}\bar{i_b}i_{b-1}\ldots i_0$, where $\bar{i_b}$ is the complement of $i_b$ and $0 \leq b \leq p-1$. Define $SHUFFLE(i)$ and $UNSHUFFLE(i)$ to be the integers with binary representation $i_{p-2}i_{p-3}\ldots i_0 i_{p-1}$ and $i_0 i_{p-1}\ldots i_1$ respectively. In the perfect shuffle model processor $i$ is connected to processors $i^{(0)}, SHUFFLE(i)$ and $UNSHUFFLE(i)$. Figure 1.9 shows an example of perfect shuffle structure with 8 processors.

### 1.3.5 Shuffle-Exchange Network

The shuffle-exchange (butterfly) network consists of $(k+1)\cdot 2^k$ processors, organized as $k+1$ ranks of $N = 2^k$ processors each. Let us denote the $i$th processor on the $r$th rank by $P_{ri}$, $0 \leq i \leq N-1, 0 \leq r \leq k$. Then processor $P_{ri}$ on rank $r > 0$ is connected to two processors on rank $r-1$, the two processors $P_{r-1,j}$ such that either $j = i$ or the binary representation of $j$ differs from that of $i$ only in the $r$th place from the left ([Wis][Ull]). Figure 1.10 is an example of shuffle-exchange network with $k = 3, N = 8$.

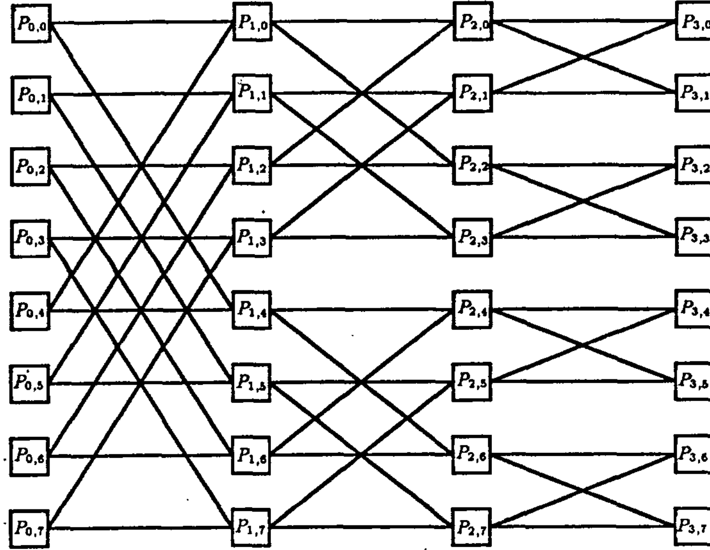


Figure 1.10: The structure of a shuffle-exchange network with $k = 3, N = 8$

### 1.3.6 Cube-Connected-Cycles

The cube-connected-cycles ([PrV]) consists of $N = 2^k$ processors, $k = r + 2^r$ for some positive integer $r$. $N$ processors are partitioned into $2^{k-r}$ groups, each group has $2^r$ processors. Each processor has a unique index $(i, j)$, $0 \leq i \leq 2^{k-r} - 1$ and $0 \leq j \leq 2^r - 1$, where $i$ is the group index and $j$ is the index within each group. Let $i = i_{k-r-1} \ldots i_0$ and $i^{(j)} = i_{k-r-1} \ldots i_{j+1} \bar{i}_j i_{j-1} \ldots i_0$. Then

If $0 < j < 2^r - 1$ then processor $(i, j)$ connect to processors $(i, j-1), (i, j+1)$ and $(i^{(j)}, j)$.

If $j = 0$ then processor $(i, j)$ connect to processors $(i, 2^r - 1), (i, j+1)$ and $(i^{(j)}, j)$.

If $j = 2^r - 1$ then processor $(i, j)$ connect to processors $(i, j-1), (i, 0)$ and $(i^{(j)}, j)$.

Figure 1.11 is an example of cube-connected-cycles with $r = 1, k = 3, N = 8$.

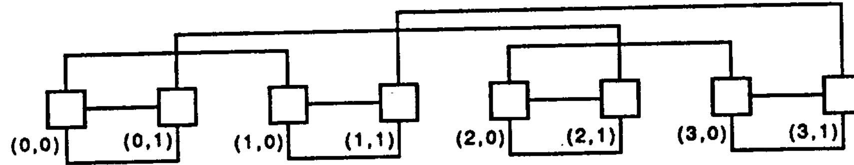



Figure 1.11: The structure of a cube-connected-cycles with $r = 1, k = 3, N = 8$

## 1.4 Basic Parallel Techniques

In this section, we will review some basic parallel operations, such as *path doubling, parallel prefix computation, list ranking, sorting, random access read* and *random access write.* These well known parallel techniques are used extensively in our parallel routing algorithms. They are important in the time complexity analysis of our algorithms.

Given $n$ elements $a_0, a_1, \ldots a_{n-1}$ and an associative operation $*$, the prefix computation is to compute all $n$ initial products $a_0 * a_1 * \ldots * a_i, i = 0, 1, \ldots n-1$. When solved in parallel, this problem is known as *parallel prefix.* The problem was first proposed by Wyllie ([Wyl]) who introduced an $O(\log n)$ time algorithm using $O(n)$ processors. This result has been gradually improved in a series of papers ([Vi1],[WaH],[KR1],[MiR],[CV1],[CV2]).

If all the data are stored in consecutive memory locations and there are $n$ processors, then we can solve the problem by the following well known algorithm.

*Algorithm Parallel Prefix I*

*Input*: $n$ elements $a_0, a_1, \ldots a_{n-1}$ stored in consecutive memory locations and an associative operation $*$.

*Output*: $n$ initial products $a_0 * a_1 * \ldots * a_i, i = 0, 1, \ldots n-1$.

1. The ith processor $PE_i$ reads $a_i$ from memory, $0 \leq i \leq n-1$, and set $b_i$ as the value of $a_i$. Set $j = 1$.

2. For each processor $PE_i$, if $i - j \geq 0$, then set $b_i = b_i * b_{i-j}$.

3. Set $j = 2j$.

4. Repeat Steps 2. and 3. $\lceil \log n \rceil$ times. The $b_i$'s contain the final results,$0 \leq i \leq n-1$.

Each step of above algorithm can be done in constant time (For example, in Step 2 $PE_i$ can read out $b_{i-j}$ in constant time on the CREW-PRAM model), so the time complexity of this algorithm is $O(\log n)$ with $O(n)$ processors. If we have only $p$ processors, $p < n$, then we can partition these $n$ elements into $p$ blocks, each containing about $\frac{n}{p}$ contiguous elements. Assigning one processor to each group and using the standard sequential algorithms, which scans each element in sequence and generates $i$th product after scanning $i$th element, we can get the prefix within each group in $O(\frac{n}{p})$ time. Then use algorithm Parallel Prefix I to find the prefix between blocks and modify each group independently using the sequential algorithm. We can get all the $n$ prefix results in $O(\frac{n}{p}+\log p)$ with $p$ processors, $p \leq n$, which is provably optimal. If $p > n$ we can do nothing better than $O(\log n)$. Hence, we have the following property:

**Lemma 1.1**: If all the data are stored in consecutive memory locations then parallel prefix computation can be done in $O(\frac{n}{p} + \log n)$ time with $p$ processors on the CREW PRAM model, for any positive integer $p$.

Now we consider the case where these $n$ elements are stored in a linked list. A linked list specifies the order of the elements so that element $i+1$ can be found only after element $i$ is found, the first element in the list can be found in constant time. To solve the prefix computation with this input data structure

and $O(n)$ processors, we can use algorithm Parallel Prefix I by introducing the *path doubling* technique to get the following algorithm.

*Algorithm Parallel Prefix II*

*Input*: $n$ elements $a_0, a_1, \ldots a_{n-1}$ stored in a linked list, each node $i$ has two pointers $succ(i)$ and $pred(i)$ to point to successor and predecessor in the linked list, $(pred(0) = succ(n-1) = nil)$, and an associate operation $*$.

*Output*: $n$ initial products $a_0 * a_1 * \ldots * a_i, i = 0, 1, \ldots n-1$.

1. Assign one processor to each element $a_i$, $0 \leq i \leq n-1$, and set the value of $b_i$ the same as $a_i$.
2. For each $i$, if $pred(i) \neq nil$ set $b_i = b_i * b_{pred(i)}$ and $pred(i) = pred(pred(i))$.
3. Repeat Step 2. $\lceil \log n \rceil$ times, then $b_i$'s, $0 \leq i \leq n-1$, are the final results.

In Step 2. we use the *path doubling* technique $(pred(i) = pred(pred(i)))$. At each iteration the distance the pointers span is doubled by setting the pointer to the value of the pointer belonging to the element pointed to on the previous iteration. At each step a processor doubles the number of elements for which it has calculated the prefix result. Hence "growth" is achieved by doubling at each step. Figure 1.12 is an example of parallel prefix computation, in which, given a linked list of 8 numbers, we want to find all the $i$th initial summation, $0 \leq i \leq 7$.

If we have only $p$ processors, $p < n$, [KR1] presents an algorithm to solve this problem. The basic operation is to replace pairs of elements by their products, until a single value is obtained, then reverse the computation by poping out the recursion to get the prefix results. Thus, an efficient parallel prefix computation is obtained by designing a scheduling algorithm that continually schedules $\Theta(p)$ pairing for concurrent evaluation. The overall structure consists of $\Theta(\log n)$ phases. At each phase a constant fraction of the elements are paired

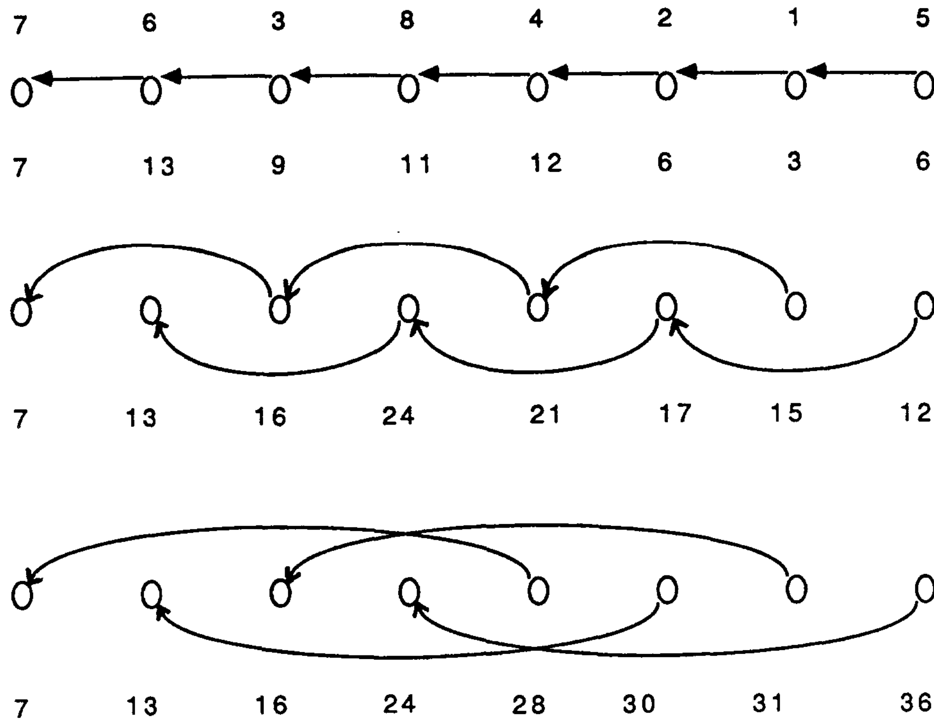



Figure 1.12: Parallel prefix computation by using path doubling technique

with their neighbors, producing for each pair a new element with the product of their values. This reduces the problem size by a constant fraction. Once there are less than $2p$ elements the $O(\log p)$ parallel algorithm is used to compute the result. The time complexity of this algorithm is $O(\frac{n}{p} + \log n)$ with $p$ processors, where $p \leq n^{1-\epsilon}$, and $\epsilon$ is any positive constant. For the case of $p > n^{1-\epsilon}$ we can use following result. [CV2] presents a list ranking algorithm to compute the distance from each element to the first element in the linked list. Their algorithm has three stages ([CV2]) :

1. Reduce the list of $n$ elements to a new list of at most $\frac{n}{\log n} + 1$ elements in $O(\log n)$ steps. The time complexity of this step is $O(\log n)$ with $O(\frac{n}{\log n})$ processors.

2. Perform the parallel prefix computation algorithm to get the ranking of the reduced list (with $O(\frac{n}{\log n})$ elements). With $O(\frac{n}{\log n})$ processors, the time

complexity is $O(\log n)$.

3. Calculate the ranking of the original list by processing the elements that were removed from the list. This can also be done in $O(\log n)$ time with $O(\frac{n}{\log n})$ processors. (The details of this step depend on the data structure used to represent the removed elements.)

Based on above results ([KR1][CV2]), which give us an $O(\frac{n}{p} + \log n)$ time list ranking algorithm with any $p$ processors, we can do parallel prefix computation with the same time complexity as that of list ranking. (First transform the linked list into consecutive memory location according to the distance of each element to the first element in the list, then we can apply Lemma 1.1.)

**Theorem 1.1**: Given $n$ elements $a_0, a_1, \ldots a_{n-1}$ and an associative operation $*$, we can compute all the $n$ initial products $a_0 * a_1 * \ldots * a_i, i = 0, 1, \ldots n-1$ in $O(\frac{n}{p} + \log n)$ time with $p$ processors in the CREW PRAM model, for any positive integer $p$.

Sorting is the process of rearranging a sequence of values in ascending or descending order. The first results on parallel sorting were related to sorting networks ([Bat]). Fast parallel sorting algorithms were presented in ([Hir],[Pre]) for theoretical models of parallel processors with shared memory. A set of results in shared memory computation has led a number of parallel sorting schemes that exhibit a $O(\log n)$ time complexity ([BDH]).

Both of Batcher's algorithms, the *odd-even sort* and the *bitonic sort*, are based on the principle of iterated merging. A specific iterative rule is applied to an initial sequence of $n = 2^k$ numbers in order to create sorted runs of length $2, 4, 8, \ldots 2^k$ during successive stages of the algorithm ([Bat]). With these methods, we can sort $n$ numbers with $O(n \log^2 n)$ comparison, i.e., in the PRAM model, this can be done in time $O(\log^2 n)$ with $O(n)$ processors.

After $O(\log^2 n)$ time complexity sorting algorithms were achieved based on above methods, researchers attempted to improve this to $O(\log n)$. The new sorting algorithms generally use *enumeration* to compute the rank of each element. Sorting is performed by computing in parallel the rank of each element and routing the elements to the location specified by their rank. Muller and Preparata first proposed a modified sorting network based on a special type of comparators ([MuP]). To sort a sequence of $n$ elements, each element is simultaneously compared to all the other in one unit of time by using a total of $n(n-1)$ comparators. The outputs of comparators are then fed into a parallel counter. After $\log n$ steps, we can determine the rank of each element. Based on this idea, we can sort $n$ elements in $O(\log n)$ time with $O(n^2)$ processors. Their algorithm was the first to use an enumeration scheme for parallel sorting.

The idea of sorting by enumeration was exploited to develop other very fast parallel sorting algorithms, which improve Muller and Preparata's result by reducing the number of processors ([Hir],[Pre]). Hirschberg's sorting algorithm sorts $n$ numbers with $O(n)$ processors in time $O(\log n)$, provided that the numbers to be sorted are in some fixed range, say between 0 and $m-1$. His algorithm creates $m$ buckets and assign one number to each processor. The processor that gets the ith number is labled $P_i$ and it is responsible for placing the value $i$ in the appropriate bucket. The memory contention problems can be solved by substantially increasing the memory requirements. Then the $m$ arrays must be merged. The processors perform this merge operation by searching in a binary tree search method ([Hir]). Other related $O(\log n)$ sorting algorithms can be found in [Pre],[AKS],[Col],[Leg].

**Theorem 1.2**: With $O(n)$ processors in the CREW PRAM model, we can sort $n$ number in $O(\log n)$ time.

Using the result in the parallel prefix computation (Theorem 1.1), we have

following property :

**Theorem 1.3**: If all the $n$ elements lie in the range $[1, N]$ and, for any number in $[1, N]$, only $O(1)$ elements have that value, where $N = O(n)$, then sorting can be done in $O(\frac{n}{p} + \log n)$ time with $p$ processors, for any positive integer $p$.

Even though the above discussion has been based on the CREW PRAM model, all these parallel techniques can be applied to other parallel computation models discussed previously. For example, with $O(n)$ processors, parallel prefix can be done in $O(\sqrt{n})$ time on a two-dimensional array, $O(\log n)$ time on Hypercube, Perfect Shuffle, Shuffle-Exchange Network and Cube-Connected Cycles. Similarly the sorting can be done in $O(\sqrt{n})$ time on a two-dimensional array, $O(\log^2 n)$ time on Hypercube, Perfect Shuffle, Shuffle-Exchange Network and Cube-Connected Cycles.

In the CREW PRAM model each processor can access any memory location in constant time. This is not possible for the other parallel models defined in the previous section. This data broadcasting problem can be posed in two different ways : (1) *Random Access Read (RAR)*: Each processor $i$ has an index $j$, $0 \leq i, j \leq n-1$, which means processor $i$ is to receive a data from processor $j$. (2)*Random Access Write (RAW)*: Similar to RAR except that processor $i$ wants to write a data into processor $j$'s register. [NaS] presents an algorithm to solve these problems. Both RAR and RAW can be transferred to a combination of integer sorting and other related operations ([NaS]). Hence RAR and RAW have the same time complexity as the integer sorting. i.e. RAR and RAW can be done in $O(\sqrt{n})$ time on a two-dimensional array, $O(\log^2 n)$ time on Hypercube, Perfect Shuffle, Shuffle-Exchange Network and Cube-Connected Cycles.

## 1.5 Summary of Results

In this thesis, we investigate several VLSI routing problems. The known strategies to handle these problems seem to be inherently sequential. We develop new techniques that lead to efficient parallel algorithms for *river routing* problems ([CJ1]), the problems of *wiring module pins to frame pads* ([CJ2]) and *channel routing* in the knock-knee model ([CJ3]).

Our main results can be briefly stated as follows:

**(1)** $O(\frac{n}{p} + \log n)$ algorithms, with $1 \leq p \leq n$ processors on the CREW PRAM model, to find the minimum separation required to river route $n$ nets across a channel between two rectangles, and to determine all the essential information for the final planar layout. We partition the nets into blocks such that the wiring problem is reduced to wiring each block simultaneously and independently. The wiring of a net can be specified by the coordinates of its bend points. Not all of the bend points are needed to determine the overall wiring. We will introduce the *characteristic bend points* which will uniquely define the overall wiring and develop efficient parallel algorithms to determine the characteristic bend points.

**(2)** $O(\frac{n \log n}{p} + \log^2 n)$ algorithms, with $1 \leq p \leq n$ processors on the CREW PRAM model, to determine the optimal offset of the river routing problem. We partition the nets into blocks and then determine the upper bound and lower bound of the possible separations after sliding one side of terminals. A binary search over the above bounds will yield an optimal offset in $O(\log n)$ iterations. Each iteration requires $O(\log n)$ time. This give $O(log^2 n)$ algorithms to determine the optimal placement to slide one of the rectangles so as to achieve the minimum channel width.

**(3)** $O(\frac{n}{p} + \log n)$ algorithms, with $1 \leq p \leq n$ processors on the CREW PRAM

model, to detect the routability of the routing problem within a rectangle. This includes the detection of *planarity* connection between nets and detecting whether the *area* is large enough for the detailed routing. Planarity can be detected easily by the combination of the techniques such as path doubling, prefix computation and sorting. Our approach to area testing is to partition nets into blocks, then determine *the wiring capacity* and the *wiring density* between blocks. Based on this information, we can detect the routability.

**(4)** $O(\frac{n}{p} + \log n)$ algorithms, with $1 \leq p \leq n$ processors on the CREW PRAM model, to determine the detailed routing of the routing problem within a rectilinear polygon. Our strategy for the routing problem will consist of identifying a set of net groups and the *representative net* of each group and then performing the wiring of each such net with the nets "covered" by it separately.

**(5)** $O(\frac{n}{p} + \log n)$ algorithms, with $1 \leq p \leq n$ processors on the CREW PRAM model, to determine the layout of wiring between module pins to frame pads. For any routing problem $< \mathcal{M}, \mathcal{F}, \mathcal{N} >$, we partition the nets into groups such the nets in the same group share a 45 degree diagonal. For each net, we call the intersection point of the wiring with the diagonal the *intermediate terminal.* The wiring of each net consists of two parts. The first one starts from the module terminal and routes as close to the module boundaries as possible until the intermediate terminal is reached. The other part is between the intermediate terminal and the frame terminal and stays as close to the frame edges as possible. The first part can be determined by the techniques of determining the union of rectilinear polygons. For the second part, we move the intermediate points vertically to a horizontal line $L$ such that the separation distance is enough to solve the corresponding river routing problem. With the above methods, we can determine the wiring of each net.

**(6)** $O(\frac{n \log n}{p} + \log^2 n)$ algorithms, with $1 \leq p \leq n$ processors on the CREW

PRAM model, to do the routability testing of wiring module pins to frame pads. For the routability testing, we can find out the routing of the outermost nets in each group and then detect the intersection between these nets, module and frame boundary by the methods of [MiS]. The problem is routable if there exist no intersections.

(7) $O(\frac{n \log n}{p} + \log^2 n)$ algorithms, with $1 \leq p \leq n$ processors on the CREW PRAM model, to minimize the wire length of the layout given by (5). The algorithms consist of identifying horizontal and vertical wells (U-wires defined in [BaP]) and reducing them whenever possible. This process is repeated $O(\log n)$ times and there is no reducible U-wires left, then the resulting wiring is of minimum length.

(8) $O(\frac{n}{p} + \log n)$ algorithms, with $1 \leq p \leq n$ processors on the CREW PRAM model, to generate the layout of channel routing problem in the knock-knee model. Our approach consists of two main steps: (a) Partition the nets into $d$ chains satisfying certain properties to be outlined in Chapter 6, where $d$ is the density of the corresponding channel routing problem. In particular, the nets in each chain define a set of nonoverlaping intervals. (b) Assign a track number to each chain. Then wire all the nets simultaneously.

(9) $O(\frac{n}{p} + \log n)$ algorithms, with $1 \leq p \leq n$ processors on the CREW PRAM model, to do the layout modification and three layer assignment for the wiring of (8). The layout generated by (8) can be realized with four layers ([BrB]). We have developed an algorithm to modify the wiring such that it can be laid out in three layers. [PrL] provides necessary and sufficient conditions for the realization of a wiring in three layers. The problem is essentially reduced to finding a legal partition of the core of the diagonal diagram (see chapter 6). Our strategy can be summarized by two algorithms. The first algorithm modifies the wiring to eliminate the possible configuration that can create problems, and the second

algorithm shows how to generate a legal partition.

The rest of this dissertation is organized as follows. Chapter 2 gives an overview of general VLSI routing and the sequential strategies of the routing problems of interest. Chapter 3 deals with the *separation* and *offset problems* of river routing between two rectangles. The *routing problem within a rectilinear polygon* is tackled in Chapter 4. Chapter 5 considers the problem of *wiring module pins to the frame pads.* In Chapter 6 we develop a novel routing strategy to solve *channel routing problem in the knock-knee model.* The implementation and conclusions are discussed in Chapter 7.

CHAPTER
TWO

# VLSI ROUTING

## 2.1 Introduction

With the increase in the complexity of chip fabricated today, the design effort and turnaround time increase at a high rate especially in the layout design phase. For example, in the design of Z8000 microprocessor, about 6,600 man-hours, corresponding to 50% of the whole design effort, was required in its layout design phase ([Ric],[UKS]). Circuit layout is a crucial part of VLSI design. It deals with the physical design of chip to fulfill the device, circuit and interconnection specifications, and at the same time, to satisfy the design rules for a given technology. The layout problem is usually divided into subproblems of chip planning and wirability analysis, partitioning, assignment, placement, global routing and detailed routing ([Oht]).

Partitioning is a process of creating the physical hierarchy of the hardware elements used to realize the design. It is essentially the subdivision of logic complexes into smaller subcomplexes. The assignment problem is to find one-to-one correspondence between logic circuit elements and placement modules. The placement problem is to find appropriate locations for individual modules on the chip or board ([GoM]).

Global Routing is the preliminary step of the complete routing process. The main objective for global routing is to develop a good routing plan based on the given placement in such a way that detailed routing can be completed efficiently. It calls for a routing plan in which each net is assigned to particular regions on the chip reserved for routing. The aim is not only to make 100% assignments of nets to regions for routing, but also to minimize, for example, the total wire length leading to a minimal chip area. This problem can be formulated from a mathematical point of view, for example, based on Steiner tree and a global routing graph ([KuM], [Hak]).

For the detailed routing, a set of points on unconnected cells that must be connected is given. The space between these cells can be as simple as a rectangle or as complex as a maze. Many different techniques exist for routing, depending on the number of wires to run, the complexity of the routing space and the number of layers available for crossing ([Rub],[Oht]).

Channel Routing is one of the most important problems of VLSI routing. It is a special case of the routing problems when interconnections have to be performed within a rectangular strip with no obstacle inside and with terminals on the opposite sides of the rectangle. The final result depends on what routing model is used. In the standard two-layer model, net terminals are located on vertical grid lines, two wiring layers are available for interconnections — one layer is used exclusively for vertical segments, another for horizontal and a via is introduced for each layer change. Many well known results are based on this model, for example, the Dogleg Router ([Del]), the Greedy Router ([RiF]), the Hierarchical Router ([BP1],[BP2],[BHP]), et al. ([BrR],[BBL]). Most known problems under this model are $NP$−complete ([Del][LaP][RBM][Szy]). The knock-knee model is a multilayered routing model where wires are free to be routed in both directions (vertical and horizontal) on each layer, but overlap is forbidden and corner overlap is allowed (knock-knee). i.e. The wires form a set

of edge disjoint paths. Two wires may cross each other or bend at the same grid point. Efficient serial solutions exist for several problems based on this model ([PrL],[SaP],[MPS],[Sar], [MeP]). Other important variations of channel routing models are discussed in ([SaS],[SSR],[MaK],[RBM],[Ham]). Switchbox Routing can be viewed as a more general version of channel routing. Switchbox is sometimes referred to as "Four Sided Channel". It is a rectangular routing area with no obstacles inside and with terminals on all four sides of the rectangle ([HaO],[Hs1],[De2]). River Routing is an instance of VLSI routing within a channel or switchbox when there is only one layer available for interconnections. Optimal sequential solutions exist for several such problems ([Tom],[DKS],[SeD],[LeP],[Hs2],[Mir]).

