Parallel Algorithms For Automating VLSI Physical Design

Ravikumar P. Chennagiri

CENG Technical Report: 91-25

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Electrical Engineering)

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
(213)740-4481

PARALLEL ALGORITHMS FOR AUTOMATING VLSI PHYSICAL DESIGN

Ph.D. Dissertation
by

Ravikumar P. Chennagiri

August 1991

Guidance Committee

Dr. Sarma Sastry (Chairperson)
Dr. Alice Parker
Dr. C.S. Raghavendra
Dr. Richard Arratia (Outside Member)

Dedication

To My Parents

Acknowledgements

I would like to thank my thesis advisor Dr. Sarma Sastry for offering me his support, understanding, and guidance throughout my graduate career. I am grateful to Dr. Alice Parker, Dr. C.S. Raghavendra, and Dr. Richard Arratia for carefully reading my thesis proposal and my dissertation; their advice has been very helpful. Financial support for this research came from a grant by the Powell Foundation and from a scholarship awarded to me by ACM/SIGDA. The Department of EE-systems offered me a teaching assistantship during my first two graduate years. The School of Journalism awarded a teaching assistantship to me for one semester. The Computer Science Department kindly allowed me to make use of the Intel iPSC/2 hypercube computer. The University Computing Services permitted the use of Alliant FX/80.

I am grateful to Dr. V.K. Prasanna Kumar (EE-Systems Department, USC), Prof. George Zobrist (Computer Science Department, University of Missouri-Rolla), and Prof. L. M. Patnaik (Department of Computer Science and Automation, Indian Institute of Science) for their encouragement.

I thank my friends at USC who made my graduate years memorable. Bill Bates, the graduate administrator in the EE-systems Department, and Laura Yoneda, the graduate administrator in the School of Journalism have both been great friends; Dr. Baron Grey, part-time instructor at USC, has been a good friend and great help in solving some of my LaTeX problems; Carrol Gordon and Lea Vasquez have cheerfully offered secretarial assistance.

And last, but certainly not the least, my thanks go to my parents (Mrs. Komala Rao and Mr. C.H. Prahlada Rao) and my brother C.P. Mohan for their encouragement and affection.

Contents

List Of Tables

List Of Figures

Abstract

Since the birth of VLSI in late seventies, the past decade has seen an extraordinary growth in the level of integration. Unfortunately, the tools for VLSI design have not kept pace with this growth, due to the computational complexity of design problems. Typically, the execution time of a VLSI design algorithm grows at a rate faster than the size of the design problem. As a result, design tools exhibit poor speed performance for large problem sizes. Code optimization and higher clock rates are only tentative solutions to this problem. To obtain an order of magnitude improvement in performance, parallel processing is the only logical alternative. This dissertation discusses the application of parallel processing techniques to VLSI layout applications.

Most subproblems related to VLSI layout are computationally hard *combinatorial optimization* problems. There are three ways in which parallel processing can be used to solve hard optimization problems. The first is to parallelize good heuristic algorithms that give near-optimal solutions. The second alternative is to take advantage of the inherent structure in some of the subproblems and attempt exact solutions; the theory of *perfect graphs* is useful in following this approach. Many layout problems can be reduced to optimization problems on perfect graphs and can be solved exactly in polynomial time. The third alternative is to use *parallel randomized search*, where multiple processors are used to concurrently search the combinatorial state space of the optimization problem. All these approaches are examined for a particular application, namely, VLSI layout. Experimental results on two computing platforms are discussed, the Intel iPSC/2 and the Alliant FX/80.

Chapter 1

Introduction

1.1 VLSI Design and Parallel Processing

Since the birth of VLSI in late seventies, the past decade has seen an extraordinary growth in the level of integration. Unfortunately, the tools for VLSI design have not kept pace with this growth, due to the computational complexity of design problems. Typically, the execution time of a VLSI design algorithm grows at a rate faster than the size of the design problem. As a result, design tools exhibit poor speed performance for large problem sizes. Code optimization and higher clock rates are only partial solutions to this problem. To obtain an order of magnitude improvement in performance, parallel processing is the only logical alternative.

In the past few years, parallel supercomputing has become a reality. A number of scientific applications such as fluid dynamics, theoretical physics, structural dynamics and seismic processing have benefited significantly from the higher speed offered by parallel computing. Being CPU-intensive, VLSI design is also a good candidate for parallel processing. Consider the following statistics.