Now we will describe the known sequential algorithms of the VLSI routing problems of interest. All these algorithms seem to be inherently sequential and new techniques have to be developed to obtain efficient parallel algorithms for these routing problems.

## 2.2 River Routing Between Rectangles

The class of general river routing problems involves routing between ordered sequences of terminals such that the final layout is planar. One such problem is the wiring of two ordered sets of terminals $\{b_1, b_2, \ldots, b_n\}$ and $\{t_1, t_2, \ldots, t_n\}$ across a channel between the parallel boundaries of two rectangles. The width of the channel is the vertical distance between the two lines forming the 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 unit distance. The *offset problem* is to find the distance to slide one of the rectangles so as to achieve the minimum possible separation between the rectangles.

In the following discussion, $b_i$ and $t_i$ will be also used to denote the hori-

zontal coordinates of the terminals relative to an arbitrary origin. Suppose we slide the rectangle containing $b_i$'s horizontally and call the amount of horizontal displacement $k$ as the *offset*. That is, with offset $k$, terminal $b_i$ move to horizontal position $b_i + k$, for $1 \leq i \leq n$. We say a pair $(d, k)$ is feasible if there is a feasible wiring with separation $d$ and offset $k$. Then the separation problem becomes : Given a fixed offset $k$, find the minimum separation $d$ such that $(d, k)$ is feasible. The offset problem is to find a feasible pair $(d, k)$ that minimizes $d$. An $O(n)$ time algorithm for the separation problem and an $O(n \log n)$ time algorithm for the offset problem are given in [DKS] and [SeD]. [Mir] improves their results to get an $O(n)$ algorithm for the offset problem. The material of this section is essentially given in [DKS],[Tom],[LeP] and [Mir].

After careful analysis ([Mir]), a necessary and sufficient condition for the feasibility of $(d, k)$ is given : $b_{i-d} + d \leq t_i - k \leq b_{i+d} - d$. Let $x_i = b_i - i$ and $y_i = t_i - i$. We see that $x_1 \leq x_2 \leq \ldots \leq x_n$ and $y_1 \leq y_2 \leq \ldots \leq y_n$. Let $x = (x_1, x_2, \ldots x_n)$ and $y = (y_1, y_2, \ldots y_n)$. We conclude that the pair $(d, k)$ is feasible if and only if $x_i - y_{i+d} \leq k \leq x_{i+d} - y_i$, for $1 \leq i \leq n - d$. Let

$$u(d, x, y) = min\{x_{i+d} - y_i | 1 \leq i \leq n - d\} \quad (1)$$

$$l(d, x, y) = max\{x_i - y_{i+d} | 1 \leq i \leq n - d\} \quad (2)$$

**Theorem 2.1**: ([Mir]) The pair $(d, k)$ is feasible iff $l(d, x, y) \leq k \leq u(d, x, y)$.

Theorem 2.1 implies a simple $O(n)$ time algorithm for the separation problem. Based on Theorem 2.1, the offset problem becomes : Find the minimum separation $d^*$ such that $l(d^*, x, y) \leq u(d^*, x, y)$. Once $d^*$ is known, then any value $k^*$, such that $l(d^*, x, y) \leq k^* \leq u(d^*, x, y)$ is a possible offset. Now we give an intuitive description of the $O(n)$ time algorithm of the offset problem developed by [Mir]. Let $x'$ be the $\lceil \frac{n}{2} \rceil$ vector, which is obtained from $x$ by removing

the odd indexed entries of $x$. That is $x'_i = x_{2i}$, for $1 \leq i \leq \lceil \frac{n}{2} \rceil$. $y'$ can be defined similarly. Let $d'$ be the minimum separation such that $l(d', x', y') \leq u(d', x', y')$.

**Theorem 2.2**: ([Mir]) Let $d^* = min\{d \geq 0 | l(d, x, y) \leq u(d, x, y)\}$ and $d' = man\{d \geq 0 | l(d, x', y') \leq u(d, x', y')\}$, then $|d^* - 2d'| \leq 1$.

As a result of Theorem 2.2, we can use this relation recursively to compute the separation and offset in $O(n)$ time ([Mir]).

Figure 1.1 shows an example of river routing problem between two rectangles.

## 2.3 Routing Within a Rectilinear Polygon

The routing problem of nets within a simple rectilinear polygon introduced in([Pin]) is a generalization of the standard river routing problem. In this case we are supposed to connect a set of terminals $a_1, a_2, \ldots, a_n$ on the boundary of a simple rectilinear polygon to another set of terminals $b_1, b_2, \ldots, b_n$ on the boundary of the same polygon such that all the wires lie within the polygon and no two wires intersect. *Routability testing* is to determine whether or not a one layer routing is possible and *detailed routing* is to specify the actual wiring of the $n$ nets, if they are routable.

To detect the routability, first we have to detect the planarity of the possible wiring, i.e. whether the terminals as they appear along the boundary of the polygon match. A sequential method in [Pin] proceeds as follows: Initialize the stack $S$ to be empty. Cut the boundary of the polygon at any point and straighten it out. Scan the terminals through this straight line. For each terminal, compare the net number to the number at the top of $S$. If they are equal, pop $S$, else push the number at the top of $S$. If at the end of the scan $S$

is empty, there exists no intersection. Otherwise, they are not realizable in one layer.

If the connection is planar, then we have to detect whether the area inside the polygon is large enough for the wiring. First, Pinter generates the boundary of each *group* which consists of nets with both terminals on the same side of the polygon. Then, for each convex corner of the above *group* boundaries, draw a horizontal, a vertical and a 45 degree line segments (rays) emitting from each convex corner (pointing outward). Based on the number of nets which are not in any of the *groups* and the intersection between those rays and the *group* boundaries, we can determine whether the area is large enough for the wiring ([Pin]).

For the detailed routing, Pinter uses the template given in the algorithm for planarity checking. Whenever a net is being popped off the stack, simply route it according to the following *greedy routing rule*: Each net is routed when all nets between its terminals along the boundary (according to the linear order) have already been routed. Each net stays as close as possible to the boundary by initially routing nets along the original boundary (for "intermost" nets), and then staying as close as possible to the contour formed by previous nets toward the boundary. Since the number of bendpoints can be as large as $\Omega(n^2)$, the worst case time complexity of this routing algorithm is $O(n^2)$, while the routability testing can be done in $O(n)$ ([Pin]).

Figure 1.2 is an example of routing problem within a rectilinear polygon.

## 2.4 Wiring Module Pins to Frame Pads

The problem of wiring module pins to frame pads is to connect a set of pins on a circuit module (a rectilinear polygon) to a set of pads lying on the boundary of a chip. This problem has been addressed in the sequential

context by Baker and Pinter ([BaP]). Their main idea is the following : Divide the routing problem into two subproblems that can be solved independently by opposite greedy strategies : one routing wires as close to the module as possible, and the other routing wires as close to the edges of the frame as possible. These are referred to as *Greedy-In* (GI) and *Greedy-Out* (GO) strategies respectively. Their algorithm is called the GIGO algorithm.

To give an intuitive description, we assume the circuit module is a rectangle. First, use GI strategy to route each net outward one unit from the module terminal, then turn in the required direction and route each net until it is blocked by another net, then take each net outward one unit again, and continue this procedure until each net reaches the corresponding 45 degree diagonal drawn through the corner of the (rectangular) circuit module. We call this intersection point an *intermediate terminal* of the corresponding net. Then, use GO strategy to route each net from pad terminal to intermediate terminal. For each net, starting at the pad terminal, keep the wire as close to the frame edge as possible while moving in the correct direction and keeping it a unit distance from the frame edge and the previous net, until it can be directly connected to the intermediate terminal by a horizontal or vertical line. (see Figure 2.1)

Figure 1.3 shows an example of a routing problem between module pins and frame pads.

## 2.5 Channel Routing in the Knock-Knee Model

An instance of the channel routing problem (CRP) is a channel consisting of a rectangular grid and a set of nets whose terminals lie on the grid points of the (horizontal) parallel boundaries. A routing in the knock-knee model consists of a set of edge-disjoint paths (made up of gridline segments) connecting the terminals of each net. Hence a shared grid point could be one of two types:

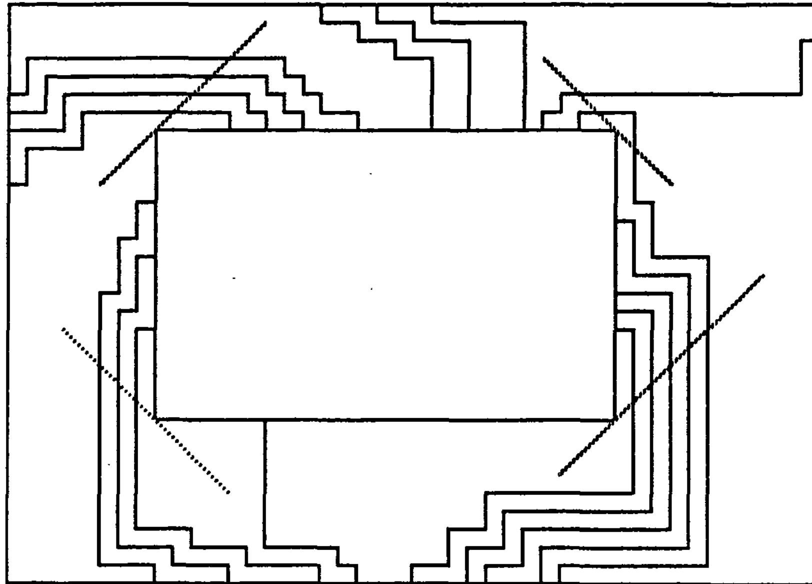


Figure 2.1: GI and GO routing strategies

*crossing* and *knock-knee.* An optimal sequential algorithm to generate a layout of $n$ nets is presented in [PrL]. It has the following properties : (1) it has the minimum channel width $d$ (density), (2) it can be realized with three layers, (3) it has at most $3n$ vias and (4) any two wires share at most four grid points. The time complexity of their algorithm is $O(n \log n)$ and can be modified to run in $O(n)$ time. Their algorithm constructs the wire layout track by track (from bottom to top), by laying each track from left to right. During the process of laying out a track each of the wires routed so far is extended, by adding a vertical edge and, possibly, a chain of horizontal edges. As a consequence, processing of a track modifies the CRP, by giving rise to a (residual) CRP to be solved in the remaining (upper) potion of the channel. Obviously, after processing a track, the density of the residual CRP has to be one less than that of the current CRP. Their overall strategy can be viewed as a nontrivial extension of the *line packing* (or *left edge*) algorithm, where a mechanism is provided to solve conflicts arising in columns. Each wire routed by the algorithm has one of the shape shown in

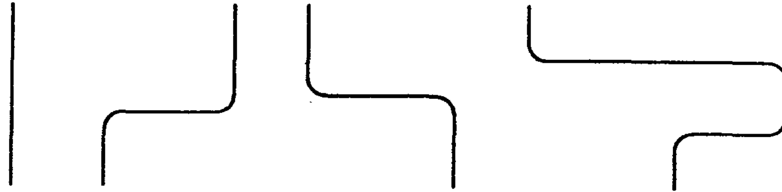


Figure 2.2: Possible shapes of wires routed by the sequential channel routing algorithm

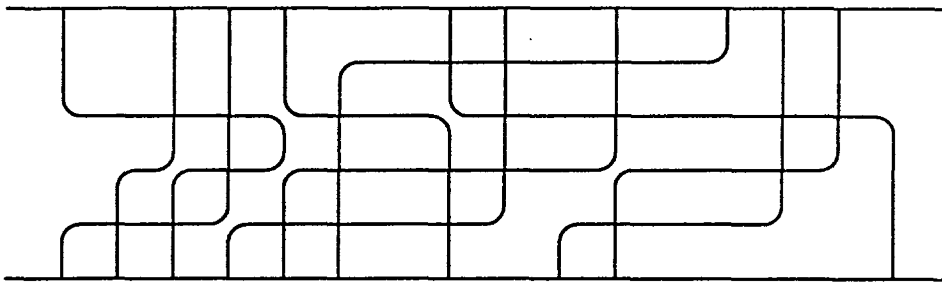


Figure 2.3: The layout of a channel routing problem generated by the sequential algorithm

Figure 2.2. Figure 2.3 is an example of channel routing problem and the wire layout generated by their algorithm.

Another version which proceeds one column at a time from left to right is given in [MPS]. The wire layout produced by their algorithm can either be directly realized with three layers or it can be modified to achieve this objective. To do the layer assignment they convert the layout into a *diagonal diagram* by replacing each knock-knee with a *diagonal* as shown in Figure 2.4.

**Theorem 2.3**: ([PrL]) A wire layout can be realized with three layers if there exists a *legal partition contour* $P$ for the corresponding diagonal diagram with following properties:

1. Every internal vertex of the partition contour is incident with an even

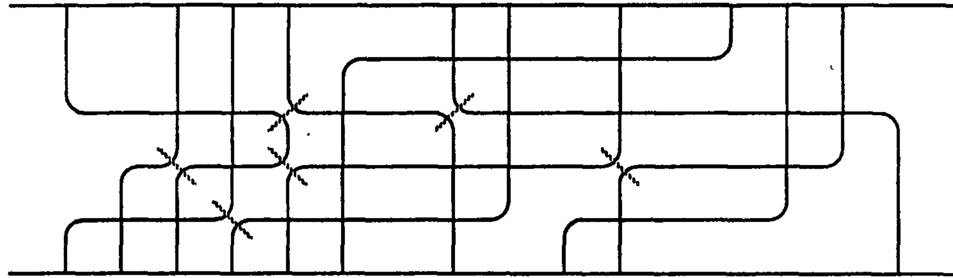


Figure 2.4: Mapping between knock-knee and diagonal diagram

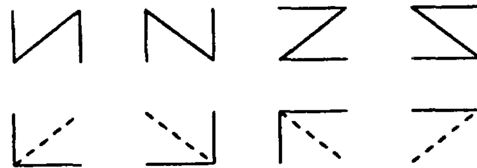


Figure 2.5: Forbidden patterns

number of edges in $P$. i.e. $P$ defines a two-colorable map.

2. The set of diagonal edges in $P$ coincides with the set of diagonals in the corresponding diagonal diagram of the layout.

3. $P$ does not contain any of the following patterns (see Figure 2.5):

   (a) The one where two nondiagonal edges incident on the two extremes of a diagonal edge form acute angles with it.

   (b) The one where a horizontal and vertical edge meet at a grid point and there is no diagonal internal bisecting the resulting 90 degree angle.

**Proof**: See [PrL].

After converting the layout to the corresponding diagonal diagram, every

column contains one of the following: (1) no diagonal, (2) a single diagonal \, (3) a single diagonal /, or (4) two diagonals, of which the lower is \ and the upper is /. We augment the diagonal diagram by adding / on a dummy track above all the tracks for every column of type (2), and placing \ on a dummy track below all the tracks for every column of type (3). We call these additional diagonals as *dummy* diagonals. After this modification, all nonempty columns are of type (4).

In addition to the diagonals, all the line segments in the partition contour will be constructed by adding vertical edges only. The partition algorithm will advance column-by-column from left to right and add appropriate vertical segments in such a way that certain properties are satisfied ([PrL]). To avoid a forbidden contour pattern, it is necessary to transform the layout into an equivalent layout (two layouts are equivalent if they connect the same set of terminal pairs). The modification is based on exchanging wire segments between two nets that cross each other in (at least) two points. Figure 2.6 is an example. (More details can be found in [PrL]). In terms of the diagonal diagram, the transformation corresponds to changing two crossings into knock-knees, i.e., to the introduction of two diagonals. Figure 2.7.(a) is a example of diagonal diagram and Figure 2.7.(b) shows the corresponding legal partition by adding vertical line segments. After the above operations (modify layout and add vertical line segments to construct partition), the properties stated in Theorem 2.3 will always be satisfied.

**Theorem 2.4:** ([PrL]) Any Channel Routing Problem of density $d$ can be wired with three layers within a channel of capacity $d$.

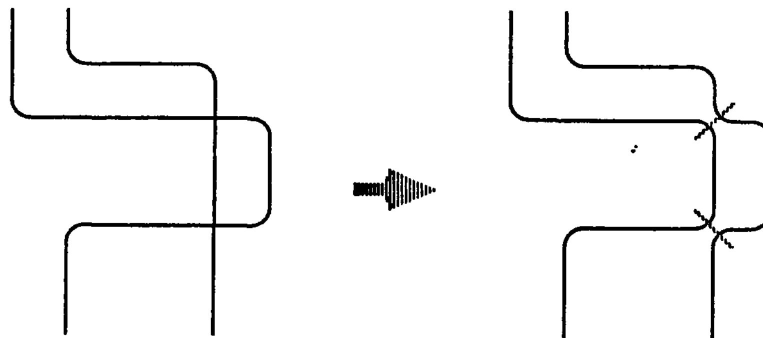


Figure 2.6: Changing in the wiring of two nets crossing in at least two grid-points

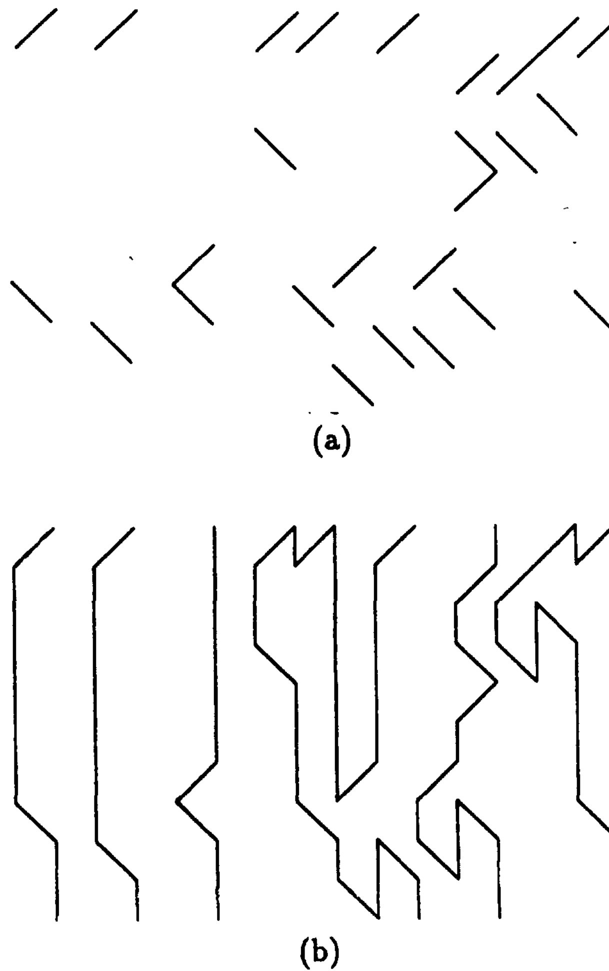


Figure 2.7: (a) Diagonal diagram (b) legal partition

CHAPTER
# THREE

# RIVER ROUTING BETWEEN RECTANGLES

## 3.1 Introduction

It is well-known that many of the optimization problems arising in VLSI routing are NP-complete (e.g. [KrL],[LaP],[SaB],[Szy]). As we have mentioned in chapter two, one exception is the class of *river routing* problems associated with a hierarchical layout strategy such as Bristle-Blocks([Joh]). See ([CoS],[DKS],[LeM],[LeP],[Mir],[Pin],[SeD],[Tom]) for more examples. Efficient serial solutions have already appeared in the literature for most of these problems.

In this chapter, fast parallel algorithms for several river routing problems are presented. In particular, we develop $O(\frac{n}{p} + \log n)$ algorithms for the *separation problem* and $O(\frac{n \log n}{p} + \log^2 n)$ algorithms for the *offset problem* on the CREW PRAM model. These algorithms are efficient in the sense that the number of processors used is $1 \leq p \leq n$, where $n$ is the size of the input. Most of the known algorithms seem to be inherently sequential and new techniques are developed to obtain these parallel algorithms.

The rest of this chapter is organized as follows. The main problems are

introduced in the next section, while section 3.3 deals with the separation problem. The offset problem is considered in section 3.4.

## 3.2 Definitions

The class of general river routing problems involves routing between ordered sequences of terminals such that the final layout is planar. One such problem is the wiring of two ordered sets of terminals $\{b_1, b_2, \ldots, b_n\}$ and $\{t_1, t_2, \ldots, t_n\}$ across a channel between the parallel boundaries of two rectangles. The width of the channel is the vertical distance between the two lines forming the 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 unit distance. We will restrict ourselves to the case where the wires are rectilinear, i.e., there is a grid structure such that each wire consists of a set of grid line segments. Our methods generalize for all the other known variations ([SeD],[Tom]). The *offset problem* is to find the distance to slide one of the rectangles so as to achieve the minimum possible separation between the rectangles.

## 3.3 The Separation Problem

Let $\{N_i =< b_i, t_i > \mid 1 \leq i \leq n\}$ be an instance of the channel separation problem. Notice that $b_i$ and $t_i$ will be also used to denote the horizontal coordinates of the terminals relative to an arbitrary origin. A net $N_i$ is a *right net* if $b_i < t_i$. If $b_i > t_i$, then $N_i$ is a *left net.* Otherwise, it is a *vertical net.* We can partition the nets into *right blocks*, *left blocks* and *vertical blocks.* A set of right nets $N_i, N_{i+1}, \ldots, N_p$ is a right block if it is a maximal block with the property $b_k < b_{k+1} \leq t_k$, for any $i \leq k < p$. We can similarly define left blocks

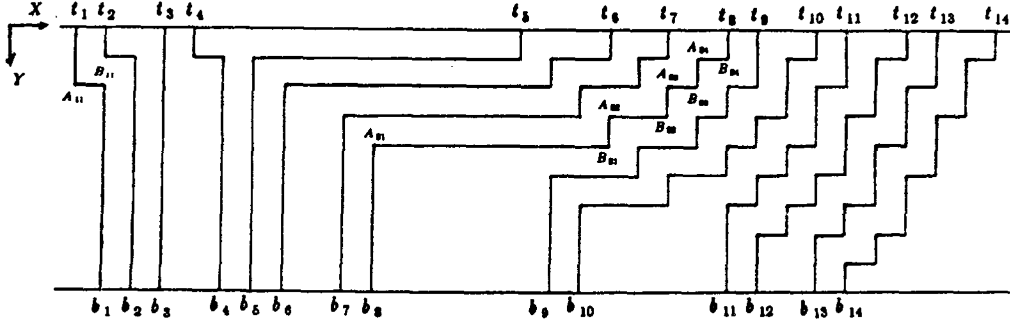



Figure 3.1: Basic river routing problem

and vertical blocks.

For any right block $\{N_i, N_{i+1}, \ldots, N_p\}$, define the *rank* of net $N_k$ to be $k - i + 1$, $i \leq k \leq p$. For a left block $\{N_i, N_{i+1}, \ldots, N_p\}$, the *rank* of net $N_k$ is defined to be $p - k + 1$, $i \leq k \leq p$.

The wiring problem is reduced to wiring each block separately. We will concentrate on the wiring of right blocks. Obvious changes can be made to deduce the corresponding algorithm for left blocks.

The wiring of a net can be specified by the coordinates of its bend points. For example, net $N_1$ of Figure 3.1 has the bend points $A_{11}, B_{11}$. For each net $N_i$, we have $2k$ bend points, $A_{i1}, A_{i2}, \ldots, A_{ik}$ and $B_{i1}, B_{i2}, \ldots, B_{ik}$, for some $k$. Not all of these bend points are needed to determine the overall wiring. Let's call $A_{i1}$ and $B_{i1}$ (bend points closest to the bottom row) the ***characteristic*** bend points and all the others ***ordinary*** bend points. Notice that the characteristic bend points uniquely define the overall wiring since once we have the wiring of $N_{i-1}$ and the characteristic bend points $A_{i1}$ and $B_{i1}$, we can determine all the ordinary bend points of $N_i$ very easily. Figure 3.1 shows an example of a river routing problem and a wiring achieving the minimum separation.

The algorithm to find the minimum separation is based on the following lemma.

**Lemma 3.1**: 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 bend points of $N_i$ are $A_{i1} = (b_i, i - \hat{j} + 1)$ and $B_{i1} = (t_{\hat{j}} + i - \hat{j}, i - \hat{j} + 1)$.

**Proof**: Since $\hat{j}$ is the minimum $j \leq i$ such that $t_j + (i - j + 1) \geq b_j$, there is no terminal point at $(t_{\hat{j}} - 1, 0)$. Hence there is a bend point at $(t_{\hat{j}}, 1)$ for net $N_{\hat{j}}$. The number of vertical grid lines between $t_{\hat{j}}$ and $b_i$ is $b_i - t_{\hat{j}}$ and hence smaller than the number of nets between $N_{\hat{j}}$ and $N_i$,i.e., $i - \hat{j} + 1$ horizontal tracks are needed to route net $N_i$. A simple argument will show that the coordinates of the characteristic bend points have the values stated in the Lemma.

We now show how to compute in parallel the index $\hat{j}(i)$ for each $i$. A simple binary search method will yield a CREW PRAM algorithm that runs in $O(\log\ n)$ time with $O(n)$ processors. However this does not seem to yield an optimal algorithm when implemented on the mesh. We use the following procedure.

*Algorithm Index*

*input*: A set of nets $< b_i, t_i >$, $1 \leq i \leq n$, forming a right block.

*output*: $\hat{j}(i)$ such that $\hat{j}(i)$ is the minimum $j$ such that $b_i - t_j \leq i - j - 1$, for each $1 \leq i \leq n$.

1. Compute $b'_i = b_i - i$ and $t'_j = t_j - j - 1$ for each $i$ and $j$.

2. Sort the $t'_j$s, say $t_{p_1} \leq t_{p_2} \leq \ldots \leq t_{p_n}$.

3. For each $p_i$, determine $f(p_i) = min\{p_k | i \leq k \leq n\}$.

4. Sort the $b'_i$s and the $t'_j$s such that if a $b'_i = t'_j$, the $b'_i$ is pushed to the lower rank.

5. For each $b'_i$, let $t'_{p_j}$ be the closest $t'_{p_k} \geq b'_i$. Then $f(p_j) = \hat{j}(i)$.

The correctness proof of the above is straightforward and will be left to the reader. We now give the algorithm to find the minimum separation as well as the characteristic bend points of all the nets.

*Algorithm Separation*

*Input*: A set of $n$ nets given by $\{N_i =< b_i, t_i > |1 \leq i \leq n\}$.

*Output*: The characteristic bend points and the minimum separation.

1. Partition the nets into blocks by creating the following chains.

**For** each $i$ **do**

**If** $t_i > b_i$ and $b_i < b_{i+1} \leq t_i$, **then** $Next(N_{i+1}) = N_i$.

**If** $t_i < b_i$ and $t_i \leq t_{i+1} \leq b_i$, **then** $Next(N_i) = N_{i+1}$

For each net, determine the sink reachable from this net and its corresponding rank.

2. Apply Algorithm Index to get the index $\hat{j}(i)$ for each $i$. Use Lemma 3.1 to obtain all the characteristic bend points.

3. Let the characteristic bend points be $B_{i1} = (x_{i1}, y_{i1})$, $1 \leq i \leq n$. Then the minimum separation is $d + 1$, where $d = max\{y_{11}, \ldots, y_{n1}\}$.

**Theorem 3.1** : Algorithm Separation correctly finds the characteristic bend points of the $n$ input nets as well as the minimum channel separation. This algorithm can be implemented on an $O(n)$ processor CREW PRAM with the corresponding running time of $O(\log\ n)$. If all the terminals lie in an interval

$[1, N]$, where $N = O(n)$, then Algorithm Separation runs in time $O(\frac{n}{p} + \log n)$ time with $p$ processors, $1 \leq p \leq n$, on the PRAM.

**Proof**: The correctness proof follows from the discussion at the beginning of this section (See also [DKS], [LeP] and [Tom]). Step 1 of the algorithm uses the path doubling technique. The basic operations in step 2 are sorting, prefix computation and some constant time operations. Step 3 can be done by the prefix computation. These conclude the time complexity analysis.

If all the terminals lie in an interval $[1, N]$, where $N = O(n)$, and at most two terminals have same x-coordinate, then path doubling, list ranking and prefix computation can be done in time $O(\frac{n}{p} + \log n)$ time with $p$ processors, $1 \leq p \leq n$, on the PRAM, as mentioned in chapter one. Sorting also can be done with same time complexity by using parallel bucket sort and prefix computation theorem. This concludes the $O(\frac{n}{p} + \log n)$ time complexity.

From the characteristic bend points we can easily obtain all the bend points of all the nets.

## 3.4 The Offset Problem

Given two sequences of terminals $t_1, t_2, \ldots, t_n$ and $b_1, b_2, \ldots, b_n$ to be wired through a channel, the horizontal displacement of the $b_i$'s relative to the $t_i$'s is called the *offset*. Our goal in this section is to develop a fast parallel algorithm to compute the offset that minimizes the separation needed to wire all the nets.

Assume that the $t_i$'s are fixed. The $b_i$'s should be moved to the right or left so as to achieve the smallest possible separation. As before, partition the nets into blocks and let $D_r$ be the largest separation of a right block, say $R$, and

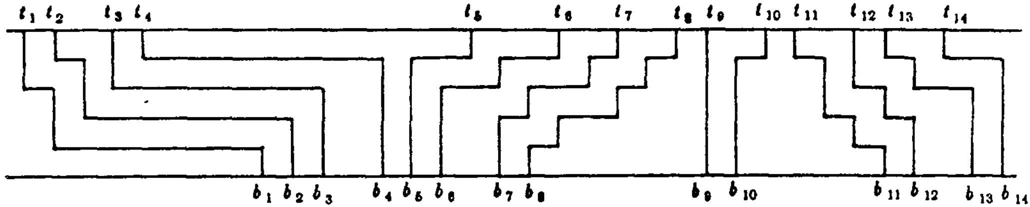



Figure 3.2: Minimum separation with offset 7

let $D_l$ be the largest separation of a left block, say $L$. We need the following fact established in ([DKS]).

**Lemma 3.2**: Sliding a left block to the right will not decrease the required separation for wiring this block. Similarly, sliding a right block to the left cannot decrease the width of the channel required by that block. In particular, if $D_l = D_r$, the current offset achieves the minimum possible channel width.

One can check that for the routing problem in Figure 1, $D_l = 3$ and $D_r = 9$. Figure 2 shows the wiring obtained with an offset of $k = 7$. In this case, $D'_l = D'_r = 5$.

Assume without loss of generality that $D_r > D_l$. Notice that $D_r \leq n$. Suppose that we want to find out whether there exists an offset $k$ such that the resulting separation is not greater than $d$, $D_l \leq d \leq D_r$. Since the $t_i$'s are fixed, each $b_i$ will have to be restricted to a certain fixed interval. Following arguments similar to those used in the previous section, it is not hard to see that $b_i$ has to satisfy the following two conditions.

$$b_i + k - t_{i-(d-1)} \geq d - 1 \quad (1)$$

$$t_{i+d-1} - b_i - k \geq d - 1 \quad (2)$$

It is clear that 1 and 2 are equivalent to

$$\alpha_{id} \leq k \leq \beta_{id} \quad (3)$$

where $\alpha_{id}$ and $\beta_{id}$ are given by

$$\alpha_{id} = d - 1 - (b_i - t_{i-d-1})$$

$$\beta_{id} = (t_{i+d-1} - b_i) - (d - 1)$$

Such a $k$ exists if, and only if,

$$max_i \alpha_{id} \leq min_i \beta_{id} \quad (4)$$

**Lemma 3.3**: An offset of $k$ which induces a separation of $d$ exists if, and only if, condition (4) is satisfied for each $i$.

**Proof**: It is clear that (4) is equivalent to (1) and (2). A set of nets can be river routed within a channel of width $d$ if, and only if, for each net $i$, the

distances between $b_i$ and $t_{i-(d-1)}$ and between $b_i$ and $t_{i+d-1}$ are large enough to accommodate the $d-1$ nets between $t_i$ and $t_{i-(d-1)}$ and between $t_i$ and $t_{i+d-1}$, i.e., if and only if, the number of vertical tracks is at least $d-1$.

A straightforward binary search over the interval $(D_l, D_r)$ using the above lemma will yield an optimal offset in $O(\log\ n)$ iterations. On a sequential machine each iteration will require $O(n)$ time, while on a PRAM with $1 \leq p \leq n$ processors each will require $O(\frac{n}{p} + \log n)$ time. Therefore we have the following theorem.

**Theorem 3.2**: Finding an offset that generates the minimum separation can be done on a CREW PRAM in time $O(\frac{n \log n}{p} + \log^2 n)$ with $p$ processors, where $1 \leq p \leq n$.

CHAPTER
# FOUR

# ROUTING WITHIN A RECTILINEAR POLYGON

## 4.1 Introduction

A more general version of the river routing problem that is known to have an efficient serial algorithm is to perform planar routing where the ports lie on the boundary of a simple rectilinear polygon ([Pin]). In this case, we are interested in whether the routing is possible or not and, if it is possible, we have to provide the detailed routing. Several interesting subproblems such as finding the contour of the union of a set of rectilinear polygons or determining whether a set of nets can be wired within a set "passages" are also tackled.

In this chapter, we present $O(\frac{n}{p}+\log n)$ algorithms to detect the routability and $O(\frac{n}{p}+\log n)$ algorithms to determine the detailed routing with $1 \leq p \leq n$ processors on the CREW PRAM model. Most of the known algorithms seem to be inherently sequential and new techniques are developed to obtain these parallel algorithms.

The rest of this chapter is organized as follows. The main problems are introduced in the next section, while section 4.3 deals with the detailed routing. The routability testing is considered in section 4.4.

## 4.2 Definitions

The routing problem of nets within a simple rectilinear polygon introduced in([Pin]) is a generalization of the standard river routing problem. In this case we are supposed to connect a set of terminals $a_1, a_2, \ldots, a_n$ on the boundary of a simple rectilinear polygon to another set of terminals $b_1, b_2, \ldots, b_n$ on the boundary of the same polygon such that all the wires lie within the polygon and no two wires intersect. *Routability testing* is to determine whether or not a one layer routing is possible and *detailed routing* is to specify the actual wiring of the $n$ nets, if they are routable. In the serial case, a simple greedy strategy ([Pin]) produces an optimal solution for detailed routing. However, such a scheme is inherently sequential and hence an alternative method should be developed for the parallel case. We will start by discussing the detailed routing problem whose solution will be used in the routability testing algorithm.

Our strategy for the detailed routing problem will consist on identifying a set of critical nets and then performing the wiring of each such net with the nets "covered" by it separately. It turns out that these special nets will also play a crucial role in the routability testing problem.We will restrict ourselves to the rectangle case. However all the algorithms can be generalized to any rectilinear polygon. As in the previous chapter, we will assume that the $x$ and $y$ coordinates of the terminals are integers which lie in an interval $[0, N]$, where $N = O(n)$.

## 4.3 Detailed Routing

We will begin with few definitions. Given the set of nets $\{N_i =< a_i, b_i > \mid 1 \leq i \leq n\}$ whose terminals lie on the boundary of a rectangle $R$, we let

the lower left corner of $R$ be the origin of an $(x, y)$ coordinate system. The four corners of $R$ have coordinates $(0,0)$,$(l,0)$,$(l,h)$ and $(0,h)$, where $l$ and $h$ are respectively the length and height of the rectangle. If we cut $R$ at $(0,0)$ and straighten the boundary clockwise into a line, the corresponding linear coordinate of a point $P$ on the boundary will be denoted by $d(P)$. It is trivial to compute $d(P)$ from the two-dimensional coordinates of $P$.

Let $N_i =< a_i, b_i >$ be an arbitrary net. The terminals $a_i$ and $b_i$ divide the boundary of $R$ into two parts. The part of length $\leq h + l$ will be called the *internal boundary* of $N_i$. The other part will be called the *external boundary.* A net $N_i$ is *covered* by another net $N_j$ if the terminals of $N_j$ are in the external boundary of $N_i$ and the terminals of $N_i$ are in the internal boundary of $N_j$. A *representative net* is a net that is not covered by any other net. Figure 4.1 shows an example of a detailed routing problem such that $N_1$, $N_6$ and $N_{14}$ are the representative nets. We can partition the nets into *groups* such that each group consists of a representative net and all the nets covered by it. The groups in Figure 4.1 are $\{N_1, N_2, N_3, N_4, N_5\}$,$\{N_6, N_7, N_8, N_9, N_{10}, N_{11}, N_{12}, N_{13}\}$, and $\{N_{14}, N_{15}\}$.

Given a net $N_i$, trace the boundary of $R$ counterclockwise starting from any point on the external boundary of the net, the terminal that we meet first is the *left* terminal, while the other is the *right* terminal. The net adjacent to the left terminal will be called the *left* net of $N_i$, while the net adjacent to the right terminal will be called the *right* net of $N_i$. For example, in Figure 4.1, $a_4$ and $b_4$ are respectively the left and right terminals of net $N_4$. $N_2$ is the left net and $N_1$ is the right net of $N_4$.

**Lemma 4.1**: Suppose a given instance of the above problem is routable. Then the routing can be performed by routing each group of nets separately.

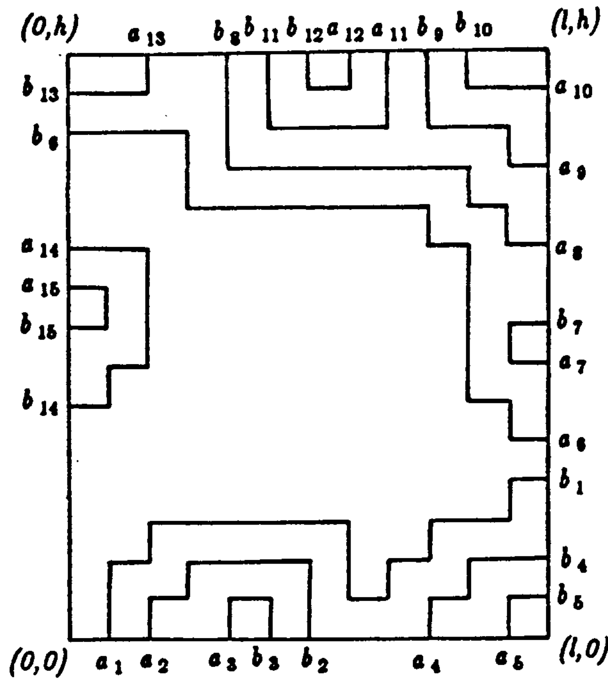


Figure 4.1: Basic river routing around a rectangle boundary

**Proof:** Take any net $N$ that does not cover any other net. Route it as close to the boundary as possible. Remove $N$ and consider the contour formed by the new wiring and the old boundary. Repeat the procedure until all the nets are wired. It is clear this strategy will result in a legal wiring if the nets are routable. Hence each group of nets can be routed separately.

The general strategy for specifying the routing will be the following: (i) identify the proper groups , (ii) find the representative nets, and (iii) specify the routing of the representative net of each group. We will start by tackling (i) and (ii). Essentially we want to create a chain of the nets involved in each group such that a representative net is a sink. The terminals of each net $N_i = < a_i, b_i >$ will be labeled as follows. Let $N_k = < x, y >$ be the net in the external part of $N_i$ with $x$ closest to $a_i$. If $a_i$ lies in the internal part of $N_k$, then set $label(a_i) = 0$ ($N_i$ is not a representative net). Else, $a_i$ and $b_i$ lie in the external part of $N_k$.

If the distance between $y$ and $b_i$ in the external part of $N_k$ is $\leq h + l$, then set $label(a_i) = i$ ($N_i$ is a representative net). Otherwise, set $label(a_i) = k$ ($N_i$ is a representative net if and only if $N_k$ is). In a similar fashion we can determine $label(b_i)$. For example, in Figure 4.1, $label(a_2) = 0$, $label(b_2) = 4$, $label(a_1) = 14$ and $label(b_1) = 1$. Now chains whose nodes are the nets are created by setting the link of an arbitrary net $N_i = < a_i, b_i >$ as follows.

- If $label(a_i) = i$ or $label(b_i) = i$, then set $link(N_i) = N_i$.
- If $label(a_i) = 0$ or $label(b_i) = 0$, then set $link(N_i) = 0$.
- Else set $link(N_i) = N_{label(a_i)}$, assuming $a_i$ is the left terminal.

The lists created above are chains with possibly one of them being a circular list. We can identify the circular list (if any) fast by a path doubling technique and break it arbitrarily. From now on, we assume that we have a set of chains. Referring to Figure 4.1 again, we have $link(N_1) = N_1$, $link(N_6) = N_6$, $link(N_8) = N_7$, $link(N_{14}) = N_{14}$, and $link(N_i) = 0$ for all the remaining nets. The following lemma holds.

**Lemma 4.2**: A net $N_i$ is a representative net if and only if the sink reachable is not 0

**Proof:** Suppose $N_i$ is not a representative net. Then there exists another net $N_k$ that covers it. Choose $N_k$ to be closest to one of the terminals of $N_i$. If two terminals of $N_i$ and $N_k$ are adjacent, then it is easy to check that $link(N_i) = 0$. Otherwise there are several nets between $N_i$ and $N_k$. $N_i$ will be linked to some of these nets such that the last one on the chain is adjacent to $N_k$, i.e., the sink will be labeled 0.

**Lemma 4.3**: Let $N_{r1}, N_{r_2}, \ldots, N_{r_k}$ be all the representative nets and let $R(N_{r_i})$ be the number of nets in the internal boundary of $N_{r_i}$. Then $\sum_{i=1}^{k}(R(N_{r_i}) +$

1) $= n$. Moreover, there exists a wiring strategy such that $N_{r_i}$ has at most $2(R(N_{r_i}) + 1)$ bend points.

The proof is simple and will be omitted.

**Corollary:** The total number of bend points of all the representative nets is $O(n)$, where $n$ is the number of nets.

We are now ready to state the algorithm to identify the representative nets and all the nets within each group.

*Algorithm Groups and Representatives*

*Input*: A set of nets $N_i$ specified by their terminals.

*Output*: The groups and their representatives.

1. Compute the $d$-coordinate value of each terminal and sort these values.

2. Determine $label(x)$ for each terminal $x$ and set up the corresponding chains breaking the circular list if it exists.

3. For each node, determine the sink reachable from that node. Determine all the representative nets.

4. Create new chains as follows. Link each *nonrepresentative* net point to its left net. Make each *representative* net into a sink. Identify the corresponding groups and compute $R(N_{r_i})$ for each representative net $N_{r_i}$.

**Lemma 4.4**: Let $n$ be the number of nets. Then the above algorithm can be implemented to run in time $O(\frac{n}{p} + \log\ n)$ with $p$ processors on the PRAM.

**Proof:** The fast efficient algorithms for sorting, path doubling and prefix computation can be used to obtain the run time stated in the lemma.

We now turn to the problem of routing each group separately. Our goal here is to identify the bend points of each representative net. Note that in general the total number of bend points of all the nets could be $\Omega(n^2)$, where $n$ is the number of nets. However the total number of bend points of the representative nets is always $O(n)$.

Let $N =< x, y >$ be a net in a group whose representative is $N_r$. Let $k$ be the number of nets between $N$ and $N_r$, including both $N$ and $N_r$. The *bounding perimeter of rank k* is the rectilinear boundary of the region determined by $N$ such that the wiring of $N_r$ cannot lie inside it, i.e., this is the boundary of the region within the rectangle of all the points of distance $\leq k$ of the rectangle boundary determined by $N$. Consider again the case of Figure 4.1. Let $B_i$ be the bounding perimeter induced by net $N_i$. Figure 4.2 shows the contours $B_3$, $B_5$, $B_2$, $B_4$ and $B_1$. We claim that the following lemma is true.

**Lemma 4.5**: The union of all the bounding perimeters of all the nets within a group determines the contour of the group and hence determines the wiring of the representative net.

**Proof:** The proof is by induction on the number of nets within a given group. The basis $k = 1, 2$ can be checked easily.

Suppose that the lemma holds for groups with $k > 2$ nets. Let $N$ be the $(k+1)$st net added to a given group. Define the rank of a net to be the number of nets between the representative net $N_r$ and the net. If the rank of $N$ is $\leq 2$ it is easy to check by inspection that the lemma holds. Suppose the rank of $N$ is greater than 2. Then pick any net of rank 2 and remove it. Apply the induction hypothesis and put back the removed net. The details can be verified easily.

We now discuss the problem of determining the contour of a group of nets from the corresponding bounding perimeters. Flatten the rectangle into a line

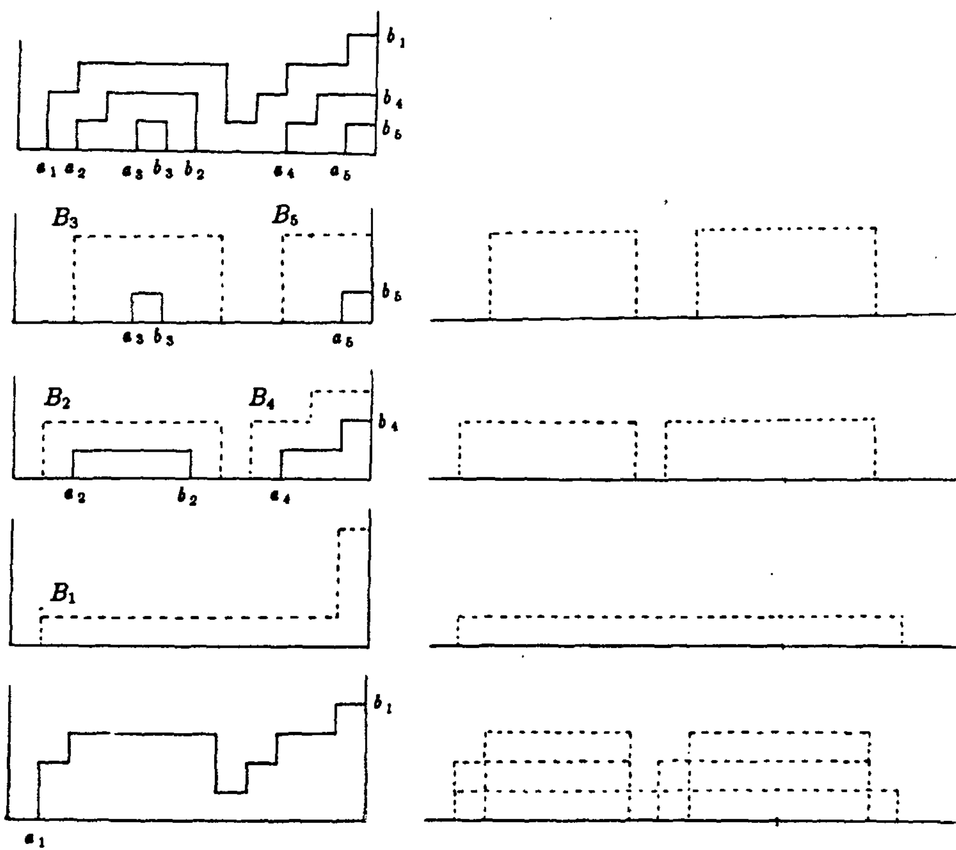



Figure 4.2: The union of all bounding perimeters

as before by making a cut at the left terminal of the representative net. Suppose a terminal $p$ gets mapped into $\bar{p}$. A bounding perimeter connecting $p$ and $q$ of rank $k$ will get mapped into a simple rectangle with endpoints $\bar{p}$ and $\bar{q}$ and height $k$. Denote the mapped bounding perimeters by $\bar{R}_1, \bar{R}_2, \ldots, \bar{R}_l$. These rectangles determine a contour $R$ given by the d-coordinates of its extreme points. Each such extreme point $(p, h)$, where $p$ is the d-coordinate and $h$ is the distance from the line, is mapped into $(p', h)$, where $p'$ is on the boundary of the rectangle corresponding to the point of d-coordinate $p$ on the line. Few of these points around the corners may not be mapped into extreme points of the contour within the rectangle, but rather onto the boundary. These can be determined quickly and then eliminated. We are now ready to state the algorithm.

*Algorithm Contour*

*Input:* A group of nets with their representative.

*Output:* The bendpoints of the corresponding contour.

1. Assign a weight of 1 to the left terminal and -1 to the right terminal of each net. Order the terminals counterclockwise starting from the left terminal of the representative net. For each terminal, compute the sum of the weights of all terminals proceeding it, including itself. Call that sum *the rank* of the terminal. The rank of a net is the rank of its left terminal (number of nets between itself and the representative net including both).

2. Determine all the bounding perimeters.

3. Cut the boundary of the rectangle at the left terminal of the representative net and stretch it clockwise into a line. Map the bounding perimeters into this line . Each corresponding perimeter can be identified by $(\bar{p}, \bar{q}, k)$.

4. Sort the triplets $(\bar{p}, \bar{q}, k)$ according to $k$. For each $k$, determine the union of line segments at distance $k$.

5. From each line segment generated at step 4, determine the corresponding bendpoints. The overall contour can be specified by the bend points.

6. Map the bend points of the contour on the line back into the rectangle. Eliminate those points within the rectangle which are not bend points.

**Lemma 4.6**: If the number of nets in the group is $n$, then algorithm contour can be implemented in time $O(\frac{n}{p} + \log n)$ with $p$ processors.

The proof is easy and will be omitted.

The main result of this section is stated in the following theorem.

**Theorem 4.1**: Detailed routing of the representative nets of $n$ nets within a simple rectilinear polygon can be done in $O(\frac{n}{p} + \log n)$ time with $p$ processors, $1 \leq p \leq n$, on the PRAM.

## 4.4 Routability Testing

In this subsection we address the problem of testing whether or not it is possible to route all the nets within the given rectangle without actually determining the routing. The above algorithm assumes that the nets are routable. Notice that the detailed routing algorithm does not generate enough information to do the routability testing since it only produces the bend points of the representative nets. However this algorithm will be part of the routability testing algorithm as we will see in this section. The resulting algorithm will have the same time performance as the detailed routing algorithm. Several subproblems

which had to be solved are interesting in their own and require new parallel methods for their solutions.

Given a set of nets $\{N_i =< a_i, b_i >,\ 1 \leq i \leq n\ \}$, where the terminals lie on the boundary of the rectangle $R$, these nets may be unroutable for one of the following reasons:

- The graph determined by the nets when restricted to lie within the rectangle is *nonplanar*.
- The wiring of all the nets requires more area.

We start by handling the first issue (easy case). The algorithm tests for any given net $N =< x, y >$, whether any net with a terminal in the internal boundary of $N$ has the other terminal in the external boundary of $N$.

*Algorithm Interconnect Planarity*

*Input:* A set of nets $\{N_i =< a_i, b_i >, 1 \leq i \leq n\}$, where the terminals lie on the boundary of a rectangle.

*Output:* Determine whether the interconnect pattern is planar.

1. Compute the d-coordinates of all the terminals. Sort the terminals according to these coordinates.

2. For each net $N_i =< a_i, b_i >$ assign the following weights to its terminals: if $d(a_i) > d(b_i)$, then $w(b_i) = 1$ and $w(a_i) = -1$; else, $w(b_i) = -1$ and $w(a_i) = 1$.

3. Calculate the rank (i.e., sum of weights of all predecessors) of each terminal.

4. Output no if there exists a net $N =< x, y >$ (say $d(x) > d(y)$) such that $rank(y) \neq rank(x) + 1$. Otherwise output yes.

**Lemma 4.7:** The above algorithm correctly checks whether the interconnection pattern of the given nets is planar. This algorithm runs in time $O(\frac{n}{p} + \log n)$ time with $p$ processors, $1 \leq p \leq n$.