- More than 50 hours of VAX 8800 CPU time were expended in simplifying an industrial PLA example using the Espresso logic minimization program [M+90].
- 84 VAX-hours were reported for placement of 2700 standard cells by a program based on Simulated Annealing [Kra86].

- On a 1 MIPS CPU, 5000 hours were necessary to run a fault simulation of a chip with 100,000 gates [MH85].
- A performance-directed placement algorithm took 16 hours on a VAX 8800 to place a sea-of-gates circuit with 1400 cells [JK89].
- Circuit verification required more than 10 hours of VAX 8800 CPU time when applied to some benchmark sequential circuits [AGN90].

Why are design algorithms CPU-intensive? This chapter will attempt to find answers to this question. The need for faster design tools in a VLSI design environment is then brought out. This serves as the motivation for employing parallel processing in electronic design automation. The process of VLSI design is then explained, with emphasis on physical design aspects. Finally, the organization of this dissertation is outlined.

1.2 Complexity of Design Problems

VLSI design is a complex process, composed of several subtasks such as logic design, logic simulation, circuit simulation, layout, and design verification. Each of these phases is computationally intensive due to four chief reasons :

1. Combinatorial optimization is involved in many stages of VLSI design. These optimization problems are, by and large, NP-complete [GJ79]. Polynomial-time algorithms which guarantee exact solutions are hitherto unknown for most design problems such as logic minimization, partitioning, placement, routing, and compaction. Therefore, heuristic algorithms are used which run in polynomial time and produce nearly optimal designs. However, this is not telling the full story. Most heuristic design algorithms are superlinear in time complexity i.e. their execution time grows faster than the rate at which the problem size is increased. As a result, when the problem size is large, even heuristic algorithms demand huge amounts of CPU-time. This is especially true for

the new class of optimization algorithms such as simulated annealing, genetic algorithms and stochastic learning algorithms.

2. Many design steps are numerically intensive, involving enormous amounts of floating-point arithmetic. Device-level circuit simulation, for example, involves numeric problems such as solving systems of linear equations and solving systems of integro-differential equations. Several physical design algorithms also involve numerical computation. The force-directed placement algorithm formulates the circuit placement problem as a non-linear optimization problem and applies the Newton-Raphson procedure to solve the problem. Placement and routing algorithms based on Simulated Annealing involve a great deal of floating-point arithmetic. Hall's algorithm for circuit placement approaches the problem by formulating it as an eigensystem computation. These algorithms are described in subsequent chapters of the dissertation.

3. VLSI design tools are forced to handle large problem sizes due to the increased level of integration and packaging density. In fact, the state-of-the-art VLSI foundries can handle much larger problem sizes than today's CAD tools can handle [Rav90a]. VLSI designs with a million transistors are not uncommon today. This means layout programs must be capable of placing thousands of cells and be able to route thousands of nets. Logic minimization programs must be capable of dealing with a hundred literals or more. Circuit simulators must be able to handle hundreds of thousands of devices.

4. VLSI design is an iterative process. As explained later in this chapter, designing a VLSI chip involves several subproblems, namely, functional design, logic design, circuit design and physical design. After each of these steps, the design is verified for correctness through a process of simulation. Thus functional simulation follows functional design, logic simulation follows logic design, and device-level circuit simulation follows circuit design. The physical design is verified for the presence design rule errors. In order to make sure that the physical layout of the chip actually corresponds to the intended circuit, a

circuit is *extracted* from the layout and simulated at the device level. If design errors are found at any of the verification steps, one or more of the previous design steps must be repeated to correct the errors. Similarly, design steps are repeated when the performance of the design does not match its specification. Due to this "iterative" nature of the design process, design tools must be used several times during the design life cycle.

1.3 The need for speed

It is highly desirable that design tools be as fast as possible, since circuit design is an iterative process. Due to intense competition, time is a critical resource during the development of a new integrated circuit. Especially when application-specific integrated circuits (ASIC) are involved, a design house cannot afford the risk of having a slow software tool. Sometimes circuit quality is sacrificed for time e.g., a fault simulation is not carried out at all, or a compaction algorithm is not executed to optimize chip area. When the design software is interactive, its failure to produce quick responses will seriously affect the creativity of the design engineer.