Before we tackle the question of whether the rectangle is large enough to accommodate all the nets, several definitions are needed. A *single side* net is a net whose terminals lie on the same side of the rectangle. If the terminals lie on adjacent sides then the net is called *corner* net. It is a *cross* net if the terminals lie on opposite sides. Partition the single side nets corresponding to each specific side into *single side blocks* such that each net except one (*cover net*) is covered by one or more nets in the block. Moreover each such block is maximal. A *corner block* is a maximal set of corner nets corresponding to the same corner such that each net except one (*cover net*) is covered by one or more nets within the block. Moreover no other net outside a block is covered by the cover net. For example, in Figure 4.1, $N_2$ is a single side net, $N_1$ is a corner net and $N_6$ is a cross net. The single side blocks are $\{N_2, N_3\}$, $\{N_7\}$, $\{N_{11}, N_{12}\}$ and $\{N_{14}, N_{15}\}$, whose corresponding cover nets are $N_2$,$N_7$,$N_{11}$ and $N_{14}$. $\{N_4, N_5\}$, $\{N_9, N_{10}\}$ and $\{N_{13}\}$ are the corner blocks with $N_4$,$N_9$, $N_{13}$ as the corresponding cover nets.

We start by determining the separate minimal wirings (i.e. as close to the boundary as possible) of the cover nets.

*Algorithm Wiring of Blocks*

*Input:* A set of nets whose terminals lie on the boundary of a rectangle and whose interconnection pattern is planar.

*Output:* The wiring of each cover net.

1. Determine the single side and the corner blocks. For each block, specify the cover net.

2. Determine the wiring of all single side cover nets and all the corner cover nets by using algorithm Contour.

**Lemma 4.8**: The above algorithm correctly determines the minimal wiring of each cover net in time $O(\frac{n}{p} + \log n)$ with $p$ processors, $1 \leq p \leq n$.

Once the block cover nets are wired, it should check if there is enough space to route the remaining nets. The techniques developed below could be used to determine whether any wiring of two nets is legal (i.e. the corresponding wires do not intersect). Our approach consists of determining *the wiring capacity* and the *wiring density* between blocks. The wiring capacity between two blocks is the number of nets that can be wired between these two blocks, while the wiring density is the number of wires that have to be wired between these two blocks. Determining the density is relatively easy and will be outlined in the algorithm below. The capacity between blocks on two orthogonal sides of the rectangle boundary is computed as follows. Given a block $B$ consider all the convex corners of $B$. Generate 45 degree "rays" from each such corner and determine the line segment where it intersects another block contour or the original rectangle boundary. Based on this information, one can determine the width of the narrowest passage between $B$ and any other block. The details are provided in the algorithm below.

*Algorithm Intersection*

*Input:* Contours of single side and corner blocks on two orthogonal sides of rectangle boundary.

*Output:* Intersection points of rays emanating from convex corners.

1. Consider the case of the the lower right corner. The other cases can be dealt with in a similar fashion. Sort all the line segments determined by the block

contours and the right side of the rectangle $R$. Determine the projection of each line segment on the diagonal (45 degree ray emanating from the lower right corner), say line segment $i$ is projected into line segment $p(i)$ on the diagonal.

2. Sort the projections according to their order on the diagonal and compute $p'(i) = p(i) - \bigcup_{j=1}^{i-1} p(i)$.

3. For each ray $y$ coming out of a corner of contour on the horizontal side of the original rectangle, find its intersection with the diagonal. If the intersection point lies in $p'(j)$, then ray $y$ intersects segment $j$. Determine the intersection point of ray $y$ and line segment $j$.

4. If a ray $y$ intersects the original rectangle boundary, then rotate to find the intersection with the next line segment belonging to some block contour (see Figure 4.3 and ray $y_B$ ). Now determine the point of intersection.

For example, one can check that in Figure 4.3 $p'(CD) = C'D'$ and $p'(EF) = D'F'$. Hence rays $y_A$ and $y_B$ intersect $CD$ and $EF$ respectively. If we rotate $y_B$, we can find the intersection with the next line segment $GF$.

**Lemma 4.9**: The above algorithm finds the intersection points of rays emanating from convex corners with the line segments of contours on two orthogonal sides of the bounding rectangle in time $O(\frac{n}{p} + \log n)$ with $p$ processors, $1 \leq p \leq n$.

**Proof:** Since the nets are on two orthogonal sides, it is clear that the line segment intersected by a ray can be determined by the projections on the diagonal. The running time of the algorithm can be easily verified.

*Algorithm Density and Capacity*

*Input:* The wiring of cover nets of the blocks and the terminals of the remaining nets.

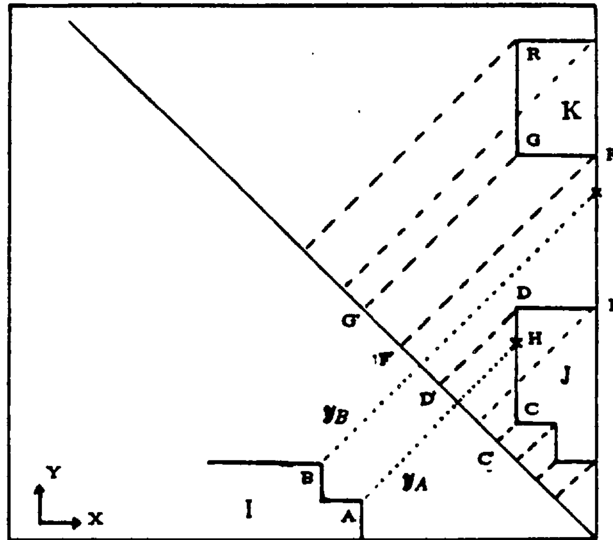



Figure 4.3: Intersection between rays and block contours

*Output:* Determine whether or not there is enough space between the contours of the block cover nets on orthogonal sides to wire the remaining nets

1. Assume that the d-coordinates of all terminals are available. Cut the rectangle $R$ at the left terminal A of a cover net such that $d(A)$ is minimum and straighten it into a line. Let $d'$ be the new coordinate system. Determine $d'(P)$ for all terminals $P$.

2. Assign weights to the terminals of each corner or cross net as follows: +1 to the terminal with smaller $d'$-coordinate, -1 otherwise. Order these terminals according to their $d'$-values and compute the *rank* of each terminal. Each block is assigned the *rank* of terminal adjacent to the left terminal of its cover net. The density between two adjacent blocks is equal to the difference in their ranks.

3. Use algorithm Intersection to compute the intersection point of each ray with a single side contour, corner block contour or the original boundary of the rectangle. The capacity between blocks can then be calculated easily.

4. Test whether the capacity is at least as large as the corresponding density. Otherwise the problem is unroutable.

**Lemma 4.10**: Testing the routability of $n$ nets between two orthogonal sides of a rectangle can be done in $O(\frac{n}{p} + \log n)$ time with $p$ processors on the CREW PRAM model.

**Proof:** The correctness proof involves two steps. The first is to show that the density can be computed correctly at step 2 (easy) and the second step is to show that the capacity determined by each ray is enough to determine the routability of nets between orthogonal sides. We will concentrate on the latter step.

A typical case is shown in Figure 4.3. If a ray emitted from the convex corner $A$ intersects at point $H$ of block $J$, then the capacity between $A$ and $J$ is given by $c_{AJ} = max\{|x_A - x_H|, |y_A - y_H|\}$. If $c_{AJ} \geq d_{AJ}$ (density between A and block J), then there is enough space between $A$ and $H$ to route the cross nets. Now consider any convex corner $R$ of a block (say $K$) above $H$ in the right side of the rectangle. Then $c_{AR} = c_{AH} + |y_H - y_R|$. But $d_{AR} \leq d_{AH} + d_{HR}$ and hence $d_{AR} \leq c_{AR}$ since $d_{HR} \leq c_{HR} \leq |y_H - y_R|$. It follows that if it is routable between $I$ and $J$, then it is routable between $I$ and $K$, for any $K$ above $J$. The other cases can be treated similarly.

The running times can be verified easily.

We now address the routability problem between two opposite sides of the bounding rectangle. If there are corner nets or cross nets between any two blocks on opposite sides, then at least one of these nets must be a representative net and hence the problem is unroutable if this representative net intersects another representative net. Therefore we have the following algorithm.

*Algorithm Routability Testing*

*Input:* Contours of single side and corner blocks, and the remaining nets.

*Output:* Whether or not the nets are routable.

1. Use algorithm Density and Capacity to test whether there is enough space between any two block cover nets on orthogonal sides.

2. Generate the wiring of the representative nets using the algorithm developed in the previous subsection. The problem is routable if and only if no two representative nets intersect.

**Theorem 4.2**: Testing the routability of $n$ nets within a simple rectangle could be done in $O(\frac{n}{p} + \log n)$ time with $p$ processors, for any $1 \leq p \leq n$.

**Proof**: We only have to justify the running time of step 2. Suppose we want to check whether any representative net $N$ which has one terminal say on the bottom side intersects another representative net. Sort the horizontal segments of $N$ and the convex corners of all other representative nets by their $y$-coordinates. For each convex corner $a$, let $p(a)$ be the line segment whose $y$-coordinate is the closest to $a$ from below and let $q(a)$ be the line segment closest to $a$ from above. Based on this we can determine whether $a$ is inside the wiring of $N$. A similar strategy will work for all the other cases.

CHAPTER

# FIVE

# WIRING MODULE PINS TO FRAME PADS

## 5.1 Introduction

We consider the problem of connecting a set of pins on a module to a set of pads lying on the boundary of a chip. This problem has been addressed in the sequential context by Baker and Pinter ([BaP]). The boundary of the module is assumed to be an arbitrary rectilinear polygon and the chip is assumed to be a rectangle containing the module. The module pins and the chip pads are fixed. Our goal is to find out whether a *one-layer* routing exists within the given area and if it does to determine such a wiring. We also address the problem of changing a given such wiring so that the resulting wiring is of minimum length.

The known algorithms to solve the above problems seem to be inherently sequential. We develop new algorithms that possess fast and efficient parallel implementations in addition to their suitability for serial implementations.

The rest of this chapter is organized as follows. Several basic definitions and basic strategies will be introduced in the next section. Section 5.3 will contain a new algorithm for routability testing, while the following section will

show how to obtain a minimal length wiring from an arbitrary wiring.

## 5.2 Definitions

An instance of routing problem between module pins and frame pads is given by a triplet $< \mathcal{M}, \mathcal{F}, \mathcal{N} >$, where $\mathcal{M}$ is an arbitrary rectilinear polygon representing a module, $\mathcal{F}$ is a rectangle representing a frame (assuming a horizontal bottom edge), and $\mathcal{N}$ is a set of two-terminal nets such that one terminal is on $\mathcal{M}$ and the other is on $\mathcal{F}$. We assume that $\mathcal{F}$ contains $\mathcal{M}$ and that each boundary segment of $\mathcal{M}$ is parallel to a frame edge. We are supposed to determine a one-layer routing of $\mathcal{N}$ whenever it exists. We borrow some definitions from [BaP]. A *spoke* is a line segment perpendicular to a frame edge and which extends from the frame edge to the closest module edge. It is shown in [BaP] that if a wiring exists, then there is a routing such that no wire crosses any spoke more than twice and that all crossings of a spoke are in the same direction. If the crossings of one spoke for one net are specified, then the crossings of any other net are uniquely defined. Therefore in the rest of this paper we will assume that such information is provided as part of the input since there are five such possibilities and the algorithm could be run on each one of these separately.

As we mentioned in chapter two, the routing strategy used in [BaP] consists of combining the outputs generated by two greedy algorithms. One algorithm (*greedy-in*) begins at the module terminals and routes as close to the module boundaries as possible. The other (*greedy-out*) begins at the frame terminals and stays as close to the frame edges as possible.

Let $n$ be the length of the input (number of nets and boundary segments of the module). The algorithm presented in [BaP] runs in time $O(n^2)$ and finds

a detailed routing of all the nets whenever such a routing exists. This algorithm is optimal since the number of bend points in the detailed routing could be $\Omega(n^2)$. We address three problems in the parallel context:

- Given an instance of our problem, can we route all nets within the space provided?
- Given an instance of our problem, determine a detailed routing of all nets whenever such a routing exists.
- Given a preliminary routing for an instance of our problem, derive a routing with the minimum total wire length.

We will present fast parallel algorithms for all the three problems. As a byproduct, we obtain a new $\Theta(n)$ serial algorithm for the first problem. We present this algorithm in the next section.

## 5.3 Routability Testing

Consider the set of nets whose frame terminals are on the *bottom* frame edge. We will first examine the possibility of routing these nets within the space provided. Recall that we are assuming that the crossings of each net at a given spoke are given as a part of the input. Since we cannot route all the nets fast with a linear number of processors, we have to extract enough information about the wiring of certain critical nets to perform the routability testing. We accomplish this by decomposing the nets into groups such that the routability testing can be done by examining the "contours" of these groups. Before presenting a more formal description of our overall strategy we introduce the following definitions.

Let the *module-bottom line* be the gridline parallel to the bottom frame edge and which passes through points on the module boundary closest to the bottom frame edge. This line will be partitioned into segments, say $A_1B_1$ ,$A_2B_2$ ,...,$A_kB_k$, by the module boundary. Figure 5.1 shows an example whose module-bottom line goes through $A_1B_1$, $A_2B_2$ and $A_3B_3$. The *bottom boundary* consists of the set module segments from $A_1$ to $B_k$ in a counterclockwise direction. We define the *type* of a net $< a, b >$, $a \in \mathcal{M}$, $b \in \mathcal{F}$, as follows.

- *Type 1* if the direction of the wiring from $a$ to $b$ is clockwise and $a$ is not on the bottom boundary of the module. Nets $N_1$, $N_2$, $N_3$, $N_4$, $N_5$ and $N_6$ in Figure 5.1 are type 1 nets[1].

- *Type 2* if $a$ is on the bottom boundary between $B_j$ and $A_{j+1}$, for some $j$, such that the distance (number of module edges) between $a$ and $A_{j+1}$ is greater than or equal the distance between $a$ and $B_j$. Nets $N_8$, $N_{11}$ and $N_{12}$ in Figure 5.1 are of type 2.

- *Type 3* if the direction of the wiring is counterclockwise and $a$ is not on the bottom boundary of the module. Nets $N_{15}$, $N_{16}$ and $N_{17}$ in Figure 5.1 are type 3 nets.

- *Type 4* same as type 2 but distance is less than. In Figure 5.1, nets $N_7$ and $N_{10}$ are of type 4.

- *Type 5* if none of the above, i.e., $a$ is on some segment $A_iB_i$ of the module-bottom line. Nets $N_9$, $N_{13}$ and $N_{14}$ are type 5 nets.

A *consecutive* set $\mathcal{S}$ of nets (with one terminal on the bottom frame edge) consists of nets of the same type such that there is no net of a different type

[1]From now on, we will use the notation $N_i =< a_i, b_i >$.

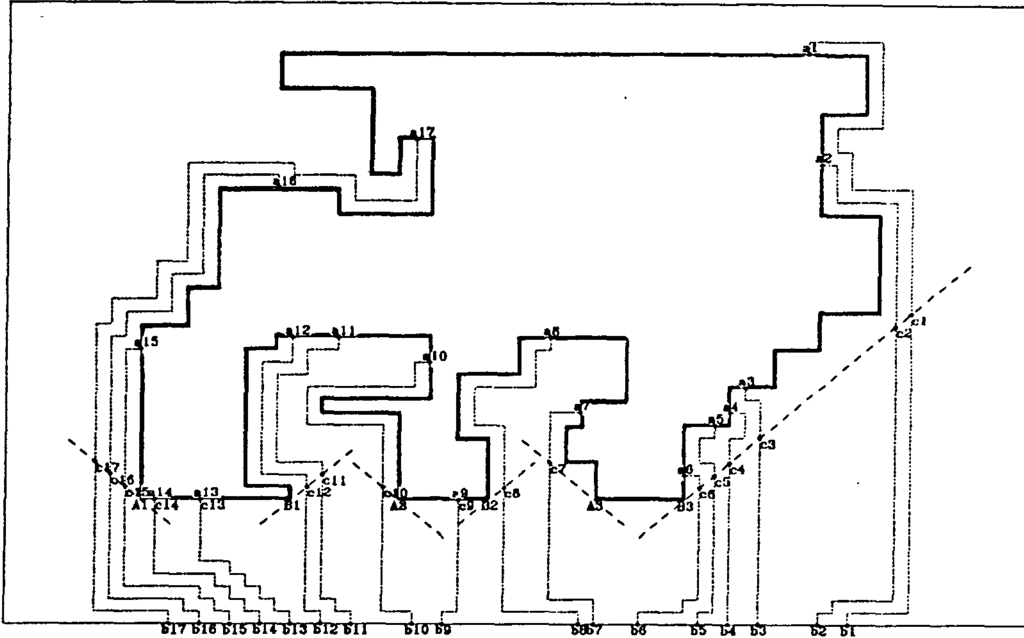


Figure 5.1: Illustration of Different Types of Nets

whose module terminal lies between two terminals of nets in $\mathcal{S}$. A *group* is a consecutive set of nets $\mathcal{S}$ such that either (1) there exists a convex corner $C$ with the property that all convex corners between $C$ and any terminal in $\mathcal{S}$ (counterclockwise direction for types 1 and 2, otherwise clockwise) are above the 45 degree diagonal drawn through $C$ and the diagonal has a positive slope (relative to the bottom frame edge) for types 1 and 2 and negative slope otherwise, or (2) all the terminals of $\mathcal{S}$ are on the same module segment. For example, the groups of Figure 5.1 are given by: $G_1 = \{N_1, N_2, N_3, N_4, N_5, N_6\}$, $G_2 = \{N_7\}$, $G_3 = \{N_8\}$, $G_4 = \{N_{10}\}$, $G_5 = \{N_{11}, N_{12}\}$, $G_6 = \{N_{15}, N_{16}, N_{17}\}$, $G_7 = \{N_9\}$ and $G_8 = \{N_{13}, N_{14}\}$.

We are ready to give an outline of our routability testing algorithm.

*Algorithm Routability Testing*

1. Partition the nets with one terminal on the bottom frame edge into groups

and identify the corresponding corners and diagonals.

2. Determine the outer contour of each group as well as the intersection points (*intermediate terminals*) of the routing with the corresponding diagonal assuming a greedy strategy as close to the module as possible (greedy-in).

3. Move the intermediate points vertically to a horizontal line $L$ such that the separation distance is enough to solve the corresponding river routing problem. Find the characteristic bend points of nets corresponding to the induced river routing problem.

4. Determine if there is any intersection between the wirings of any two different nets or between the wiring of any net and the module or frame boundary.

Notice that if we use the above strategy to produce a *detailed* routing, then the wiring obtained will in general be different from the one generated by the method of [BaP].

In the rest of this section we will present fast and efficient parallel algorithms to perform each of the steps outlined in the routability strategy.

It is not hard to see that step 1 can be easily done in time $O(\frac{n}{p} + \log n)$ time on a CREW-PRAM with $1 \leq p \leq n$ processors. The details will be left to the reader. Let $G_i$ be a group of nets with a 45 degree diagonal. Each net of this group has one terminal on the module boundary and another on the bottom frame edge. Assume that these nets are of type 1 or 2. Similar definitions and algorithms can be developed for the other types of nets. Let $N =< a, b >$ be a net of this group such that $a$ is furthest from the corner of this group. Then $N$ is called the *representative* net of $G_i$. In Figure 5.1, $N_1$, $N_7$, $N_8$, $N_{10}$, $N_{11}$ and $N_{17}$ are the representative nets of $G_1$, $G_2$, $G_3$, $G_4$, $G_5$ and $G_6$ respectively. Notice that the wiring of a representative net will determine the contour of the wirings of all nets in $G_i$ obtained by a greedy-in strategy. Our next goal is to

| | a1 | a2 | a3 | a4 | a5 | a6 |
|---|---|---|---|---|---|---|
| Sequence Number : | 6 | 5 | 4 | 3 | 2 | 1 |

| | A | B | C | D | E |
|---|---|---|---|---|---|
| Bounding Value : | 1 | 4 | 6 | 6 | 5 |

| | N1 | N2 | N3 | N4 | N5 | N6 |
|---|---|---|---|---|---|---|
| Bounding Point : | E | E | B | A | A | A |

Figure 5.2: Determination of Intermediate Terminals

show how to determine this contour and the intermediate terminals (i.e., points of intersections of the wiring with the diagonal).

Each convex corner lying between two terminals of two nets in a group $G$ may introduce a corner in the wiring of a certain net. If the wiring of a certain net intersects the diagonal of the group *before* such a corner is introduced, we say that the intersection takes place *near* that corner. For example, Nets $N_1$ and $N_2$ intersect the diagonal near $E$ in Figure 5.2.

*Algorithm Intermediate Terminals*

*Input*: Corners of module boundary, module terminals of a group of nets (type 1 or 2), and the corner and the diagonal of the group.

*Output*: The intermediate terminal of each net in the group.

1. Rank each terminal of the group according to its order counterclockwise around the boundary. Call the corresponding rank of a net *sequence number*.

2. Calculate the distance between each convex corner and the diagonal of the group. For each convex corner $C$, determine an integer $k$ such that each net with sequence number $\geq k$ intersects the diagonal near $C$. $k$ will be called the *bounding value* of $C$.

3. For each net $N$, determine the closest convex corner $Q$ with bounding value less than or equal its sequence number. $Q$ is called the *bounding point* of $N$. From this information, determine the intermediate terminals of all the nets in the group.

Figure 5.2 shows the lists obtained by the above algorithm for a group of type 1 nets.

**Lemma 5.1**: The above algorithm correctly determines the intermediate terminals of all the nets in the group. It can be implemented to run on the CREW-PRAM in $O(\frac{n}{p} + \log n)$ parallel time with $1 \leq p \leq n$ processors.

**Proof**: The correctness of the algorithm is easy to establish and will be left to the reader. As for the time complexity, it can be estimated as follows.

Step 1 can be done by sorting and a parallel prefix computation.

Step 2 requires few constant time operations.

Step 3 can be done as follows:

- sort corners counterclockwise and rank them.
- sort corners and terminals in increasing order with sequence number or bounding value as the primary key and the ranks obtained from 1 as secondary key.
- For each terminal, find the predecessor which has the maximum rank.

Finding the contour of each group routed by a greedy-in strategy can be done by using the methods of [CJ1].

We can now move all the intermediate terminals vertically to a horizontal line $L$ (parallel to the bottom frame edge) such that the separation distance is enough to solve the corresponding river routing problem. By the method developed in [CJ1], we can find the characteristic bend points as well as the separation needed. This gives enough information to complete the routability testing (Step 4 of Algorithm Routability Testing).

**Lemma 5.2**: If the given problem is routable, then the intersection of wiring of each net produced by the greedy-in and the greedy-out strategies is a point below the intermediate terminal of the net and has the same x-coordinate as the intermediate terminal.

**Proof Sketch**: If the intersection is above the intermediate terminal, then there exists a net $N$ whose greedy-out wiring intersects a corner $C$ before intersecting the wiring generated by the greedy-in strategy. Therefore the given instance is unroutable.

The information obtained above about the wiring of each net can be generated regardless of whether the given instance routable or not. As a matter of fact, a routing exists if and only if there is no intersection between the wirings of any two different nets or between the wiring of a net and the module or the boundary frame. We now discuss how to test for such an intersection.

For each frame edge, the nets whose terminals are the extreme terminals on that edge are called *boundary* nets. There are eight such nets which can be partitioned into four *adjacent* pairs. For example, the right boundary net of the bottom frame edge is adjacent to the bottom boundary net of the right frame edge.

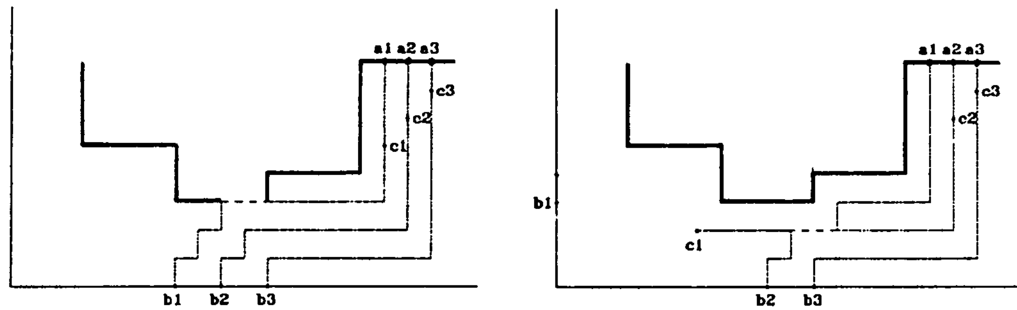



Figure 5.3: Case 1. Intersections

For a given placement, the nets may be unroutable because: (i) the area between module boundary and frame edge is not enough, or (ii) the area between module edges is not enough. Case (i) can be detected by the following intersections:

1. Intersection between (a) wiring generated by greedy-out strategy of those nets with one terminal in a fixed frame edge and (b) module boundary or wiring generated by greedy-in strategy of those nets with one terminal in another frame edge. See Figure 5.3 for an example.

2. Intersection between the wiring generated by greedy-in strategy and frame boundary. See figure 5.4 for an example.

3. Intersection between the wiring of adjacent boundary nets. See Figure 5.5 for an example.

Each of the cases above will be reduced to testing the intersection between two set of line segments. Consider the case of the wiring generated by the greedy-out strategy applied to those nets with one terminal on the bottom frame edge.

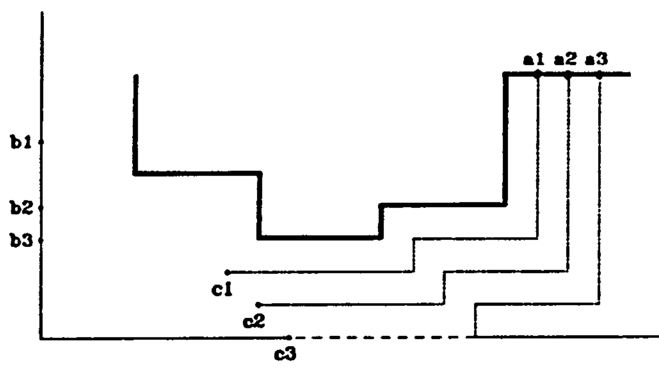



Figure 5.4: Case 2. Intersections

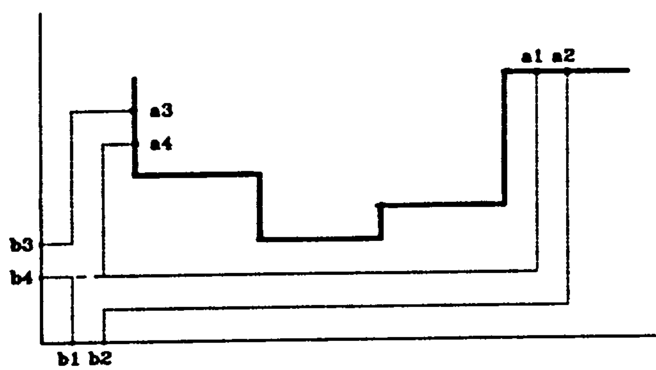



Figure 5.5: Case 3. Intersections

Similar arguments can be used for all the other cases. Let $S_1$ be the set of horizontal segments given by:

1. All horizontal segments between two characteristic bend points of each net.
2. All horizontal wire segments of vertical blocks.
3. All the horizontal wire segments of the right most net of each left block.
4. All the horizontal wire segments of the left most net of each right block.

Let $S_2$ be the set of all vertical wiring segments generated by a greedy-in strategy of the outermost net with no terminal on the bottom frame edge and all the vertical line segments of the module boundary.

**Lemma 5.3**: Case 1 intersection occurs if and only if there exists an intersection between $S_1$ and $S_2$. The number of line segments involved is $O(n)$ and hence the testing can be done in time $O(\frac{n \log n}{p} + \log^2 n)$ on the CREW-PRAM with $1 \leq p \leq n$ processors.

**Proof**: From the river routing algorithms described in [CJ1], we know that the number of line segments in $S_1$ is less than $2n$. Moreover, it is obvious that the number of line segments in $S_2$ is less than $2n$. Therefore the total number of line segments involved is $O(n)$. Intersection can be determined by the methods of [MiS] with corresponding time complexity of $O(\frac{n \log n}{p} + \log^2 n)$ with $1 \leq p \leq n$ processors.

Similarly for case 2, define $S_1$ as the set of all the wire segments of the representative nets of the groups and $S_2$ as the set consisting of the four frame edges. Again it can be checked that case 2 intersections can be detected by

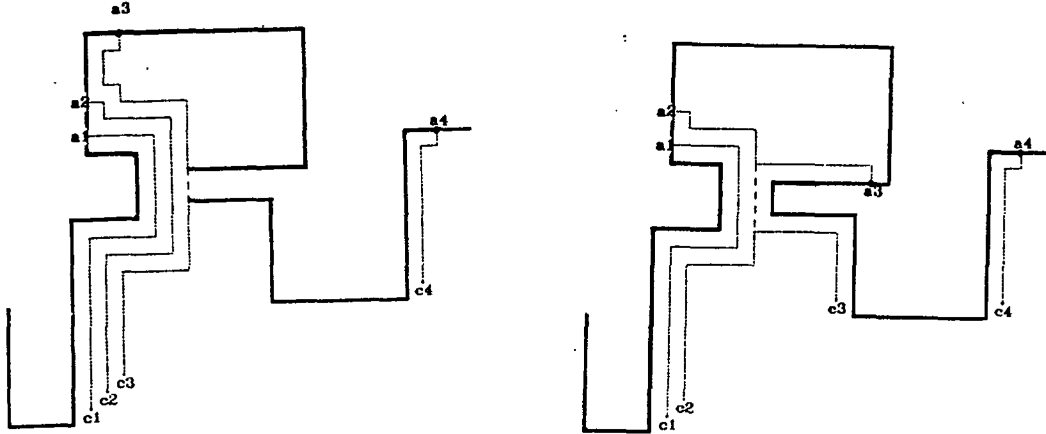



Figure 5.6: Intersections Determined by Test Pairs

determining whether or not $S_1$ and $S_2$ intersect. We leave the third case to the reader.

We now address the problem of whether there is enough area to do the wiring between the module edges. Let $< a_i, b_i >$ and $< a_j, b_j >$ be extreme nets (an extreme net of a group is a net whose module terminal is the first or last when the terminals are sorted in a clockwise direction ) of two groups $G_k$ and $G_l$ such that $a_i$ and $a_j$ are adjacent along the module boundary. This pair of nets is called a *test pair*. Figure 5.6 shows how to use such test pairs.

**Lemma 5.4**: The intersection between the wirings generated by the greedy-in strategy can be tested by examining the intersection of test pairs The total number of line segments involved is $O(n)$ and hence this can be done in $O(\frac{n \log n}{p} + \log^2 n)$ time on the CREW-PRAM with $1 \leq p \leq n$ processors.

The intersection between a greedy-in wiring and a module segment can be easily determined by the method of ([MiS]). Therefore we have the following.

**Theorem 5.1**: Given an instance of the routability testing problem, we can test whether a solution in $O(\frac{n \log n}{p} + \log^2 n)$ time on a CREW-PRAM model with $p$ processors, where $1 \leq p \leq n$ and $n$ is the length of the input.

## 5.4 Minimizing Wire Length

Suppose that in addition to the input $< \mathcal{M}, \mathcal{F}, \mathcal{N} >$, a wiring of all the nets in $\mathcal{N}$ is also provided. Our problem is to modify the given wiring in such a way that the total wire length is minimized. The strategy of deleting *empty U's* outlined in [BaP] minimizes the total wire length. In this section, we will develop a fast and efficient parallel algorithm which minimizes the total wire length.

A *U-Wire* is a sequence of three successive segments resulting from two successive 90 degree turn clockwise or counterclockwise. A U-wire is *reducible* (empty in the terminology of [BaP]) if the line segment one unit from the base is not occupied by another wire or module edge, or is occupied by the base of a reducible U-wire. It is shown in [BaP] that a routing with no reducible U's achieves the minimum total wire length. However their algorithm is inherently sequential.

It is clear that shapes more complicated than reducible U's have to be considered if a fast parallel algorithm is desired. We will assign *types* to each segment of the given module and wiring as follows. Trace the module boundary clockwise starting from an arbitrary point. Each horizontal segment traversed from left to right is of *type 1*, otherwise it is of *type 2*. A vertical segment is of *type 1* if it is traversed top down; otherwise, it is of *type 2*. We now extend this classification to each segment of the wiring. If we traverse a wire from its

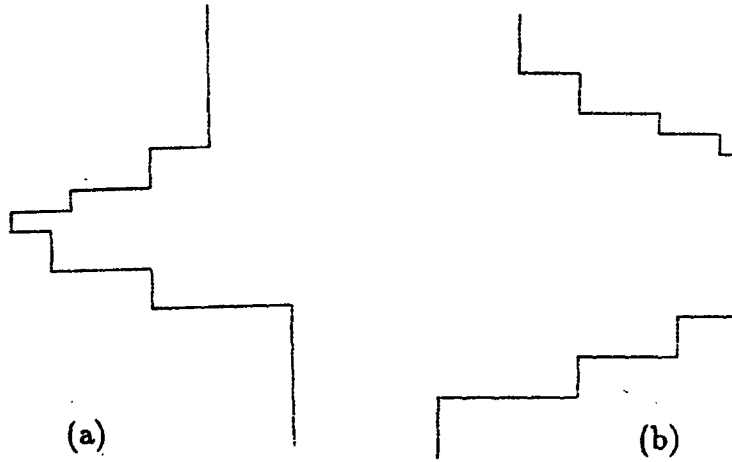



Figure 5.7: Left and Right Wells

module terminal to its frame pad, then a horizontal segment is of *type 1* if it is traversed from left to right, otherwise it is of *type 2*. We can similarly extend the definition to vertical segments. A *horizontal well* is a maximal consecutive sequence of wire segments $e_1, e_2, ..., e_k$, such that $e_1$ and $e_k$ are horizontal with nonempty vertical projections, and $e_1, e_3, ..., e_{2t+1}$ are of one type (for some $t$) and the rest of the horizontal segments are of the other type. Notice that we can have *left* horizontal wells (Figure 5.7(a)) and *right* horizontal wells (Figure 5.7(b)). In a similar fashion, we can define *vertical wells.* Given a net $N$ with a well $W$, an *obstacle* of $W$ is a set $S$ of consecutive module segments or wire segments of a net of *different type* such that $S$ lies inside $W$. For example, in Figure 5.1 net $N_{17}$ has a vertical well with a set of module segments as an obstacle while net $N_{11}$ has a horizontal well with a set of wire segments as an obstacle.

We can shrink wells whenever possible as follows. Let $W$ be a horizontal well with initial segment $e_1 = (B_1, A_1)$ and last segment $e_2 = (B_2, A_2)$ with (say) the x-coordinate of $B_1$ less than or equal the x-coordinate of $B_2$. In addition,

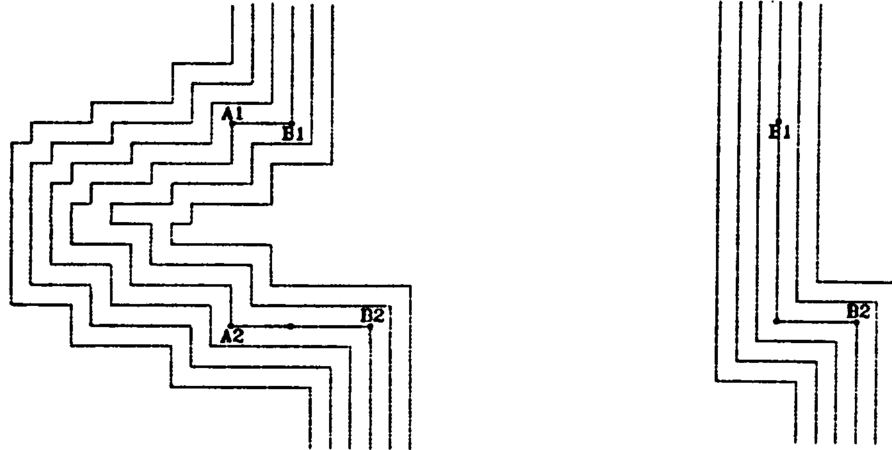


Figure 5.8: Transformation of Wells with no Obstacles

suppose there is no obstacle inside $W$. Then we can apply the transformation shown in Figure 5.8 to shorten $W$. If $W$ has an obstacle inside it, then we find a maximal set of wells with the same obstacle and apply the transformation shown in Figure 5.9 . We now show the following.

**Lemma 5.5**: Suppose there are $k$ reducible wells for a given wiring. Then after applying the transformations described above, the number of reducible wells will be $\leq \frac{k}{2}$.

**Proof**: At most one well will be created between two previous wells after the reduction step.

It follows from the above lemma that if $n$ is the number of wire segments given as input then after $\log n$ iterations of the above transformations, no reducible wells will remain and therefore the resulting wiring is as short as possible.

**Theorem 5.2**: Given an initial wiring, we can change this wiring so that the resulting wiring is of minimum total length in time $O(\frac{n \log n}{p} + \log^2 n)$ with $p$

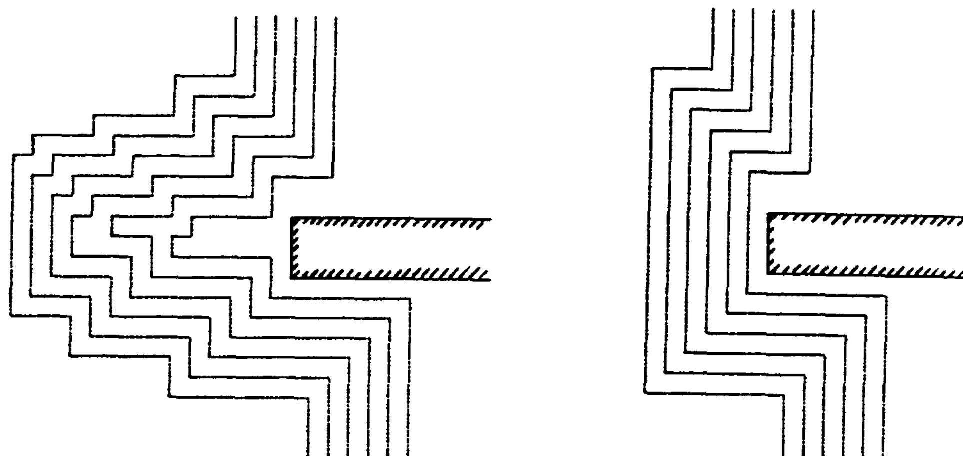


Figure 5.9: Transformation of Wells with an Obstacle

processors on the CREW-PRAM model, where $1 \leq p \leq n$ and $n$ is the length of the input.

**Proof**: The algorithm consists of identifying horizontal and vertical wells and reducing them whenever possible. This process has to be repeated $O(\log n)$ times as implied by the previous lemma. Since there will be no reducible U-wires, the resulting wiring is of minimum length. What remains to be shown is that each iteration can be implemented in $O(\frac{n}{p} + \log n)$ time with $p$ processors, where $1 \leq p \leq n$ and $n$ is the input length. One can verify that determining the types of the segments takes $O(\frac{n}{p} + \log n)$ parallel time by using essentially sorting and that identifying the wells can be done in $O(\frac{n}{p} + \log n)$ parallel time by using path doubling and few other constant time operations. Once the wells are identified, applying the above transformations can be done with few simple operations in $O(1)$ time.

CHAPTER
SIX

# CHANNEL ROUTING IN THE KNOCK-KNEE MODEL

## 6.1 Introduction

In this chapter, we consider the channel routing problem of two-terminal nets in the knock-knee model. A routing algorithm that uses $d$ tracks, where $d$ is the density, is presented in ([PrL]) such that the routing can be realized with three layers. This algorithm can viewed as a nontrivial extension of the left edge algorithm ([Oht]) in which the routing is done row by row, left to right according to a greedy strategy. However, this method seems to be *inherently sequential* even for the case when each column has at most one terminal. We develop a novel strategy to obtain the optimal routing (which is in general different from the one obtained by the [PrL] method) such that both the routing and the layer assignment algorithms have linear time sequential implementations. Moreover, they are both fully parallelizable in the sense that they can be implemented on the CREW-PRAM model in $O(\log n)$ time with $O(n)$ processors, where $n$ is the number of nets. If all the terminals lie in the range $[1, N]$, where $N = O(n)$, then these algorithms will run in time $O(\frac{n}{p} + \log n)$ time with $p$ processors, $1 \leq p \leq n$, on the CREW-PRAM model.

The rest of this chapter is organized as follows. The basic definitions needed for the rest of the chapter are introduced in the next section, while in section 6.3 we develop a novel routing strategy and establish its correctness. The layer assignment algorithm is presented in the last section.

## 6.2 Definitions

We assume that the reader is familiar with the basic definitions related to channel routing (See for example [Oht],[PrL]). In this chapter, we restrict ourselves to two-terminal nets $N = < t, b >$, where $t$ is the *top* terminal (on the top row) and $b$ is the *bottom* terminal. $t$ and $b$ will also represent the integer displacements of these terminals relative to a fixed origin. $N$ is a *left* (*right*) net if $t < b$ ($t > b$). Otherwise it is a *vertical* net. We will also represent a net $N$ as $N = [l, r]$, where $l \leq r$, $l = min\{t, b\}$ and $r = max\{t, b\}$. We refer to $l$ and $r$ as the *left* and *right* terminals of $N$ respectively. An instance of the channel routing problem (CRP) is a channel consisting of a rectangular grid and a set of nets whose terminals lie on the grid points of the (horizontal) parallel boundaries. The *local density* $d_x$ at $x$ is defined to be the number of nets $[l_i, r_i]$ such that $l_i \leq x < r_i$. The *density* $d$ is given by $d = max_x\{d_x\}$. A routing in the knock-knee model consists of a set of edge-disjoint paths (made up of gridline segments) connecting the terminals of each net. Hence a shared grid point could be one of two types: crossing and knock-knee (Figure 6.1).

Let $L_1, L_2, \ldots, L_t$ be a set of conduction layers stacked on top of each other such that $L_1$ is on the bottom and $L_t$ is on the top. A *wiring layout* is an assignment of single layer to each routing segment such that (1) no two segments of two distinct nets share a grid point on the same layer, (2) a routing path may change layers at a via and (3) no wire can use a grid point on a layer

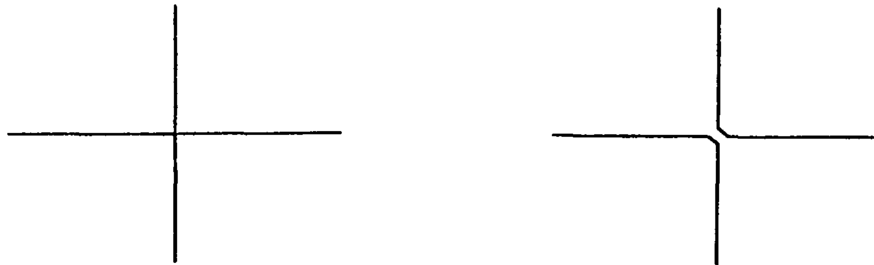


Figure 6.1: Types of shared grid points

which is between two layers with a via at that grid point. It is known that any routing in the knock-knee model can be realized with four layers ([BrB]) and that three layers suffice for the channel routing problem ([PrL]).

Given a routing of an instance of CRP, the *diagonal diagram* can be obtained by inserting a diagonal for each knock-knee, a half-diagonal for each bend. If we remove the half-diagonals, we obtain the *core* layout. It is known that a wire layout can be realized with three layers if its core can [PrL]. A *partition grid* is a grid containing all the diagonals (see [PrL] for a formal definition). A set $P$ of edges of the partition grid is called a *legal partition* if the following properties hold:

1. Every internal vertex in incident on an even number of edges of $P$.

2. The set of diagonals in $P$ is identical to that of the diagonal diagram.

3. None of the *forbidden patterns* in Figure 6.2 appear in $P$.

A legal partition of a core layout $W$ exists if and only if $W$ can be wired with three conducting layers.

We use the standard CREW (Concurrent Read Exclusive Write) shared memory model. All our results will be stated in this model. However, our

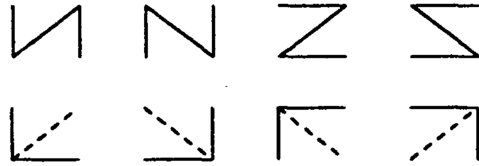


Figure 6.2: Forbidden Patterns

algorithms have fast implementations on fixed-interconnection networks such as the mesh or the hypercube. For example, all the algorithms stated in this chapter can be implemented on a $\sqrt{n} \times \sqrt{n}$ mesh in time $O(\sqrt{n})$, where $n$ is the input length.

## 6.3 Channel Routing

Given an instance of CRP of density $d$, our goal is to determine a wiring of all the nets in $d$ tracks. In addition, the resulting layout or a slight modification of it should be realizable in three layers.

The algorithm developed in [PrL] constructs the wiring track by track by lying each track from left to right. The overall strategy can be viewed as a nontrivial extension of the *line packing* (or *left edge*) algorithm, where a mechanism is provided to solve conflicts arising in columns. This approach seems to be inherently sequential even if there is at most one terminal in each column. Our method is quite different and consists of two main steps:

1. Partition the nets into $d$ chains satisfying certain properties to be outlined below. In particular, the nets in each chain define a set of nonoveralpping intervals.

2. Assign a track number to each chain. Then wire all the nets simultaneously.

We will outline how to perform each step next. The algorithm below creates chains of nets which will be modified later to satisfy all the desired properties. We will denote the successor (predecessor) of a net $N$ by $succ(N)$ ($pred(N)$).

*Algorithm Create Chains*

*Input*: terminals $l_i$'s and $r_i$'s of all the nets $N_1, N_2, \ldots, N_n$.

*Output*: $d$ chains of nets, where $d$ is the density of the corresponding channel routing problem.

1. Mark all terminals as *active*. For each left terminal $l_i$ of a net $N_i$, find the nearest right terminal $r_j$ of some other net such that $r_j$ is to the left (or in the same column) of $l_i$. If two such choices are possible, pick the one whose corresponding net is of the same type as $N_i$. Set $p(l_i) = r_j$. If no such $r_j$ exists, then set $p(l_i)$ =nil. Similarly, define $p(r_i)$ for each right terminal.

2. If $p(l_i) = r_j$ and $p(r_j) = l_i$, then set $succ(N_j) = N_i$, and mark $r_j$ and $l_i$ as *inactive*. Create a reference point $k$ between $r_j$ and $l_i$.

3. Let $R_1, R_2, \ldots, R_m$ be the intervals determined by the reference points. For each $R_i$, create $L(R_i)$ consisting of all the active left terminals, and $R(R_i)$ consisting of all the active right terminals in $R_i$.

4. Find the corresponding terminal pairs in $R(R_i)$ and $L(R_{i+1})$ and create links as before. Mark all terminals used as inactive and merge intervals $R_{2i-1}$ and $R_{2i}$ for all $i$. Repeat this step until there is one interval left.

As an example, consider the channel routing instance of Figure 6.3. The chains produced by the above algorithm are given in Figure 6.4. We also have

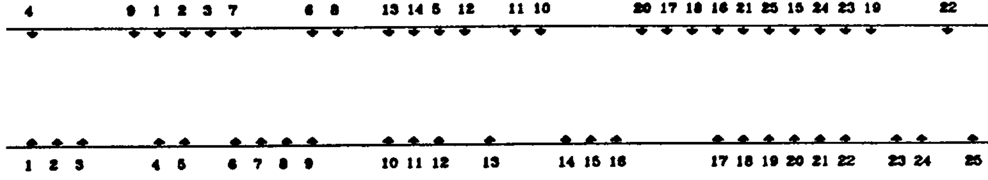



Figure 6.3: A channel routing problem

1. $N_1 \rightarrow N_{14} \rightarrow N_{15} \rightarrow N_{23}$
2. $N_4 \rightarrow N_7 \rightarrow N_8 \rightarrow N_{10} \rightarrow N_{16} \rightarrow N_{25}$
3. $N_2 \rightarrow N_5 \rightarrow N_{12} \rightarrow N_{18} \rightarrow N_{21} \rightarrow N_{24}$
4. $N_3 \rightarrow N_6 \rightarrow N_{13} \rightarrow N_{17} \rightarrow N_{19}$
5. $N_9 \rightarrow N_{11} \rightarrow N_{20} \rightarrow N_{22}$

Figure 6.4: The chains created by Algorithm Create Chains

the following.

**Lemma 6.1**: The number of chains created by the above algorithm is exactly $d$, where $d$ is the channel density. This algorithm can be implemented on the CREW-PRAM in time $O(\frac{n}{p} + \log n)$ with $1 \leq p \leq n$ processors, where $n$ is the number of nets.

**Proof** : Let $R_1, R_2, \ldots, R_m$ be the intervals created by the above algorithm, prior to a set of merging operations of step 4, such that $K_i$ is the reference point between $R_{i-1}$ and $R_i$. Let $n_{r_i}$, $n_{l_i}$ be respectively the numbers of active right and left terminals in $R_i$ and let $n_{k_i}$ be the number of nets with terminals on different sides of $K_i$.

**Claim**: The following inequalities hold true before each set of merging opera-

tions performed in step 4 of the above algorithm:

$$n_{r_i} + n_{k_{i+1}} \leq d$$

$$n_{l_i} + n_{k_i} \leq d$$

*Proof of Claim*: Notice that initially all active right terminals in $R_i$ must be to the right of the rightmost left terminal $l_j$ in $R_i$. If at the completion of step 3, $n_{r_i} + n_{k_{i+1}} > d$, then the density of the channel at a point between the right and left terminals of $R_i$ is $\geq n_{r_i} + n_{k_{i+1}} > d$, which is impossible. Similarly we can establish the other inequality. We now show that after each set of merging operations, the inequalities will hold. Consider the merging of the intervals $R_{2i-1}$ and $R_{2i}$. We know that $n_{r_{2i-1}} + n_{k_{2i}} \leq d$ and $n_{l_{2i}} + n_{k_{2i}} \leq d$. Let $c = min\{n_{l_{2i-1}}, n_{r_{2i-2}}\}$. We distinguish between two cases:

1. Suppose that $n_{l_{2i}} \geq n_{r_{2i-1}}$. Then the number of left terminals in the new merged interval $R_{i'}$ is given by $n_{l_{i'}} = n_{l_{2i-1}} + n_{l_{2i}} - n_{r_{2i-1}} - c$ and hence $n_{l_{i'}} + n_{k_{i'}} = n_{l_{2i-1}} + n_{l_{2i}} - n_{r_{2i-1}} + n_{k_{2i-1}} - c$. But $n_{l_{2i-1}} + n_{k_{2i-1}} = n_{r_{2i-1}} + n_{k_{2i}}$ and therefore $n_{l_{i'}} + n_{k_{i'}} = n_{l_{2i}} + n_{k_{2i}} - c \leq d$.

2. Suppose that $n_{l_{2i}} < n_{r_{2i-1}}$. Then the number of left terminals in the merged interval $R_{i'}$ will be $n_{l_{i'}} = n_{l_{2i-1}} - c$ and thus $n_{l_{i'}} + n_{k_{i'}} = n_{l_{2i-1}} + n_{k_{2i-1}} - c \leq d$.

In a similar fashion, we can establish the other inequality. This concludes the proof of the claim.

Let $d'$ be the number of chains created by the above algorithm. Clearly, $d' \geq d$. At the termination of the algorithm, the number of chains is equal to the number of left terminals. Using the claim above, we deduce that $d' \leq d$ and hence $d' = d$.

We now establish the time and processor bounds. One can check that a couple of sorting steps and few simple operations will take care of step 1-3. Step 4 consists of $O(\log n)$ merging operations each of which can be done in $O(1)$ time.