1.4 Towards Parallel Solutions

Ironically, the power to meet the VLSI design challenge is to be found in VLSI itself. The new generation of parallel computers have been made possible by the VLSI revolution. It is now an accepted fact that only parallel processing can deliver the *order of magnitude* improvement in performance demanded by complex design problems. Furthermore, parallel processing is a *scalable technology*; in other words, when the problem size increases by a factor, the required increase in computational power can be obtained by adding more processors.

There are two fundamentally different ways in which parallel processing can be used in a design environment – special-purpose parallel computing and general-purpose parallel computing. A special-purpose parallel computer, also known as a

"Hardware Accelerator," a "Point Accelerator," or a "Special-purpose Engine," is a machine designed to execute a single algorithm efficiently. Since programmability is not required of a hardware accelerator, it is possible to avoid the overhead of fetching instructions from memory and decoding instructions; the algorithm is "hard coded" into the control unit of the accelerator. Further, the architecture of a hardware accelerator is designed specifically to take advantage of the computational and communication structures of the algorithm. A general-purpose parallel computer, also known as a "Parallel Supercomputer," or simply, a "Supercomputer," is intended for a broader class of applications. A supercomputer comes with a programming environment comprised of parallel programming languages, parallelizing compilers and a multi-tasking operating system.

1.4.1 Hardware Accelerators

Pioneers of VLSI research advocated the design of special-purpose chips and point accelerators [Kun79, MC80]. For example, array processors were designed for sorting, signal processing, and image processing applications. These special-purpose machines are integrated into a computing system as attachments to the host CPU (see Figure 1.1). Input data from the host is uploaded into the accelerator through a high speed bus. After the accelerator completes processing the input data, the output is downloaded back to the host. Typically, point accelerators are built as systolic arrays, pipelines or SIMD arrays. A systolic array consists of a large number of identical, low-granularity processing elements. These processors are connected through near-neighbor links to form highly regular patterns such as linear array, rectangular mesh, hexagonal array, cube, tree, or pyramid. All processors execute the same algorithm, which is hardwired into the control unit of the processor. Systolic arrays have been proposed for electronic design rule verification [KS84].

Pipelined computing is the least expensive form of parallel processing. When the core of an algorithm consists of a loop which is executed a large number of times, it pays to speed up the computations performed within the loop by using a pipeline. If the loop consists of k steps, $S_1, S_2, \cdots, S_k$, a k-stage pipeline is designed, where

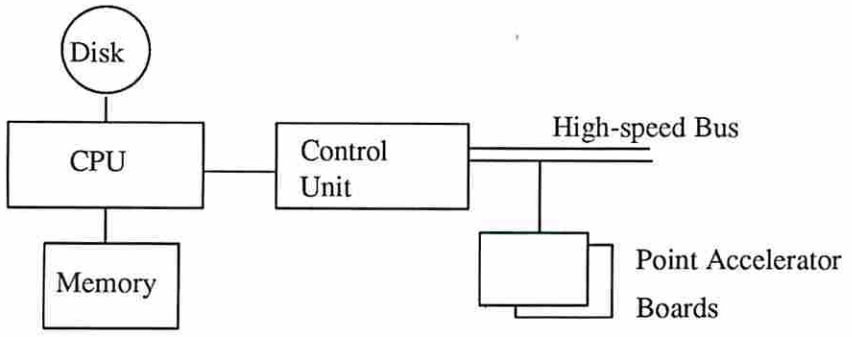

Figure 1.1: Attaching Special-Purpose Systems to a Host CPU

the ith stage executes step i, $1 \leq i \leq k$. Let the notation ${S^j}_i$ indicate the execution of step S_i during iteration j. Then temporal parallelism permits the simultaneous execution of ${S^j}_1$, ${S^{j-1}}_2$, $\cdots$, ${S^{j-k+1}}_k$. If the number of iterations n is sufficiently large compared to k, the pipeline yields a speedup of k, provided that each step requires the same execution time τ. When there is a speed mismatch between steps, buffering is necessary between stages so that a smooth flow of data can be maintained in the pipeline. Pipelined computers have been proposed for circuit placement [RP90, I+83] and Lee-style routing [W+88].