The above chains can be used to wire all the nets in $d$ tracks. However, the corresponding layout may not be realizable in three layers. We modify the above chains so that they have the following property. Let $c$ be any column. Then either

1. $c$ is empty, or
2. $c$ contains one terminal, or
3. $c$ contains two terminals of nets $N_i$ and $N_j$. Let $N_i = < c, b_i >$ and $N_j = < t_j, c >$.
   - If both $N_i$ and $N_j$ are either right or left nets, then they both belong to the same chain and one is the successor of the other.
   - Suppose that $N_i$ is a right net and $N_j$ is a left net. The other case can be dealt with similarly. Let $N_i' = succ(N_i)$ and $N_j' = succ(N_j)$. Then they either share a column or the column of $N_i'$ or $N_j'$ which is closer to $c$ has only one terminal (see Figure 6.5(b)).

The following algorithm outlines how to modify the chains so that the above property holds.

*Algorithm Modify Chains*

*Input:* A set of chains produced by the algorithm create chains.

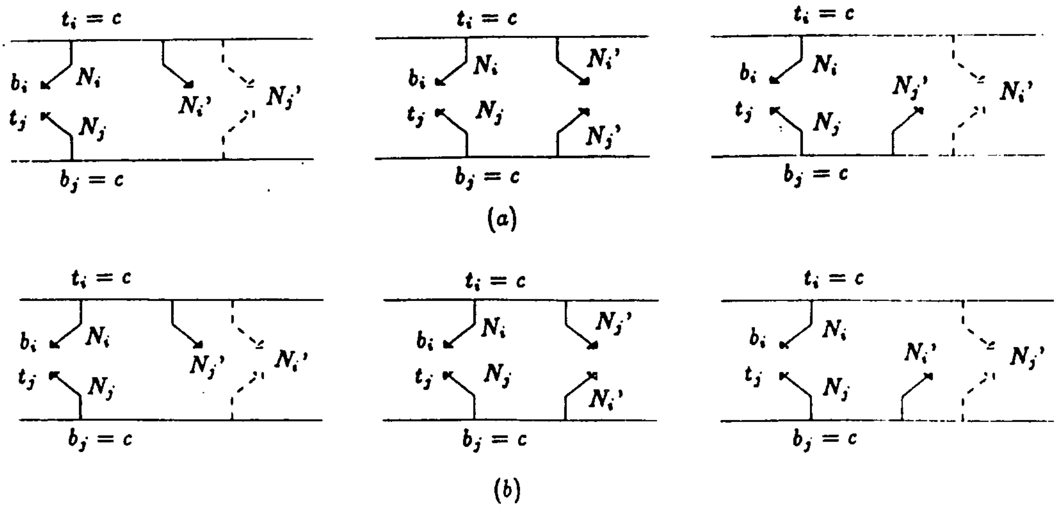


Figure 6.5: Possible successors of two nets with right terminals in the same column

*Output*: A set of chains satisfying the property stated above.

1. Mark each column with two right or two left terminals as active.

2. For each active column $c$ with a top right terminal $t_i$ and a bottom right terminal $b_j$, do the following:

   - If the left terminals of $succ(N_i)$ and $succ(N_j)$ are in the same column $c'$, then mark both $c$ and $c'$ as inactive.

   - If the left terminals are in two distinct columns, say $c'$ containing the left terminal of $succ(N_j)$ is the left one, then mark $c$ inactive if $c'$ has only one terminal.

   - Otherwise, $c'$ contains another left terminal $b'_k$. Let $N_k = pred(N'_k)$. Then create the pair $< N_i, N_k >$. Mark $c$ and $c'$ as inactive.

3. Group the pairs $< N_i, N_k >$ into maximal groups $< N_{k0}, N_{k1} >, < N_{k1}, N_{k2} >, \ldots, < N_{kt-1}, N_{kt} >$. Update the successors of these nets by setting the new

1. $N_1 \rightarrow N_6 \rightarrow N_{10} \rightarrow N_{16} \rightarrow N_{19}$
2. $N_4 \rightarrow N_7 \rightarrow N_8 \rightarrow N_{11} \rightarrow N_{20} \rightarrow N_{23}$
3. $N_2 \rightarrow N_5 \rightarrow N_{12} \rightarrow N_{18} \rightarrow N_{21} \rightarrow N_{24}$
4. $N_3 \rightarrow N_{14} \rightarrow N_{15} \rightarrow N_{22}$
5. $N_9 \rightarrow N_{13} \rightarrow N_{17} \rightarrow N_{25}$

Figure 6.6: New chains generated by Algorithm Modify Chains

successor of $N_{ki}$ to be the previous successor of $N_{ki+1}$ for all $0 \leq i < t-1$. In addition, set the new successor of $N_{kt}$ to be the previous successor of $N_{k0}$.

4. Repeat procedure for active columns with two left terminals.

5. Adjust chains in such a way that whenever the configurations of Figure 6.5(a) occur, they will be replaced by the corresponding configurations of Figure 6.5(b) (similarly for columns with two left terminals).

As an example, consider the chains of Figure 6.4. Then the above algorithm creates the new set of chains given in Figure 6.6.

**Lemma 6.2**: The above algorithm modifies the chains generated by the algorithm Create Chains such that the new chains satisfy the desired properties. Moreover, the algorithm runs in $O(\frac{n}{p} + \log n)$ time with $1 \leq p \leq n$ processors on the CREW-PRAM model.

**Proof**: To simplify the presentation we will introduce a new graph called the *link graph.* There is vertex $v_c$ corresponding to each column $c$. There is an edge between $v_c$ and $v_{c'}$ if and only if c contains a terminal of a net whose successor or predecessor has a terminal in $c'$. Notice that the link graph of each of the groups created in step 3 has the form shown in Figure 6.7(a). If $c_0'$ has another link to $a$, then $a$ cannot appear between $c_0$ and $c_1$. After the modifications

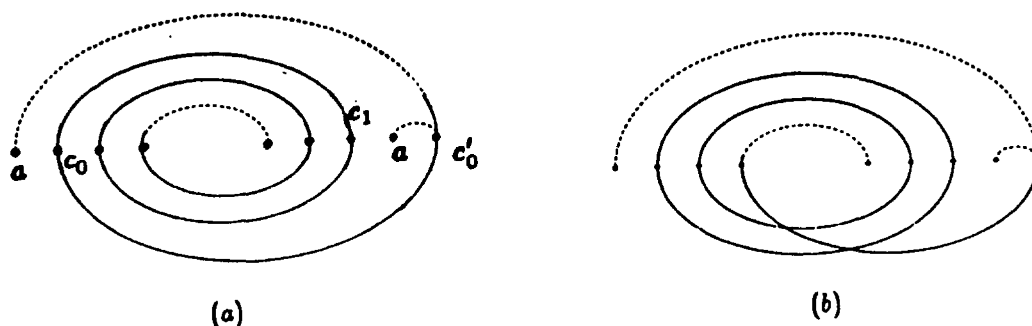



Figure 6.7: Forms of groups in the proof of Lemma 6.2

performed in step 3 the link graph of the group will be of the form given in Figure 6.7(b) with 2 link loops or paths of length 2. Hence it is clear that after step 3 no column with two right terminals could cause any problem. Each group may have generated one column with two left terminals which do not satisfy the desired property. Then step 4 of the above algorithm takes care of all these columns (Figure 6.5). Step 5 insures that columns with two terminals will be of the form given in Figure 6.5(b). The time and processor bounds of the algorithm can be easily established.

The track assignment and the wire layout will be described next. Suppose that track $k$ has been assigned to net $N = < t, b >$. Then the wire of $N$ will consist of the interval $[t_k, b_k]$ on track $k$, a vertical line segment from $b$ to $b_k$, and a vertical line segment from $t$ plus a possible detour to $t_k$. Therefore the problem comes down to determining how to connect a terminal on the upper row down vertically to its track. The algorithm below describes how to achieve this.

*Algorithm Wire Nets*

*Input*: A chain of nets as modified by the algorithm Modify Chains.

*Output*: A wire layout for each net.

1. For each chain, assign the leftmost terminal $l_i$ as the primary key, and, if $l_i$ is a bottom terminal, assign 0 as the secondary key and 1 otherwise. Sort the chains according to their keys. The track number of each chain is its corresponding rank.

2. For each column $c$, do the following:

   1. if $c$ contains one terminal of a net $N$, then connect that terminal vertically to the track of $N$.

   2. Suppose $c$ contains two terminals of a single net. Then connect these two terminals vertically.

   3. Suppose that $c$ contains two terminals of two distinct nets $N =< c, b >$ and $M =< t, c >$. If $N$ and $M$ have the same track number, then wire the terminals to this track using a knock-knee. Otherwise there is detour only if the track number of $N$ is less than that of $M$. In this case, it is a left or right detour depending on whether $c$ is a right or left terminal. The detour extends to either to the column of successor (for a right detour)or predecessor (for a left detour)of either $N$ or $M$ whichever is closer. All the cases that can arise and the corresponding routing are shown in Figure 6.8.

Consider the example of Figure 6.2 again. Then the routing obtained by the above algorithm is given in Figure 6.9 .

**Lemma 6.3**: Given an instance of the channel routing problem, the above algorithm provides a legal routing of all the nets in the knock-knee model.

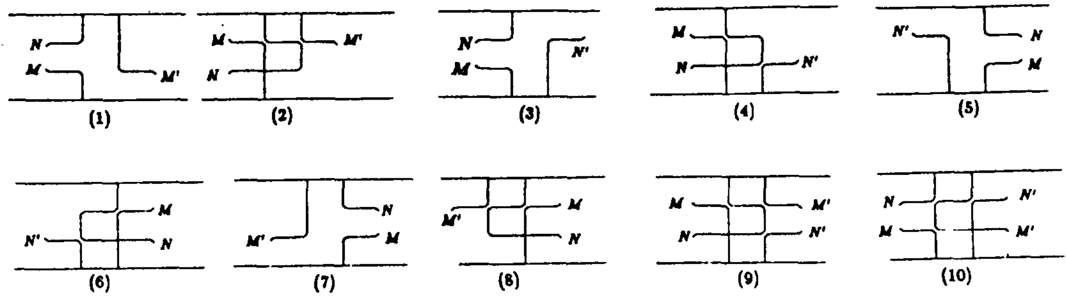



Figure 6.8: Possible detours of nets with terminals in the same column

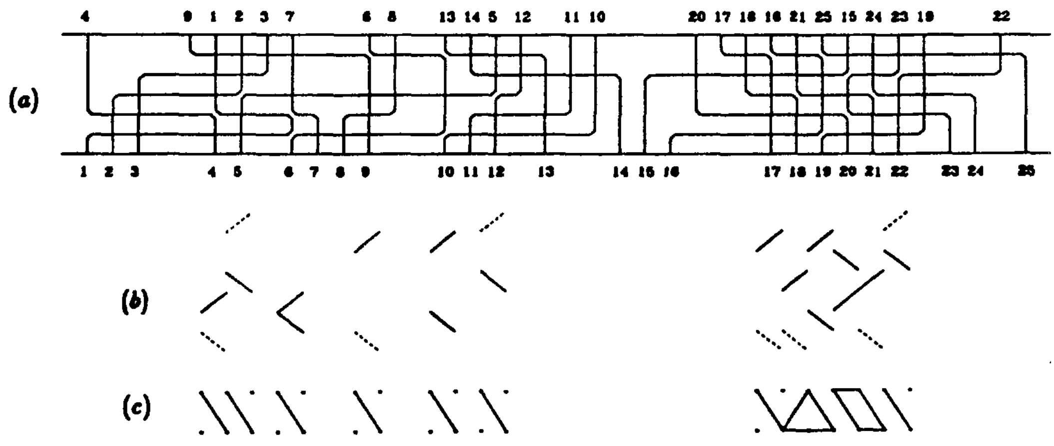



Figure 6.9: (a) The layout generated by Algorithm Wire Nets, (b) its corresponding diagonal diagram and (c) its corresponding constraint graph

**Theorem 6.1**: Given an instance of the channel routing problem of density $d$, it is possible to wire all the nets in $d$ tracks in time $O(\log n)$ time on the CREW-PRAM model with $O(n)$ processors, where $n$ is the number of nets. If all terminals lie in the range $[1, N]$, where $N = O(n)$, then the above algorithm can be implemented in $O(n)$ sequential time and in $O(\frac{n}{p} + \log n)$ parallel time with $p$ processors on the CREW-PRAM model, where $1 \leq p \leq n$.

**Proof**: The first statement of the theorem follows from the previous lemmas. If all the terminals lie in the interval $[1, N]$, $N = O(n)$, then sorting (most expensive step) takes $O(n)$ sequential time. For the parallel implementation, the most expensive steps are sorting and traversing linked lists. Using the results of ([KR1],[CV2]) we obtain the bounds stated in the theorem.

## 6.4 Layer Assignment

In this section, we show that a modified version of the routing produced by the algorithm of the previous section can be laid out in three layers. [PrL] provides a necessary and sufficient conditions for the realization of a wiring in three layers. As stated in section 6.2, the problem is essentially reduced to finding a legal partition of the core of the diagonal diagram. The routing layout produced by the algorithm in [PrL] has a special property, namely every column is either empty or contains one diagonal or a diagonal \ on the bottom and a diagonal / above it. Their algorithm proceeds from left to right, looking at each column and making vertical connections (and possibly changing the routing) so that the resulting partition is legal. Unfortunately, we encounter a major difficulty in our case. Each column of our routing layout could have two diagonals (\ and / ) in an arbitrary order (because our routing uses left and right detours). This makes it necessary to change the wire layout much more

substantially than was done in [PrL]. In the rest of this section, we outline how to overcome this difficulty.

By adding dummy diagonals if necessary, we can assume that each column is either empty or contains exactly two diagonals. As in [PrL], our partition will be constructed by adding vertical edges only. Define a *reference line* as a vertical line that touches the endpoint of some diagonal. For each reference line, the diagonals touching this line will partition it into several line segments. Number these line segments starting from the top most segment. Notice that there are two possible ways of adding vertical segments (to create a legal partition): add the odd-numbered or the even-numbered segments. We have to choose (if possible) those segments that will not create a forbidden pattern.

We define the *constraint graph* as follows. The two possible choices of vertical segments corresponding to reference line $L_i$ are represented by two vertices $v_{2i-1}$ and $v_{2i}$. Two vertices are connected by an edge if and only if the corresponding choices create a forbidden pattern. Notice that forbidden patterns can be created only between adjacent reference lines.

**Lemma 6.4**: The total number of the edges between the vertices corresponding to adjacent reference lines is $\leq 2$.

**Proof**: Since the maximum number of diagonals between two adjacent vertical reference lines is 2, there are at most two "constraints" between $\{v_{2i-1}, v_{2i}\}$ and $\{v_{2i+1}, v_{2i+2}\}$, for each $i$.

Our goal is to pick for each reference line one of its vertices such that no two such vertices are connected by an edge. This may not be possible, in which case the routing layout has to be modified. We introduce the patterns that can create potential problems. A *forbidden column* is a pair of vertices corresponding to a reference line such that no selection of its vertices will lead

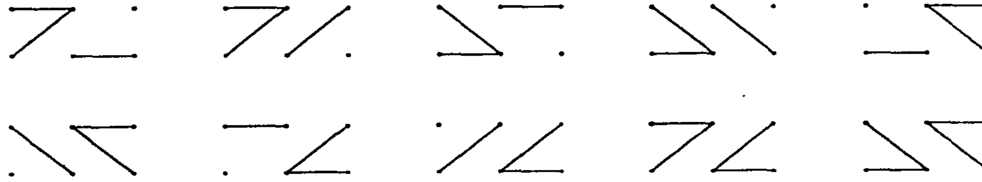


Figure 6.10: Configurations that may give rise to forbidden columns

to a legal partition. The set of configurations that *may* give rise to a forbidden column are shown in Figure 6.10.

Our goal is to modify the wiring layout if necessary so that the resulting constraint graph has no forbidden columns. We start by showing that any such graph will lead to a legal partition. The following algorithm shows how to select the proper set of vertices.

*Algorithm Select*

*Input*: Reference lines and the corresponding constraint graph with no forbidden columns.

*Output*: A subset of the vertices which will induce a legal partition of the wiring layout.

1. Mark all reference lines as *active*. For each reference line $L_i$, select $v_{2i}$ ($v_{2i-1}$) if $v_{2i-1}$ ($v_{2i}$) is incident on two edges to a single adjacent column. If such a selection is made, mark $L_i$ as inactive and assign weight 0 if $v_{2i}$ is selected, otherwise assign weight 1.

2. Create a sorted list for each set of active reference lines between two inactive reference lines.

3. For each list created in step 2, do the following. Assign a weight 0 to each

line $L_k$ in the list if there is an edge between $v_{2k-3}$ and $v_{2k}$ or between $v_{2k-2}$ and $v_{2k-1}$. Otherwise, assign a weight of 1 to $L_k$.

4. Calculate the rank of each reference line. Then select $v_{2k}$ if the rank of $L_k$ is even; otherwise select $v_{2k-1}$.

**Lemma 6.5**: Given a partition graph with no forbidden columns, Algorithm Select will generate a subset of the vertices that determine a legal partition of the wiring layout.

**Proof**: Let's start by observing that the selection made in step 4 for inactive reference lines is consistent with that of step 1 because the graph contains no forbidden columns. For the rest of the proof, it is enough to show that there is a selected vertex for each reference line such that no two selected vertices are connected by an edge. The algorithm clearly selects exactly one vertex for each reference line. Suppose that there is an edge between two selected vertices, say $v_{2k}$ and $v_{k-2}$. Then the weight of $L_k$ must be 0 (because both have even ranks). But then either $v_{2k}$ is connected to $v_{2k-3}$ or $v_{2k-1}$ is connected to $v_{2k-2}$. In the first case, $v_{2k-1}$ would have been selected; in the second case, $v_{2k-3}$ would have been selected. Similarly we can handle the other cases. Notice that the selection made in step 4 for inactive reference lines is consistent with that of step 1 because the graph contains no forbidden columns.

In the rest of this section, we will show how to modify the wiring in such a way that the corresponding constraint graph has no forbidden columns. We first introduce the following classification of reference lines (cf [PrL]): *Trivial* (Figure 6.11), *Overlap* (Figure 6.12), *Disjoint* (Figure 6.13), *Inclusion* (Figure 6.14). Each type is shown with its possible constraint graph. The only possible forbidden columns could come from: $D_1, D_3, D_6, D_8, I_2, I_4, I_6, I_8$. In most of these cases, the wiring has to be modified by adding diagonals in such a way

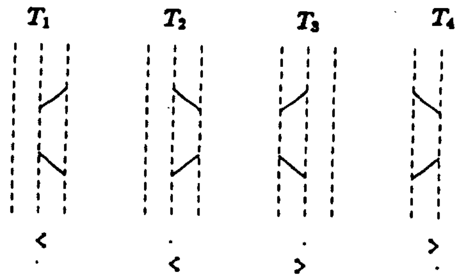



Figure 6.11: Trivial reference lines

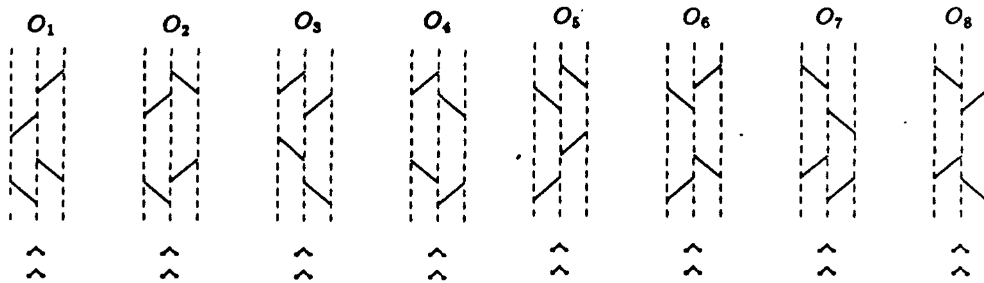



Figure 6.12: Overlap reference lines

that no forbidden column could possibly arise. The procedure involves a detailed case study which is summarized by the following algorithm.

*Algorithm Modify*

*Input*: Wiring layout produced by Algorithm Wire Nets.

*Output*: A new wiring with its modified constraint graph and a set of selected vertices.

1. Generate the diagonal diagram, delete all half diagonals and add necessary dummy diagonals as follows. If there exists exactly one diagonal \, then add

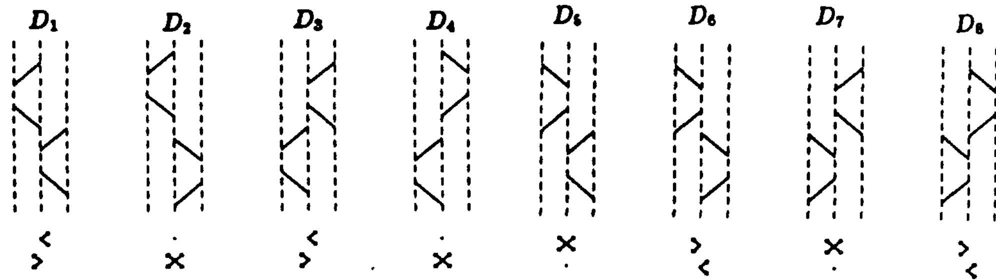



Figure 6.13: Disjoint reference lines

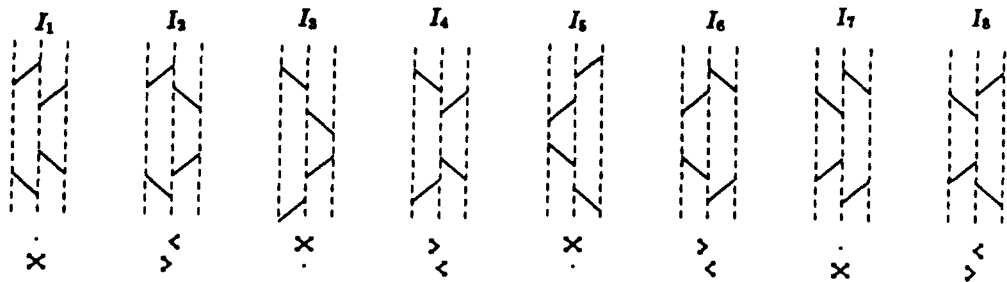



Figure 6.14: Inclusion reference lines

a dummy diagonal / in an additional row above all the rows. If there exists exactly one diagonal /, then add a dummy diagonal \ in an additional row below all the rows. Determine the constraint graph and mark all reference lines which may give rise to forbidden columns as active.

2. Handle type $I_2$ active reference lines as follows. Let $L_j, L_{j-2}, \ldots, L_{j-2k}$ be a maximal chain of active $I_2$'s. We want to modify every other $L_i$ starting with $L_j$ in a way that depends on the type of its left neighbor $L_{i-1}$. All the cases that can arise are shown in Figure 6.15 with the corresponding modifications. In each such case, a vertex of $L_{i-1}$ is selected (its degree is 0), edges between reference line $L_{i-1}$ of selected vertex and its neighbors removed and the reference lines $L_i, L_{i-1}, L_{i-2}$ are marked inactive. Handle type $I_6$ reference lines in a similar fashion.

3. Handle type active $I_4$ as shown in Figure 6.16. Select $v_{2i}$ and remove edges between $L_i$ and its neighbors. Mark $L_i, L_{i-1}, L_{i+1}$ as inactive. Handle type $I_8$ similarly.

4. Handle active type $D_1$ as shown in Figure 6.17. Select $v_{2i-1}$ and remove edges between $L_i$ and its neighbors. Mark $L_{i-1}, L_i, L_{i+1}$ as inactive. In Figure 6.18 a maximal chain of $D_1$'s is considered. $L_i, L_{i+1}, \ldots, L_k$ are all of type $D_1$. If $L_i$ or $L_k$ can give rise to a forbidden column, then modify as shown and remove all edges of $L_i - L_k$. All the odd vertices of of $L_i - L_k$ are selected. As before edges are removed for selected columns and adjacent reference lines are marked inactive. Repeat the same procedure for types $D_3, D_6$ and $D_8$.

**Lemma 6.6**: Algorithm Modify will change the wiring layout produced by Algorithm Wire Nets in such a way that the corresponding constraint graph contains no forbidden columns.

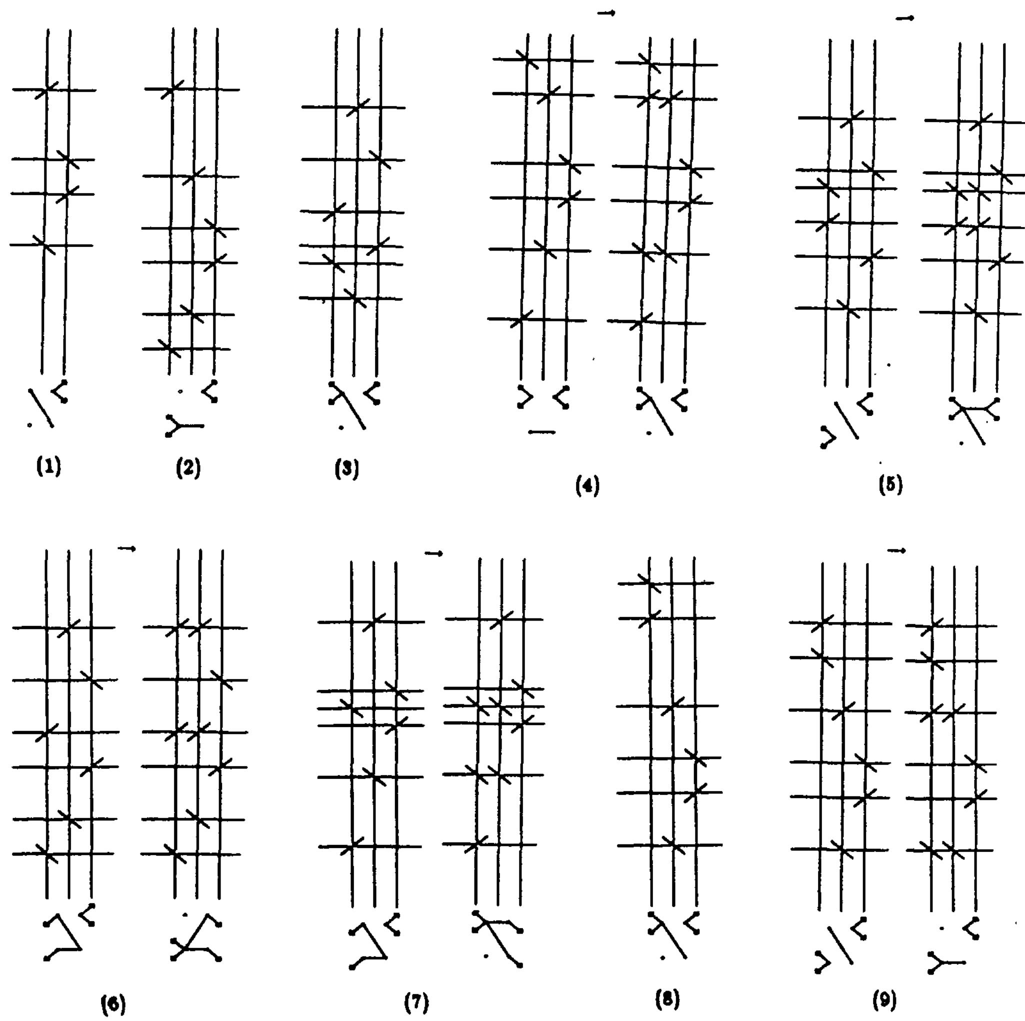



Figure 6.15: Transformations on type $I_2$ reference lines.

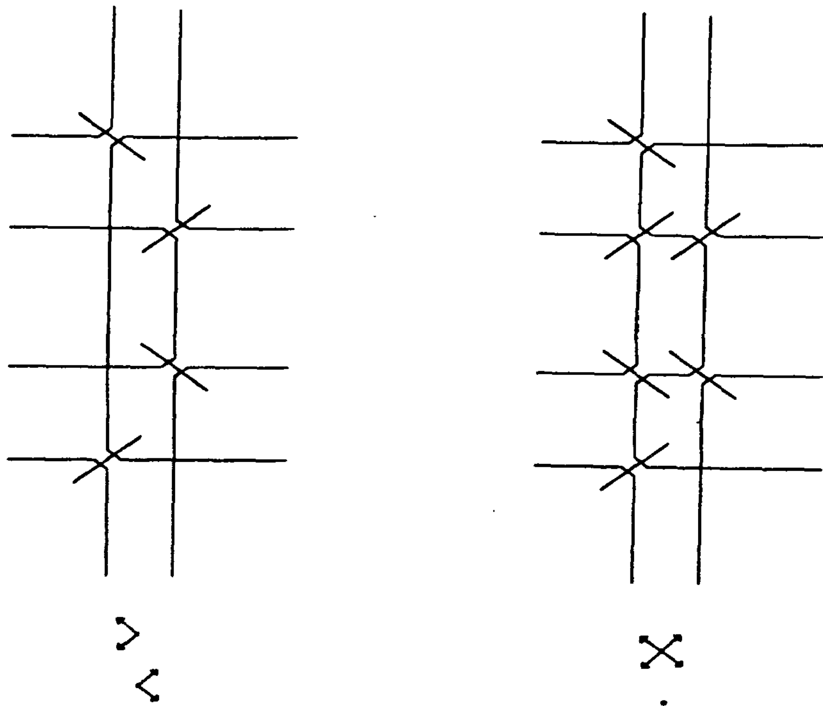


Figure 6.16: Transformations on type $I_4$ reference lines

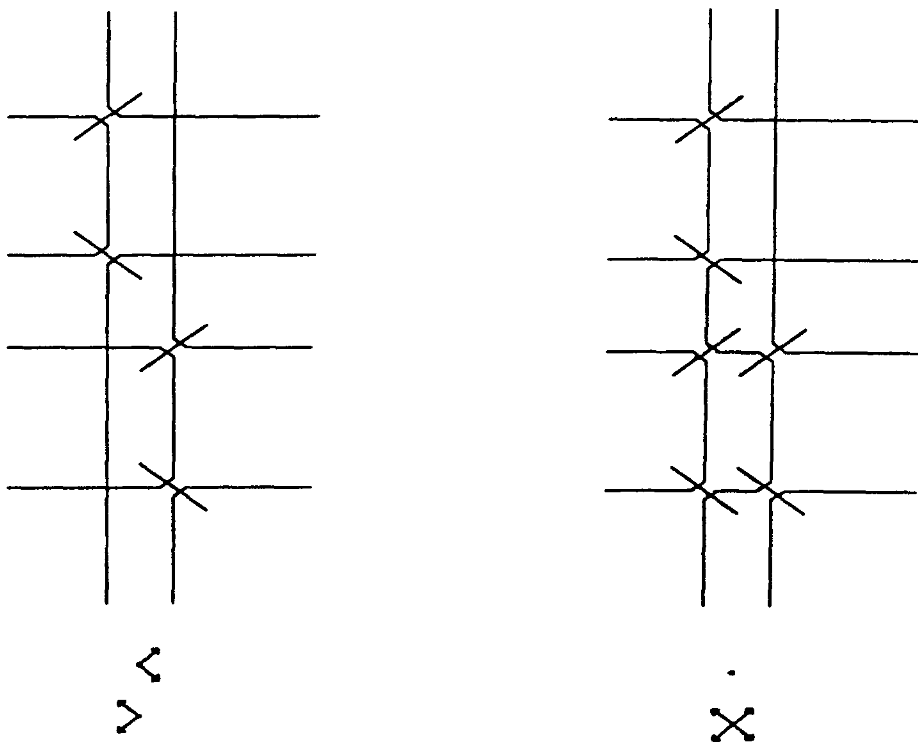


Figure 6.17: Transformations on type $D_1$ reference lines

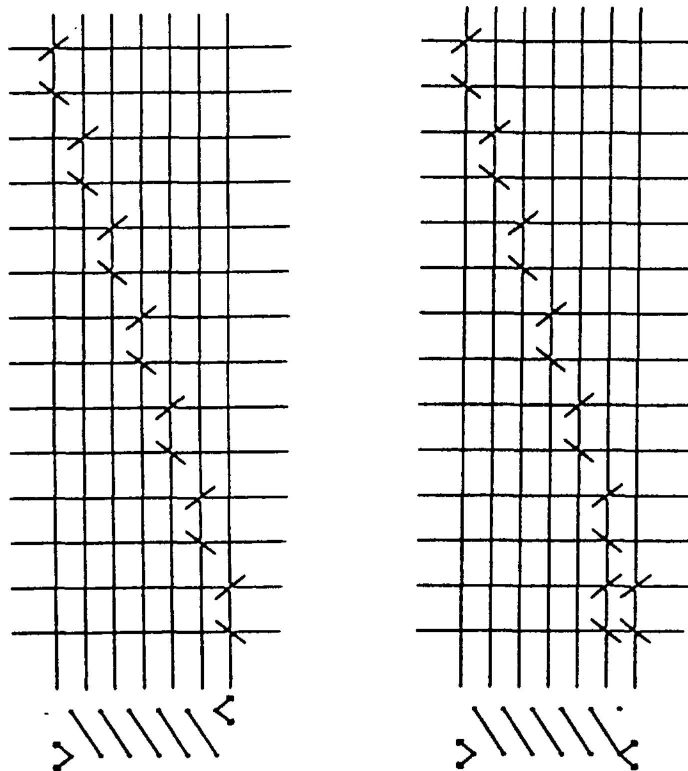


Figure 6.18: Maximal chain of $D_1$'s.

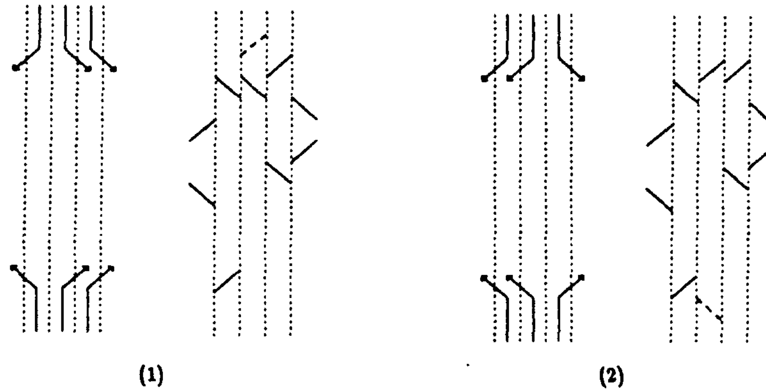



Figure 6.19: Possible wiring configurations for case 2 of lemma 6.6

**Proof**: Consider the original constraint graph in which $L_i$ was of type $I_2$ (hardest case). Then we have to show that $L_{i-3}$ will create no problems. The only nontrivial cases are the following:

1. $L_{i-3}$ is of type $I_2$. In this case the algorithm selects vertices in the columns corresponding to $L_{i-1}$ and $L_{i-4}$ and hence there are no edges left between $L_{i-2}$ and $L_{i-1}$, and between $L_{i-3}$ and $L_{i-4}$.

2. $L_{i-3}$ is of type $I_6$. Suppose that there are no dummy diagonals between $L_{i-3}$ and $L_{i-2}$ or between $L_{i-1}$ and $L_i$. The only possible wiring configurations are shown in Figure 6.19 with their corresponding diagonal diagrams. If there is a dummy diagonal between $L_{i-1}$ and $L_i$, then we can have one of the three possibilities shown in Figure 6.20. In each of these cases, one of $L_i$ or $L_{i-3}$ cannot generate a forbidden column.

3. $L_{i-3}$ is of type $I_4, I_8, D_1, D_3, D_6$ or $D_8$. One can check that none of these cases can possibly generate a forbidden column.

The remaining cases can be dealt with similarly.

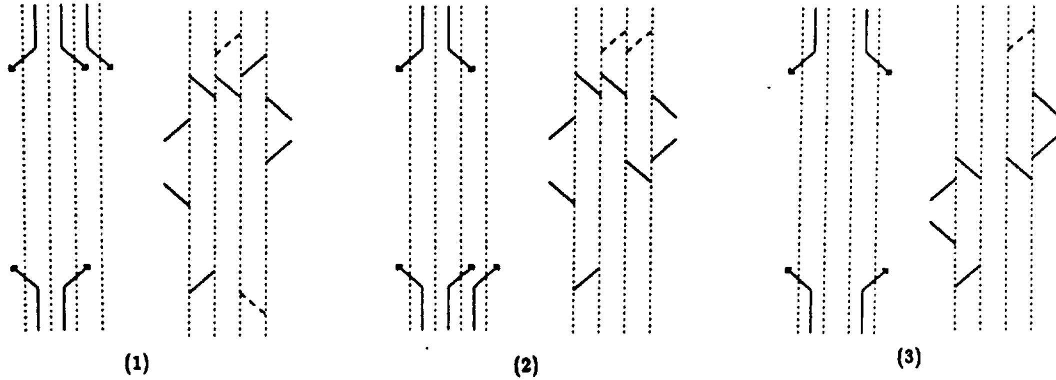



Figure 6.20: Possible configurations with dummy diagonals between $L_i$ and $L_{i-1}$.

If we go back to the example of Figure 6.2, then the routing produced by the algorithm of the previous section is given in Figure 6.9. The layer assignment algorithm will change the wiring of $N_{16}$ and $N_{21}$ (Figure 6.21) and the final layout is shown in Figure 6.22.

**Theorem 6.2**: Given an instance of the channel routing problem, it is possible to determine a three-layer assignment of the routing layout in time $O(\log n)$ time with $O(n)$ processors on the CREW-PRAM model. If all terminals lie in the range $[1, N]$, where $N = O(n)$, then this can be done in $O(n)$ sequential time and in $O(\frac{n}{p} + \log n)$ parallel time with $p$ processors on the CREW-PRAM model, where $1 \leq p \leq n$.

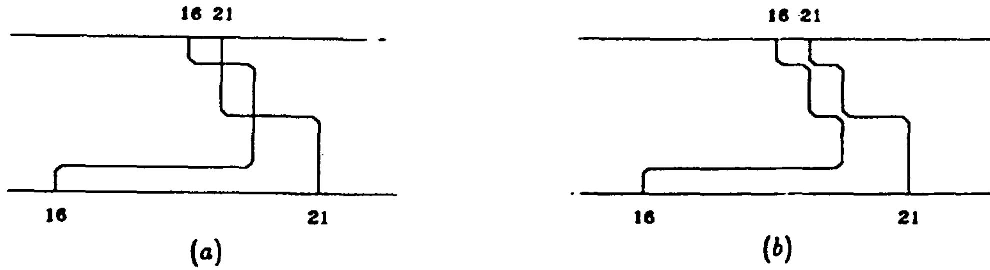



Figure 6.21: Changes in the wiring of $N_{16}$ and $N_{21}$

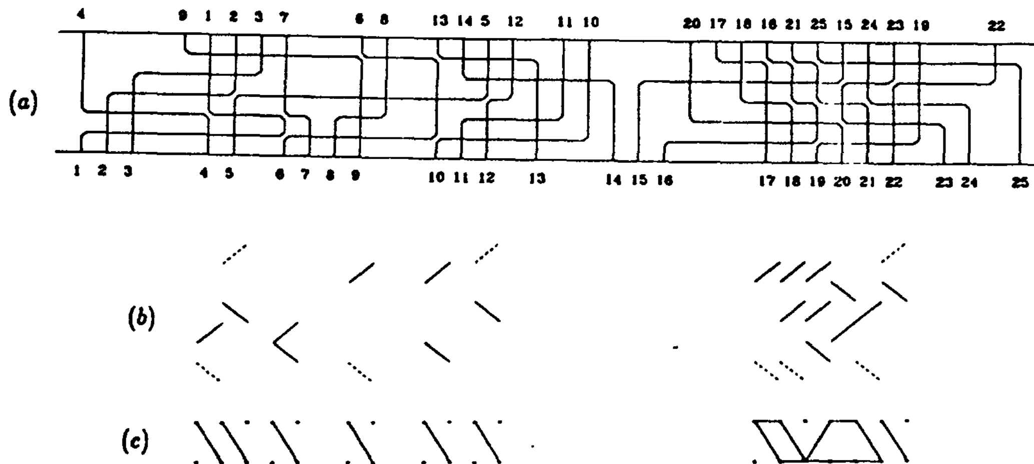



Figure 6.22: (a) The final layout after the modification of layer assignment algorithm, (b) its corresponding diagonal diagram and (c) its corresponding constraint graph

CHAPTER
# SEVEN

# CONCLUSION

## 7.1 Implementation

After describing our parallel algorithms on a theoretical model, we will discuss the implementation of these algorithms on a real parallel computer. We have implemented a number of our algorithms on a CM-2 Connection Machine, which has 16,384 processors, with $64K$ bits memory for each processor, and uses Symbolic 3600 as the front-end computer. The program is coded by using *Lisp parallel programming language. In section 7.1.1, we will briefly introduce the architecture of the Connection Machine. Section 7.1.2 will discuss the *Lisp parallel programming language, while the section 7.1.3 will analyze the performance of our results.

### 7.1.1 The Connection Machine System

The Connection Machine is a fine-grained, highly parallel, SIMD computer developed by Thinking Machines Corporation ([Hil]). Current Connection Machine Systems have either 16,384 or 65,536 processors each with 4,096 bits (in model CM-1) or 65,536 bits (in model CM-2) of memory. The system consists

of two parts: (1) the front-end computer and (2) the array of Connection Machine processor. The front-end computer is a usual Von Neumann computer. A typical front-end processor is a VAX or a Symbolic 3600. The key component of the Connection Machine is a custom designed VLSI chip that consists of 16 processor cells and one router unit of the packet switch communications network. The router is responsible for routing messages between chips. Each router handles messages for 16 processing cells. For the Connection Machine with $65,535$ processors, the connection network is formed by $4,096$ routers connected by a hypercube structure. Assume that the 4096 routers have address 0 through 4095, then the router with address $i$ will be connected to the router with address $j$ if and only if $|i-j| = 2^k$ for some integer $k$. i.e. each router is connected to other 12 routers in the network. Because any two 12-bit addresses differ by no more than 12 bits, each router is no more than 12 wires away from a neighboring router. There is also a second gridlike communications system provided on the chip for local or highly structured communications patterns. This communication system does not involve the router. Instead each processor communicates directly with its North, East, West and South neighbors. On chip, the processors are connected in a $4 \times 4$ grid. This two-dimensional grid can be extended across multiple chips. Hence, the Connection Machine processors are connected by two networks : (1) a packet-switched network based on a hypercube topology and use an adaptive routing algorithm, and (2) a two-dimensional grid-like communication system, each processor communicate directly with its North, East, West and South neighbors.

The processor array is also connected to the memory bus of the front-end so that the local processor memory can be random accessed directly by the front-end just as any general memory. The front-end computer stores data structures on the Connection Machine in the same way as the conventional machine stores them in a memory. Unlike a conventional memory, in the Connection Ma-

chine, the memory cells themselves do the processing. The computation takes place through the coordinated interaction of the cells in the data structure. Because thousands or even millions of processing cells work on the problem simultaneously, the computation proceeds much more rapidly than a conventional machine. In addition to the processing capability of each cell and the general interconnections network that can connect all the cells in an arbitrary pattern, there is a high-bandwidth input/output channel that can transfer data between the Connection Machine and peripheral devices at a much higher rate.

The control structure of a program running on a Connection Machine system is executed by the front-end in the usual way. An important practical benefit of this approach is that the program is developed and executed within the already familiar programming environment of the front-end. The program can perform computations in the usual serial way on the front-end and also issue commands to the processor array. The processor array executes instructions from a single stream generated by a microcontroller under the direction of front-end. The 64K cells CM-1 machine, including the microcontroller, processor/memory cells and communication network is packaged in a cube roughly $1.3m$ on a side.

### 7.1.2 Programming Model and *Lisp

The Connection Machine programming model is carefully abstracted from the details of the hardware that supports it, and, in particular, the number and size of its hardware processors. Programs are described in term of virtual processors. In actual implementations, hardware processors are multiplexed as necessary to support this abstraction; indeed, the abstraction is supported at a very low level by a microcontroller interposed between the front-end and the processor array, so that the front-end always deals with virtual processors.

Since most members of the artificial intelligence community, for whom the Connection Machine was originally designed, are already familiar with Lisp and Lisp is extensible, has dynamic storage allocation and is generally good for symbol manipulation, Lisp was chosen as a base for developing a Connection Machine Language.

*Lisp, which is an extension of Common Lisp, is a data parallel language designed to program the Connection Machine ([HiS],[TMC]). In *Lisp, parallelism is achieved through simultaneous operations over composite data structures rather than through concurrent control structures. In this sense, *Lisp is a relative conservative parallel language, because it retains the program flow and control constructs of a normal serial Lisp but allows operations to be performed simultaneously across each element of a large data structure. Most of the idea in the language are actually relatively independent of Lisp and would be equally applicable to the extension of other languages as Algol, C or even Fortran.

### 7.1.3 Discussion

The algorithms for river routing between rectangles and routing within a rectilinear polygon are currently being implemented by Peter Su as his Master thesis research ([Sup]). We have implemented the channel routing algorithms of section 6.3 with about 2,500 lines of *Lisp program. Figure 7.1 is a program segment which is a small part of *CreateChains* algorithm. To test the correctness and to get the performance information of this program, we have developed a random channel routing problem generating program to generate all the testing problems randomly. Figure 7.2 shows some final layouts obtained by our routing program. The running time of this program for different problem sizes is shown in Table 7.1. Figure 7.3 shows the variation of running time vs. net numbers. As predicted by our algorithm analysis, the running time is a

| Problem | Net Number | Running Time (sec) |
|---|---|---|
| (1) | 26 | 12.408493 |
| (2) | 32 | 14.259800 |
| (3) | 36 | 15.765457 |
| (4) | 43 | 15.663628 |
| (5) | 47 | 21.080296 |
| (6) | 53 | 19.494186 |
| (7) | 57 | 19.395470 |
| (8) | 72 | 26.428810 |
| (9) | 86 | 25.018417 |
| (10) | 94 | 26.637131 |
| (11) | 101 | 25.038742 |
| (12) | 201 | 29.910515 |
| (13) | 301 | 34.450699 |
| (14) | 401 | 36.039131 |
| (15) | 501 | 37.775513 |
| (16) | 1000 | 43.390503 |
| (17) | 1500 | 47.214020 |
| (18) | 2000 | 50.049774 |
| (19) | 2500 | 53.800629 |
| (20) | 3000 | 55.713371 |
| (21) | 3500 | 56.797470 |
| (22) | 4000 | 56.754066 |

Table 7.1 The running time of Channel Routing program for different problem sizes

logrithmic function of the net number.

## 7.2 Concluding Remarks

The routing phase is a critical and time consuming part of the overall VLSI design process. In this thesis, we have investigated several VLSI routing problems. We have developed new approaches to solve these problems and have obtained fast and efficient parallel algorithms for them. Among the most important contributions is that we have introduced new ideas for developing parallel CAD tools. In addition to solving the VLSI routing problems discussed in this thesis, our new parallel techniques can also be used to solve other routing

```
;; For each PE-i with "reference=-1 (1)", adjust llink and rlink
;; such that llink point to the "right margin" of previous reference
;; point ("left margin" of the same reference point) and rlink points
;; to the "right margin" of the same reference point ("left margin"
;; of next reference point).
(*when (and!! (=!! reference (!! 1)) (/=!! llink (!! -1)))
   (*set llink (pref!! rlink llink))
)
(*when (and!! (=!! reference (!! -1)) (/=!! rlink (!! -1)))
   (*if (=!! llink (!! -1))
     (*pset :default rank llink (pref!! llink rlink))
   )
   (*set rlink (pref!! llink rlink))
)
(*when (and!! (=!! reference (!! 1)) (/=!! llink (!! -1))
              (=!! rlink (!! -1)))
   (*pset :default rank rlink (pref!! rlink (pref!! llink llink)))
)


(*when (and!! (=!! reference (!! 1))
              (=!! llink rlink (!! -1)))
  (*set llink (pref!! rlink (!! 0)))
  (*pset :default (1+!! rank) llink (!! 0))
)
(*when (=!! reference rlink llink (!! -1))
  (*set rlink (pref!! llink (pref!! llink (!! 0))))
  (*pset :default (!! -1) llink (!! 0))
)
)

(*when (and!! (>=!! (self-address!!) (!! first))
              (<=!! (self-address!!) (!! last)))

  ;; O(log n) iterations of merging operation (each can be done
  ;; in O(1) time). Finally there is only interval left.
  (*let ((index (!! 0)))
    (declare (type (pvar (signed-byte 16)) index))
    (*if (=!! reference (!! 1))
      (*set index (!! 1))
      (*set index (!! 0))
    )
    (rankprefix first last index -1)
    (setq j (ceiling (log (1+ (pref index last)) 2)))
    ;; j is the total number of iterations needed.

    (do ((i 0 (1+ i)))
        ((equal i j))
      (*let ((tempwrank wrank))
        (declare (type (pvar (signed-byte 16)) tempwrank))

        ;; Only those PEs which contain terminals in even intervals
        ;; are selected.
        (*when (and!! (=!! (mod!! index (!! 2)) (!! 0))
```

Figure 7.1 A program segment of *Create Chains*

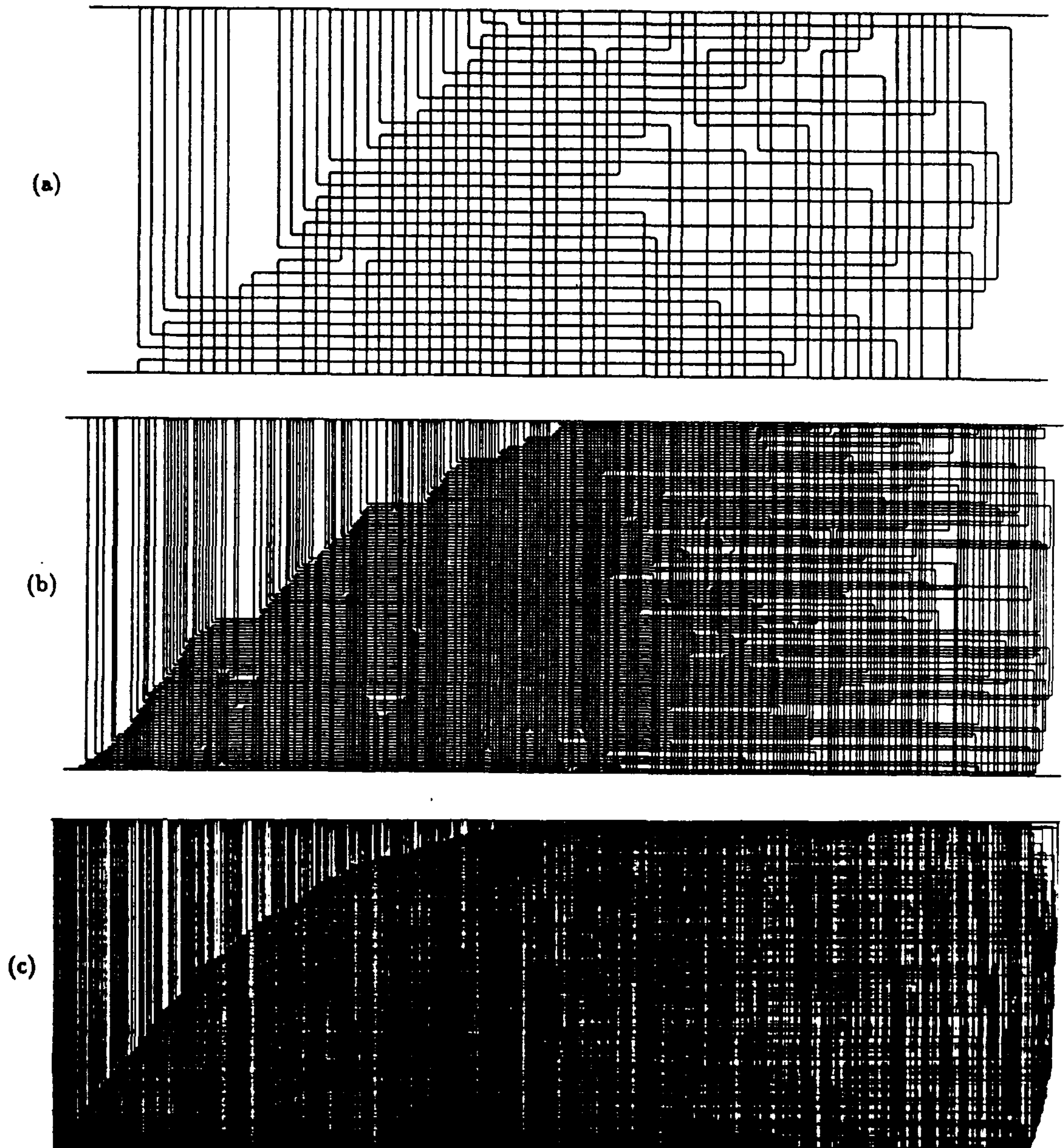



Figure 7.2 Layout of channel routing problem with (a) 53 nets, (b) 201 nets and (c) 1000 nets.