An SIMD array consists of a set of processing elements (PEs) which are controlled by a master control unit. The PEs are connected by means of a regular interconnection network. The control unit interfaces to the PEs through a control bus. Each PE is equipped with a private set of registers to store data elements that are private to the PE; the PE can carry out arithmetic and logic operations on its private data. In addition, a PE can send/receive data from its neighbor PEs. The algorithm is either hardwired or microprogrammed into the control unit. This arrangement eliminates the overhead of *instruction fetch* and *instruction decode*, resulting in a manifold improvement in speed performance. SIMD arrays exploit the *data parallelism* in the algorithm. If the same operation must be applied to all the elements in a data array, the data elements are distributed among the PEs. The control unit broadcasts an opcode to all the PEs over the control bus. Each PE carries out the appropriate operation on its data element. SIMD arrays have been

used for circuit placement [RS88, RSP91, CB83], global routing [RS89a] and Lee routing [S+86].

Designing and manufacturing a special-purpose accelerator is a skilled task and requires a great deal of time and resources. As a result, special-purpose machines tend to be expensive. Nonetheless, an accelerator delivers superior performance since it is fine-tuned for a specific application. But the fact that an accelerator can only run a single algorithm can be a handicap, especially in the constantly changing field of computer-aided circuit design; CAD programs need to be updated often due to changing design styles, design rules, new objective functions and new computing paradigms.

1.4.2 Supercomputers

Since supercomputers are programmable, they can overcome the problem of inflexibility mentioned above. In the past five years, general-purpose parallel computers have become increasingly available and affordable. Full-scale supercomputers such as CRAY-2, CRAY X-MP, and IBM GF-11 offer a performance of several thousand MIPS (Million Instructions Per Second) and are priced in the range $2M to $20M. Near-supercomputers such as IBM 3090 and Connection Machine deliver 50 to 100 MIPS and cost $1M to $4M. Minisupers such as Alliant FX/80 and Intel iPSC/2 provide a performance of 10 to 50 MIPS and are priced in the range $100K to $1.5M. The prices of supercomputers are falling and will continue to fall with improvements in VLSI design and fabrication processes. As a result, general-purpose supercomputing is becoming more and more cost-effective. In recent years, this has had an impact on design philosophy; focus has slowly shifted from developing special-purpose hardware to developing software for general-purpose supercomputers.

The flexibility of supercomputers, however, demands its price; maintenance and development of software for a parallel computer is no small task. Programming a parallel machine calls for a great deal of expertise. The programmer must have a good understanding of the machine's architecture in order to develop efficient programs. Debugging tools for parallel computers are either not available or are

quite primitive. These difficulties are expected to disappear in the coming years. Already parallel programming languages, parallelizing compilers and multi-tasking operating systems are available in the commercial market. For example, the Alliant FX/80 supports a parallelizing compiler for Fortran (Appendix A). However, today's parallelizing compilers can only detect obvious parallelism in the source code. DO loops of Fortran and vector operations are efficiently parallelized. But other inherent parallelism in the algorithm may go undetected by the compiler. A programmer must express this inherent parallelism through programming constructs. Due to increased availability of parallel processors, there is a growing awareness of parallel programming skills. As a result, there is growing optimism about the future of parallel supercomputing as a robust technology [Hwa87, Wil88]. It is the aim of this dissertation to illustrate how to harness the power of parallel computing for problems in VLSI physical design.

1.5 Physical Design of VLSI Circuits

1.5.1 Design Hierarchy

A top-down (or hierarchical) design methodology is often used in designing VLSI systems. Figure 1.2 shows the different stages in this hierarchy. The first stage is **design specification**, where the goals of the final product are written down. Typical considerations during system specification are the targeted cost of the system, its performance, architecture, and how the system will interface to the outer world. Design specification is provided by human experts.

The second stage is called **functional design**, and consists of refining the design specification in order to define the functional behavior of the system. For example, the functional behavior of an instruction-set processor is defined by its instruction set and a register level design. A register level design uses circuit blocks such as arithmetic/logic units, shift registers, multiplexers, busses, and control units. The objective of functional design is to generate a high performance architecture within the cost constraints posed by the design specification. Since there can be literally

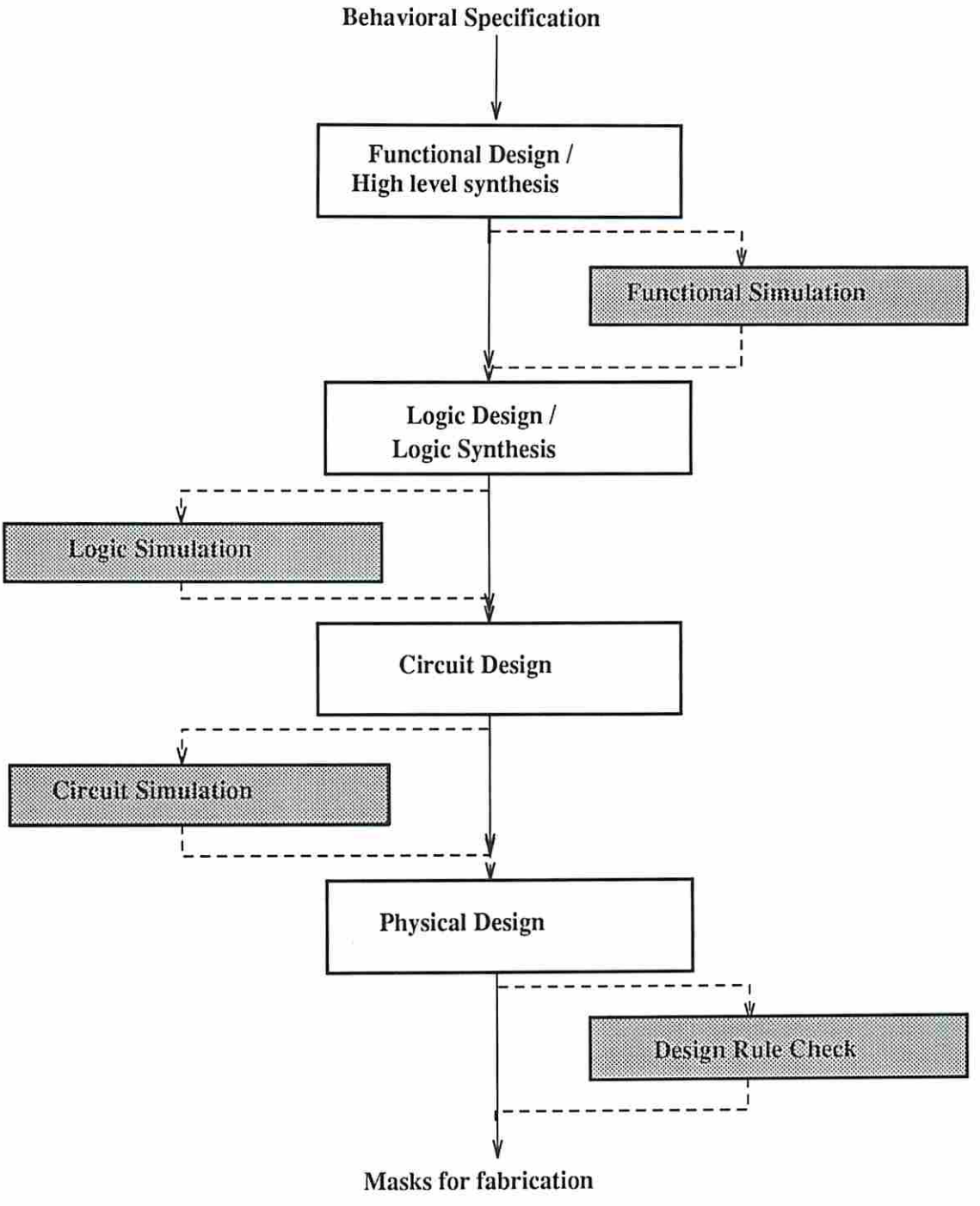

Figure 1.2: Steps in VLSI Design Hierarchy. Synthesis steps are shown in white rectangles. Analysis steps are shown in light gray rectangles.

hundreds of architectures to choose from, functional design calls for human expertise. However, if the target architecture is fixed to say, pipelined architecture or single-bus architecture, the problem of functional design can be formulated more precisely. Over the past few years, a significant amount of research effort has been spent on automating the process of functional design. This research area is popularly known as *high level synthesis*. Many tools have been developed to solve specific problems in high level synthesis. An example is the ADAM system [JKMP89] developed at the University of Southern California, which can generate a register level design for a digital circuit starting from a "data flow graph" specification of the circuit. Computers can be used to simulate the behavior of the functional design to get some early feedback regarding its correctness and performance; this step is known as **functional simulation**, *high level simulation*, or *architectural simulation*.