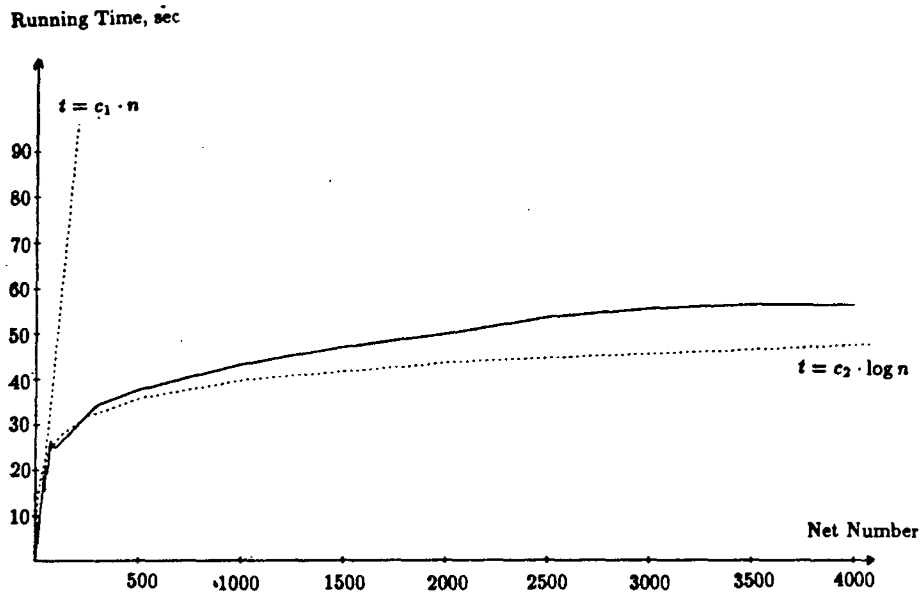



Figure 7.3 Variation of running time vs. net number

problems, such as multi-terminal routing in the knock-knee model, developing parallel heuristic for standard two layer channel routing problem et al. The main contributions of this thesis are broadly summarized below:

Because of the sequential constraints introduced by the sequential algorithm, many simple and fast serial algorithms are often hard to parallelize. To develop parallel algorithms for VLSI routing problems, we have to investigate the properties of these problems and develop completely new approaches different from the previous sequential strategies. For the river routing between rectangles, we have derived a relation between each characteristic bend point and the corresponding top terminal, which leads to an efficient parallel routing algorithm. To solve the routing problem within a rectilinear polygon, we convert the layout problem into a geometric problem of finding the union of rectilinear polygon and developed an efficient parallel algorithm for it. For the routabil-

ity testing within a rectilinear polygon, we have developed a new strategy to determine the routability by detecting the intersection between line segments or between line segment and polygon. For the routing between module pins and frame pads, we divided the problem into two parts and solve each one independently by finding out the intermediate terminals very quickly. To solve the channel routing problem in the knock-knee model, we have investigated the relations between terminals and developed a new parallel partitioning method which induced parallel routing and layer assignment algorithms. We have developed a set of properties of VLSI routing problems which lead to a better understanding of some of the theoretical aspects of these problems and introduced the completely new approaches that depart substantially from that used in the traditional sequential algorithms.

We are aiming for *efficient parallel algorithms* that run in $O(\frac{T(n)}{p} + \log^k p)$, where $p$ is the number of processors ($1 \leq p \leq n$), $T(n)$ is the running time of the best known sequential algorithm with input length $n$, and $k$ is a fixed positive constant. Most of our algorithms result in efficient parallel algorithms. Many parallel computers have only constant number of processors instead of unbounded number of processors as defined in the PRAM model. All our parallel routing algorithms can be implemented on these parallel computers with good performance. Table 7.2 is a summary of our results on the CREW-PRAM model.

Although we have discussed our algorithms in the context of the CREW-PRAM model, all of these can be mapped into other parallel computation models with efficient results. Table 7.3 is a summary of our results under other computation models.

We have also developed a completely new parallel technique – *Create Chains*, which can replace the left edge strategy or the greedy strategy for

| Problem: | Time Complexity |
|---|---|
| River Routing between Rectangles | |
| Separation Problem | $O(\frac{n}{p} + \log n)$ |
| Offset Problem | $O(\frac{n \log n}{p} + \log^2 n)$ |
| Routing within a Rectilinear Polygon | |
| Routability Testing | $O(\frac{n}{p} + \log n)$ |
| Detailed Routing | $O(\frac{n}{p} + \log n)$ |
| Wiring Module Pins to Frame Pads | |
| Routability Testing | $O(\frac{n \log n}{p} + \log^2 n)$ |
| Detailed Routing | $O(\frac{n}{p} + \log n)$ |
| Minimizing Wire Length | $O(\frac{n \log n}{p} + \log^2 n)$ |
| Channel Routing in the Knock-Knee Model | |
| Layoyt Generation | $O(\frac{n}{p} + \log n)$ |
| Layer Assignment | $O(\frac{n}{p} + \log n)$ |

Table 7.2 Summary of results on the CREW-PRAM model with $p$ processors, $1 \leq p \leq n$

| Problem: | 1 | 2 |
|---|---|---|
| River Routing between Rectangles | | |
| Separation Problem | $\sqrt{n}$ | $\log^2 n$ |
| Offset Problem | $\log n \sqrt{n}$ | $\log^3 n$ |
| Routing within a Rectilinear Polygon | | |
| Routability Testing | $\sqrt{n}$ | $\log^2 n$ |
| Detailed Routing | $\sqrt{n}$ | $\log^2 n$ |
| Wiring Module Pins to Frame Pads | | |
| Routability Testing | $\sqrt{n}$ | $\log^3 n$ |
| Detailed Routing | $\sqrt{n}$ | $\log^2 n$ |
| Minimizing Wire Length | $\sqrt{n}$ | $\log^3 n$ |
| Channel Routing in the Knock-Knee Model | | |
| Layoyt Generation | $\sqrt{n}$ | $\log^3 n$ |
| Layer Assignment | $\sqrt{n}$ | $\log^2 n$ |

1: Time complexity on $\sqrt{n} \times \sqrt{n}$ Mesh.
2: Time complexity on Hypercube, Perfect Shuffle, Shuffle Exchange and Cube Connected Cycles with $n$ processors.

Table 7.3 Summary of results on the other computation models

channel routing. With only one processor, this algorithm can be mapped into a sequential algorithm which is simpler than left edge or line packing algorithms. This new technique is useful for various routing problems, such as multi-terminal routing in the knock-knee model, a routing for the standard two layer channel routing problem.

The introduction of parallel processing in the past few years is the beginning of a new era. Up to now, most of the research is concentrated on the theoretical aspects such as data structure, graph theory, numerical computation et al. We have introduced new ideas of parallel processing to the application area of parallel CAD tools for VLSI. We believe that parallel processing can be applied to many other CAD problems such as various placement and routing problems, design rule checking problem, circuit and logic simulation, etc. We feel that new parallel strategies will be needed to handle various aspects of VLSI design.

# REFERENCE

[AHU] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, 1974.

[AKS] M. Ajtai, J. Komlos and E. Szemeredi, "An $O(n \log n)$ sorting network," Proc. 15th ACM STOC Symposium, 1983, pp. 1-9.

[AtH] M. Atallah and S. Hambrusch, " Solving tree problems on a mesh-connected processor array," Proc. 26th FOCS Symposium, 1985, pp. 222-231.

[BaP] B. Baker and R. Pinter, "An Algorithm for the Optimal Placement and Routing of a Circuit Within a Ring of Pads," Proc. 24th FOCS Symposium, 1983, pp. 360-370.

[Bar] G. H. Barnes et al. "The ILLIAC IV Computer," IEEE Trans. on Computer, C-17, 1968, pp. 746-757.

[Bat] K. E. Batcher, "Sorting Networks and Their Applications," Proc. 1986 Spring Joint Computer Conference, vol 32, pp. 307-314.

[BBL] B. S. Baker, S. N. Bhatt and F. T. Leighton, " An Approximation Algorithm for Manhattan Routing," Proc. 15th ACM STOC Symposium, 1983.

[BDH] D. Bitton, D. J. DeWitt, D. K. Hsiao and J. Menson, "A Taxonomy of Parallel Sorting" Computing Survey, Sep. 1984, pp 287-318.

[BHP] M. Burstein, S. J. Hong and R. Pelavin, " Hierarchical VLSI Layout : Simultaneous Placement and Wiring of Gate Array," Proc. International Conf. VLSI-83, Trondheim, Norway, 1983, pp. 45-60.

[BP1] M. Burstein and R. Pelavin, " Hierarchical Channel Router," INTEGRATION, the VLSI Journal, 1, 1983, pp. 21-38, also in the Proc. 20th Design Automation Conference, 1983, pp. 591-597.

[BP2] M. Burstein and R. Pelavin, " Hierarchical Wire Routing," IEEE Trans. on CAD, October 1983, pp. 223-234.

[BrB] M. Brady and D. Brown, "VLSI Routing: Four Layers Suffice," Advances in Computing Research 2 (VLSI Theory), ed. Preparata, JAI Press, Inc., Greenwich, CT, pp. 245-257, 1984.

[BrR] D. J. Brown and R. L. Rivest, "New Lower Bounds for Channel Width," Proc. 1981 CMU Conf. VLSI Systems and Computations, 1981, pp. 178-185.

[CJ1] S. C. Chang and J. JáJá, "Parallel Algorithms For River Routing," Proc. International Conference on Parallel Processing, 1988, also submitted to SIAM Journal on Computing for publication.

[CJ2] S. C. Chang and J. JáJá, "Parallel Algorithms for Wiring Module Pins to Frame Pads", Technical Report UMIACS-TR-88-2, CS-TR-1970, University of Maryland, College Park, Jan. 1988, also Submitted to IEEE Transaction on Computer for publication.

[CJ3] S. C. Chang and J. JáJá, "Parallel Algorithms for Channel Routing in the Knock-Knee Model", Proc. International Conference on Parallel

Processing, 1988, also submitted to SIAM Journal on Computing for publication.

[CJ4] S. C. Chang and J. JáJá, "Optimal Mesh Algorthms for VLSI Routing", Proc. Frontiers '88: The Second Symposium on the Frontiers of Massively Parallel Computation, 1988.

[Col] R. Cole, " Parallel Merge Sort", Proc. 27th FOCS Symposium, 1986, pp. 511-516.

[CoS] R. Cole and A. Siegel, " River routing every which way, but loose," Proc. 25th FOCS Symposium, 1984, pp. 65-73.

[CV1] R. Cole and U. Vishkin "Deterministic Coin Tossing and Accelerating casecades : Micro and Macro Techniques for Designing Parallel Algorithms," Proc. 18th ACM STOC Symposium, 1985, pp. 206-219.

[CV2] R. Cole and U. Vishkin "Approximate and Exact Parallel Scheduling with Application to List, Tree and Graph Problems," Proc. 27th FOCS Symposium, 1986, pp. 478-491.

[De1] D. N. Deutsch, "A Dogleg Channel Router," Proc. 13th Design Automation Conference, 1976, pp. 425-433.

[De2] D. N. Deutsch, "Solutions to a Switchbox Routing Problem," IEEE Trans. on CAD, 1985.

[DKS] D. Dolev, K. Karplus, A. Seigel, A. Strong and J. Ullman, "Optimal wiring between rectangles," Proc. 13th ACM STOC Symposium, 1981, pp. 312-317.

[DyR] D. W. Dymond and W. L. Ruzzo, "Parallel Random Access Machines with Owned Global Memory and Deterministic Contex Free Language

Recognition," Proc. 13th International Colloquium on Automata, Languages and Programming, 1986, pp. 95-104.

[FoW] S. Fortune and J. Wyllie, "Parallelism in random access machines," Proc. 10th ACM STOC Symposium, 1978, pp. 114-118.

[GaJ] M. R. Garry and D. S. Johnson, *Computers and Intractability.* W. H. Freeman and Co., Reading, 1979.

[Gol] L. Goldschlager, "A Unified Approach to Models of Synchronous Parallel Machines," Proc. 10th ACM STOC Symposium, 1978, pp. 89-94.

[GoM] S. Goto and T. Matsuda, "Partitioning, Assignment and Placement," Layout Design and Verification, Advances in CAD for VLSI, vol. 4, North-Holland, 1986, pp. 55-97.

[Hak] S. L. Hakimi, "Steiner's Problem in Graphs and its Implications," Networks, 1(1971), pp. 113-133.

[Ham] S. E. Hambrusch, "Channel Routing Algorithms for Overlap Models," IEEE Trans. on CAD, CAD-4, 1, 1985.

[HaO] G. Hamachi and A. Ousterhout, "A Switch Box Router with Obstacle Avoidance," Proc. 21st Design Automation Conf. 1984, pp. 173-179.

[Hil] W. D. Hillis, *The Connection Machine*, MIT Press, Reading, 1985.

[Hir] D. S. Hirschberg, "Fast Parallel Sorting Algorithms," Commun. ACM, Aug. 1978, pp. 657-666.

[HiS] W. D. Hillis and G. L. Steele Jr., "Data Parallel Algorithms," Commun. ACM, Dec. 1986, pp. 1170-1183.

[HoU] J. E. Hopcroft, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, 1979.

[Hs1] C. P. Hsu, "A New Two-Dimensional Routing Algorithm," Proc. 19th Design Automation Conf. 1982, pp. 46-50.

[Hs2] C. P. Hsu, "General River Routing Algorithms," Proc. 20th Design Automation Conf. 1983, pp. 578-583.

[HwB] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, Reading, 1984.

[Joh] D. Johannsen, "Bristle blocks: a silicon compiler," Proc. 16th Design Automation Conference, June 1979, pp. 310-313.

[KrL] M. Kramer and J. van Leeuwen, "Wire routing is NP-complete," technical report, University of Utrecht, the Netherlands, February 1982.

[KrJ] S. Krishnamurthy and J. JáJá, "Provably Good Parallel Algorithms for Channel Routing of Multi-terminal Nets," submitted for publication.

[KR1] C. Kruskal, L. Rudolph and M. Snir, "The Power of Parallel Prefix," IEEE Tran. on Computers, Oct. 1985, pp. 965-968.

[KR2] C. P. Kruskal, L. Rudolph and M. Snir, "A Complexity Theory of Efficient Parallel Algorithms," Proc. 15th International Colloquium on Automata, Language and Programming, 1988.

[Kuc] L. Kucera, "Parallel Computation and Conflicts in Memory Access", Information Processing Letter, 14, 1982, pp. 93-96.

[KuM] E. S. Kuh and M. Marek-Sadowska, "Global Routing," Layout Design and Verification, Advances in CAD for VLSI, vol. 4, North-Holland, 1986, pp. 169-198.

[LaP] A. LaPaugh, "Algorithms for integrated circuit layout: an analytic approach," Ph.D. dissertation, MIT, Cambridge, MA, November 1980.

[Lei] C. Leiserson, "Area efficient layouts," Proc 21th FOCS Symposium, 1980.

[Leg] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," Proc. 16th ACM STOC Symposium, 1984, pp. 71-80.

[LeM] C.E. Leiserson and F.M. Maley, "Algorithms for routing and testing routing of planar VLSI layout," Proc. 17th ACM STOC Symposium, May 1985, pp. 69-78.

[LeP] C. Leieserson and R. Pinter, "Optimal placement for river routing," SIAM J. on Computing, August 1983, pp. 447-462.

[Lip] Lipski, W., "On the Structure of Three-Layer Wirable Layouts," Advances in Computing Research 2 (VLSI Theory), ed. Preperata, JAI Press, Inc., Greenwich, CT, pp. 231-243, 1984.

[MaK] M. Marek-Sadowska and E. S. Kuh, "A New Approach to Channel Routing," Proc. ISCAS-82, Rome, 1982, pp. 764-767.

[MeC] C. Mead and L. Conway, *Introduction to VLSI systems*, Addison-Wiley, Reading, 1980.

[MeP] Melhorn, K. and F. Preparata, "Routing through a rectangle," Journal of the ACM, Jan. 1986, pp.60-85.

[MiR] G. L. Miller and J. H. Reif, "Parallel Tree Construction and its Applications," Proc. 26th FOCS Symposium, 1985, pp. 478-489.

[MiS] R. Miller and Q. Stout, "Mesh Computer Algorithms for Computational Geometry," Technical Report 86-18, Dept. Computer Science, State University of New York at Buffalo, July 1986.

[Mir] A. Mirzaian, " Channel routing in VLSI," 16th ACM STOC Symposium, 1984, pp. 101-107.

[MPS] K. Mehlhorn, F. P. Preparata and M. Sarrafzadeh, "Channel Routing in Knock-knee Mode : Simplified Algorithms and Proofs," Algorithmica, vol. 1, Oct. 1986, pp. 213-221.

[MuP] D. E. Muller and F. P. Preparata, "Bounds to Complexities of Networks for Sorting and for Switching," Journal of the ACM, Apr. 1975, pp. 195-201.

[NaS] D. Nassimi and S. Sahni, "Data Broadcasting in SIMD Computers," IEEE Trans. on Computers, Feb. 1981, pp. 101-107.

[Oht] Ohtsuki, T., "Layout Design and Verification," Advances in CAD for VLSI, vol. 4, North-Holland, 1986.

[Pin] R. Pinter, "River routing: methodology and analysis," Proceedings of the third CALTECH Conference on Very Large Scale Integration, March 1983, pp. 141-163.

[Pip] N. J. Pippenger, "On Simultaneous Resource Bounds," Proc. 20th FOCS Symposium, 1979, pp. 307-311.

[Pre] F. P. Preparata, "New Parallel Sorting Schemes," IEEE Trans. Computers, July 1978.

[PrL] F. P. Preparata and W. Lipski, "Optimal Three-Layer Channel Routing," IEEE Trans. on Computers, 1984, pp. 427-437.

[PrV] F. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," Comm. of the ACM, May 1981, pp. 300-309.

[RBM] R. L. Rivest, A. E. Baratz and G. Miller, "Provable Good Channel Routing Algorithms," Proc. 1981 CMU Conf. VLSI Systems and Computations, 1981, pp. 153-159.

[Ree] A. P. Reeves, "A Systematically Designed Binary Array Processor," IEEE Trans. on Computers, vol C-29, 1980, pp. 278-287.

[Ric] R. Rice, "VLSI: The Coming Revolution in Applications and Design," Compcon Spring'80, Feb. 1980, pp. 19-20.

[RiF] R. L. Rivest and C. M. Fiduccia, "A "Greedy" Channel Router," Proc. 19th Design Automation Conference, 1983, pp. 591-597.

[Rub] S. M. Rubin, *Computer Aids for VLSI Design*, Addison-Wesley, Reading, Mass. 1987.

[SaB] S. Sahni and A. Bhatt, "Complexity of the Design Automation Problem," Proc. 17th Design Automation Conference, June 1980, pp. 402-411.

[SaP] M. Sarrafzadeh and F. P. Preparata, "Compact Channel Routing of Multiterminal Nets," Annals of Discrete Mathematics 25, 1985, pp. 255-280.

[Sar] M. Sarrafzadeh, "Channel Routing with Provably Short Wires," IEEE Trans. on Circuits and Systems, sep. 1987, pp. 1133-1135.

[SaS] A. Sangiovanni-Vincentelli and M. Santomauro, "YACR : Yet Another Channel Router," Proc. Custom Integr. Circuits Conf., Rochester, NY, 1982, pp. 460-466.

[Sch] J. Schwartz, "Ultracomputers," ACM Trans. on Programming Languages and Systems, 2, 1980, pp. 484-521.

[SeD] A. Seigel and D. Dolev, "The separation for general single layer wiring barriers," Proceedings of the CMU Conference on VLSI Systems and Computations, October 1981, pp. 143-152.

[SSR] A. Sangiovanni-Vincentelli, M. Santomauro and J. Reed, "A New Gridless Yet Another Channel Router the Second (YACR-II)," Proc. ICCAD, 1984.

[Sto] H. S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Computers, Feb. 1971, pp. 153-161.

[Stu] Q. F. Stout, "Mesh-Connected Computers with Broadcasting," IEEE Trans. on Computers, Sep. 1983, pp. 826-830.

[Sup] P. Su, Master thesis research, University of Maryland, College Park.

[Szy] T. Szymanski, "Dogleg Channel routing is NP-complete," manuscript, Bell Laboratories, Murray Hill, NJ, September 1981.

[Tho] C. D. Thompson, "Area-time complexity for VLSI," Proc. 11th ACM STOC Symposium, 1979.

[TMC] **Lisp Reference Manual*, Thinking Machine Corporation, Cambridge, Massachusetts, 1987.

[Tom] M. Tompa, "An optimal solution to a wire routing problem," Proc. 12th STOG Symposium, 1980, pp. 161-176.

[UKS] K. Ueda, R. Kasai and T. Sudo, "Layout Strategy, Standardization, and CAD Tools," Layout Design and Verification, Advances in CAD for VLSI, vol. 4, North-Holland, 1986, pp. 1-54.

[Ull] J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Reading, 1984.

[ViS] J. S. Vitter and R. A. Simons, "New Classes for Parallel Complexity: a Study of Unification and Other Complete Problems for P," IEEE Trans. on Computers, May 1986, pp. 403-418.

[Vi1] U. Vishkin, "Randomized Speed-Ups in Parallel Computation," Proc. 16th ACM STOC Symposium, 1984, pp. 230-239.

[Vi2] U. Vishkin, "Implementation of Simultaneous Memory Address Access in Models that Prohibit it," Journal of algorithms, 4, 1983, pp. 45-50.

[WaH] R. A. Wagner and Y. Han, "Parallel Algorithms for Bucket Sorting and the Data Dependent Prefix Problem," Proc. International Conference on Parallel Processing, 1986, pp. 924-930.

[WeE] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, Reading, Mass. 1985.

[Wis] D. S. Wise, "Computer Layouts of Banyan/FFT Networks," VLSI Systems and Computations, Computer Science Press, Rockvill, MD, 1981, pp. 186-195.

[Wyl] J. C. Wyllie, "The Complexity of Parallel Computation," PhD Thesis, Department of Computer Science, Cornell University, 1979.

# CURRICULUM VITAE

Name: Shing-Chong Chang

Permanent Address: 3426 Tulane Dr. #33
Hyattsville, MD 20783

Degree and date to be conferred: Ph.D., 1988

Date of birth: October 22, 1958

Place of birth: Taipei, Taiwan, R.O.C.

Secondary education: Taichung First High School, Taiwan, R.O.C., 1977

| **Collegiate Institutions** | **Dates** | **Degree** | **Date of Degree** |
|---|---|---|---|
| University of Maryland, College Park, MD 20742. | 1985–1988 | Ph.D.(EE) | 1988 |
| National Taiwan University, Taipei, Taiwan, R.O.C. | 1981–1983 | M.S.(EE) | 1983 |
| National Taiwan University, Taipei, Taiwan, R.O.C. | 1977–1981 | B.S.(EE) | 1981 |

Major: Electrical Engineering

**Professional Publications:**

**(1)** "Logical Design of High Level Protocols for Local Area Network", with Tai-Ming Parng, in Proc. of 4th IEEE Int. Conf. on Distributed Computing System, San Francisco, May 1984, pp 166-172.

**(2)** "Parallel Algorithms for River Routing", with Joseph Ja'Ja', Proceedings, International Conference on Parallel Processing, Chicago, August 1988, also submitted to SIAM Journal on Computing for publication.

**(3)** "Parallel Algorithms for Wiring Module Pins to Frame Pads", with Joseph Ja'Ja', Technical Report UMIACS-TR-88-2, CS-TR-1970, University of Maryland, College Park, January 1988, also Submitted to IEEE Transaction on Computer for publication.

**(4)** "Parallel Algorithms for Channel Routing in the Knock-Knee Model", with Joseph Ja'Ja', Proceedings, International Conference on Parallel Processing, Chicago, August 1988, also submitted to SIAM Journal on Computing for publication.

**(5)** "Optimal Mesh Algorthms for VLSI Routing", with Joseph Ja'Ja', Proceedings, Frontiers '88: The Second Symposium on the Frontiers of Massively Parallel Computation, 1988.

**(6)** *Guide to the IBM Personal Computer*, (book in Chinese), Ju-Lin Publication Co., Ltd., Taipei, Taiwan, R.O.C. 1984, 350 pages.

**(7)** *Interfacing to the IBM Personal Computer*, (book in Chinese), Ju-Lin Publication Co., Ltd., Taipei, Taiwan, R.O.C. 1984, 370 pages.

**(8)** "Protocol Design for Local Area Network Distributed Operating System", Master Thesis, Department of Electrical Engineering, National Taiwan University, June 1983.

**(9)** "Parallel Algorithms for Several VLSI Routing Problems", Ph.D. Thesis, Department of Electrical Engineering, University of Maryland, College Park, August 1988.

**Professional Positions held:**

| | |
|---|---|
| 9/88- | Research Staff Member<br>IBM Thomas J. Watson Research Center<br>P.O. Box 218, Yorktown Heights, New York 10598 |
| 7/86-8/88 | Graduate Research Fellow<br>Systems Research Center<br>University of Maryland, College Park, MD 20742 |
| 1/86-6/86 | Teaching Assistant<br>Department of Electrical Engineering<br>University of Maryland, College Park, MD 20742 |
| 8/83-7/85 | Instructor<br>Department of Electronic Engineering<br>Szehai College of Technology, Taipei, Taiwan, R.O.C. |
| 7/81-6/83 | Research Assistant<br>Department of Electrical Engineering<br>National Taiwan University, Taipei, Taiwan, R.O.C. |

| | |
|---|---|
| 7/80-6/81 | Programmer<br>Ching Ling Industry Research Center, Taipei, Taiwan, R.O.C. |