The third step in the design process is **logic design**, where *structure* is added to the behavioral representation of the design. For example, the register level design is refined by using finite state machines and combinational logic blocks. The important concerns of logic design are logic minimization, performance enhancement and testability. The number of *literals* in a set of Boolean equations associated with a design indirectly controls the number of transistors in the final implementation of the logic. Therefore, many logic minimization tools attempt to minimize the number of literals. It is equally important to avoid *races* and *hazards* in the designed logic. Logic design must also consider the problems of test vector generation, error detection and error correction. Logic synthesis tools have been developed for automate the process of logic design [BHMSV84]. Following the completion of logic design, **logic simulation** is carried out to verify the correctness of design logic. Logic simulation is discussed further in [ABF90].

The fourth phase is called **circuit design**, where logic blocks are designed using electronic devices such as transistors, resistors and capacitors. **Circuit simulation** is carried out at this stage in order to verify the timing behavior of the system. Each device is suitably modeled to account for its physical behavior. Kirchoff's laws are then used to express the behavior of the electronic circuit in terms of node

voltages and branch currents. The resulting set of integro-differential equations are then solved in discrete-time i.e. node voltages and branch currents are computed at discrete time intervals. Voltage at a specific node, or current in a specific branch can then be plotted as a function of time. Signal delays from input to output can be computed using the above data. If the delay at a particular output O is unacceptably large or small, transistors along the path from the input to the output O need to be appropriately sized so as to adjust their switching delays. SPICE is a well known program for circuit simulation (see [VZN+81]).

1.5.2 Physical Design

The final step in the process of hierarchical design is called **physical design**. It is concerned with the actual layout of the system – where the components will be placed and how they will be interconnected. Physical design is perhaps the most researched topic in the area VLSI system design. For good reasons. The actual layout of the system can drastically affect the area, correctness, and performance of the final product. While predicting the cost and performance of the system, the higher stages in design hierarchy use *estimators* for the area and system delay. Some times, only the *functional area* of the system, namely the area occupied by the electronic components, is used by the higher stages. The actual area of the chip, however, is the sum of functional area and the area used up by the interconnect wiring. Similarly, circuit simulators use the switching delays of electronic devices to compute signal delays. However, the signal delay along a path in the circuit is the sum of the switching delays of the devices on the path and the transmission delay due to wires. Since the area of a VLSI chip determines its cost and the signal delays define its performance, it may be concluded that physical design controls the cost and performance of the chip in a big way. The correctness of the chip is also controlled by physical design. A circuit design which passes the test of a circuit simulator may be faulty after it has been packaged. This is because of *geometric design rule errors*. A geometric design rule specifies the minimum separation distance between two circuit features such as two polysilicon lines, two metal lines, etc. These design

rules must be followed to ensure the correctness of chip fabrication. Errors such as short circuits, open circuits, open channels, etc. may result if design rules are not respected.

This dissertation will focus on the physical design of electronic circuits. Since physical layout is of paramount importance in the design process, it is not surprising that most of the design automation efforts have been in this particular area. Many computer aids are available for layout – these vary from layout editors such as Magic [O+84] to automatic layout packages such as LTX [P+77].

1.5.3 Stages in Physical Design

Physical design of a large VLSI system is a complicated process and cannot be solved in a single step. It is further broken down into several stages. These stages depend on what *design style* is adopted. There are two important design styles – full custom and semicustom. Custom layout is carried out by expert human engineers using layout editors. A popular layout editor is Magic [O+84]. During a custom layout, there are no artificially placed constraints on the structure of the layout. Since the layout is carried out by a human expert, he/she can move up a level of design hierarchy if necessary to improve the design. For example, while drawing a 2-input NAND gate realization of a circuit, the engineer may realize that using multiple-input gates is more beneficial in certain parts. Alternately, the designer may decide to implement a certain portion of the circuit using 2-input NOR gates if doing so improves the performance of the circuit.

Since a human engineer can violate geometric design rules, design rule checking is an integral part of custom design. The Magic layout editor, for example, verifies the circuit for design rule violations as and when the circuit is drawn. Custom layout is an expensive process, demanding a great deal of time and resources. But since it results in a high quality realization of the chip, custom layout is often used when performance is a critical issue and the production volume is high e.g. a microprocessor or I/O processor.

Semicustom layout is a compromise between layout quality and resource requirement. Automatic layout generation tools can be designed if some constraints can be placed on the layout structure. The advantage of automatic layout generators is, of course, that they are faster than human engineers. On the negative side, layout tools do not have a global picture of the final product and cannot perform 'system level' optimization; nor can they optimize multiple objectives. When designing ASIC chips, area is a secondary issue when compared to quick turn around time. Moreover, the chip is not targeted to be manufactured in large quantities. Therefore, semicustom design styles tend to be popular with application-specific circuits.

Three semicustom layout styles are popular – *gate array*, *standard cell*, and *general cell*. A "gate array" is a two-dimensional array of identical logic elements (say NAND gates). Foundries sell gate arrays in a prefabricated state i.e. all the layers are in place except for the interconnections. In order to construct a meaningful logic function from the gate array, the user must add the necessary interconnect. This is called *personalization* of the array. Different topologies (or floor plans) are possible in a gate array chip:

1. *Row based floorplan:* logic elements are placed in rows and channels run horizontally between rows of logic.

2. *Column based floorplan:* logic elements are placed in columns and a channel runs vertically between two columns.

3. *Island based floorplan:* logic elements are like islands and channels run both horizontally and vertically.

4. *Sea-of-gates floorplan:* There are no channels, but only logic elements. Connecting wires must go over the logic elements in a different layer.

A standard cell integrated circuit is also a two dimensional array of logic elements. But the integrated circuit is not prefabricated. In this sense, they are more flexible than gate arrays. At the same time, they are not as flexible as custom chips are, due to two main reasons. First, the logic elements (or cells) of a standard cell IC

are predesigned. A library of cell designs is maintained which consists of standard circuits such as 3-input AND gate, full adder, 2-input multiplexer, and so on. The second important difference is that standard cell ICs have a row-based floorplan. All cells have the same y dimension (called height), whereas their x dimensions (widths) can vary. Channels run horizontally between rows of cells. Occasionally, wires may have to cross from one channel to another; in this case, a "feed through" is introduced by removing a cell and using the cleared space for carrying cross wires.

The *general cell* design style is the most flexible of the three semicustom design styles. It is sometimes referred to as *macro cell* design style. Unlike the gate array and standard cell designs, general cell design does not place restrictions on the sizes and shapes of cells. Furthermore, no floorplan is artificially superimposed. Therefore, general cell design comes closest to full custom design. The cells (also called *macros*, *modules*, or *blocks*) need not be predesigned and stored in libraries. Instead, a process called *module generation* may be used to synthesize general cells. Module generation is better than storing predesigned cells in libraries since cell parameters can be varied during generation. For example, the *aspect ratio* of the cell (the ratio of the cell length to its width) can be tuned during generation. Some amount of area-time tradeoff can be performed by appropriately sizing the transistors.

1.6 Subproblems in Physical Design

In this section, we consider the subproblems of VLSI physical design from the view point of computational complexity. It will be seen that these subproblems fall into the category of NP-complete optimization problems. When a problem P is said to be NP-complete, the implication is that *it is unlikely that there exists a deterministic polynomial-time algorithm that can solve* P. It is not the objective of this discussion to provide formal proofs of NP-completeness. To prove that a problem P is NP-complete, it is necessary to reduce another problem Q, which is well known to be NP-complete, to P through polynomial-time transformations. A common error in an NP-completeness proof is to reduce the problem P to a known NP-complete problem

Q and conclude that the two have similar complexity. The flaw in the argument is subtle; the transformation of problem P to Q represents one way of solving P – not the only way. All the same, transforming a problem P to an NP-complete problem Q is a fruitful exercise when it is already known that P is NP-complete. The transformation helps in understanding the problem P in terms of another NP-complete problem which has been well studied. If good heuristics exist for problem Q, the transformation helps in developing similar heuristics for problem P. With the above in mind, the succeeding sections will attempt to reduce layout problems to well known optimizations problems in graph theory.

1.6.1 Circuit Partition

It may or may not be possible to package an entire VLSI system into one integrated circuit. The factors that influence this decision are the available fabrication technology and the pin count. Cost may also be a deciding factor – fabrication costs increase with the increase in chip area and decrease in feature size. Whatever the reason, when it is not possible to include the entire circuit inside one chip, there is a need to partition the circuit into subcircuits of manageable size. There is a price to pay for subdivision of a large circuit. The interconnect wiring that connects one subcircuit to another has to be implemented outside the VLSI terrain as a pin to pin connection. This causes performance degradation by introducing transmission delays. Therefore, the objective of a partitioning procedure is to minimize "external wiring" i.e. the wiring that connects one subcircuit to another.

The simplest case of circuit partition is the *two-way circuit partition problem* (2CPP). The common form of input to the problem is a set of n modules along with information on how they are interconnected. To simplify things, it is assumed that all the connections are two-point. Such a circuit can be modeled by a graph G which has n nodes corresponding to n modules. There is an edge of weight C_{ij} between nodes i and j if there are C_{ij} connections between modules i and j. G is called a circuit graph. The objective of 2CPP is to partition the modules into two *equally sized* subsets X and Y such that minimum number of connections run across

the partition. It terms of the circuit graph, 2CPP translates to the *graph bisection* problem. Given a graph $G = (V, E)$, its bisection consists of two sets of vertices V_1 and V_2 such that

$$V_1 \bigcap V_2 = \phi$$

$$V_1 \bigcup V_2 = V$$

$$|V_1| = |V_2| \; and$$

$$\sum_{x=1}^{n} \sum_{y=1}^{n} \rho(x, y) \; is \; minimized. \tag{1.1}$$

$\rho(x, y)$ is an indicator function defined as follows.

$$\rho(x, y) = \begin{cases} 1 & \text{if } (x, y) \text{ is an edge in } E,\ x \in V_1 \text{ and } y \in V_2 \\ 0 & \text{otherwise} \end{cases}$$

The number of different ways of bisecting a graph on n nodes is given by

$$s = \frac{n!}{2(\frac{n}{2}!)^2} \tag{1.2}$$

Since n is large in a VLSI circuit graph, Stirling's approximation may be used to obtain

$$s \approx \frac{2^n}{\sqrt{2\pi n}}. \tag{1.3}$$

Thus, it is impractical to solve the two-way partition problem by an enumerative search.

1.6.2 Floorplanning

Physical design of an integrated circuit is accomplished in several steps. These steps depend on the chosen design style. For example, general cell design begins by deciding a floorplan for the chip. During the floorplanning stage, the circuit is viewed at the block level, where the functional blocks are components such as memory units, multiplexers, control units, and arithmetic/logic units. As explained

earlier in this section, these functional blocks are either predesigned or are generated through the process of *module generation.* Usually, the blocks are rectangular in shape, although L-shaped blocks are also possible. Given a set of rectangular blocks, the aim of floorplanning is to position them and orient them. A floorplan of an IC is analogous to an architect's drawing of a building showing the layout of various rooms and corridors. There are many possible floorplans even for a small number of rectangles. The object of a floorplanner is to find that topology which minimizes the area occupied by the *bounding rectangle* i.e. a rectangle that surrounds all the cells.

To appreciate the complexity of the floor planning problem, consider an instance of problem where the solution is known beforehand. Starting with a rectangle, it is recursively sliced into smaller rectangles as shown in Figure 1.3(a). A floorplanner must do the reverse i.e. gather the rectangular pieces into the original rectangle. Figure 1.3(b) shows a non-optimal solution which the floorplanner may generate for the same problem. A moment's reflection shows that obtaining the optimum floorplan is analogous to solving a jigsaw puzzle consisting of rectangular pieces. Unfortunately, the answer to the jigsaw puzzle is not known beforehand. Every possible way of positioning the rectangles is a solution to the problem of floorplanning. As a result, there is a combinatorial explosion of solution space when the number of rectangles n is increased. In fact, it is a non-trivial problem to count all unique solutions to the floorplanning problem. It is known that for large n, the number of possible solutions will grow exponentially in n [Jay87]. As a result, an enumerative technique to obtain the best floorplan is infeasible.

1.6.3 Circuit Placement

The floorplanning stage is not necessary for standard cell or gate array design, since the topology is fixed for both these design styles. Given a topology, the *logic placement* problem is to assign positions to the logic blocks. For example, in the standard cell design style, the position of a cell is determined by the row in which the cell is placed and its relative position with respect to other cells assigned to the same

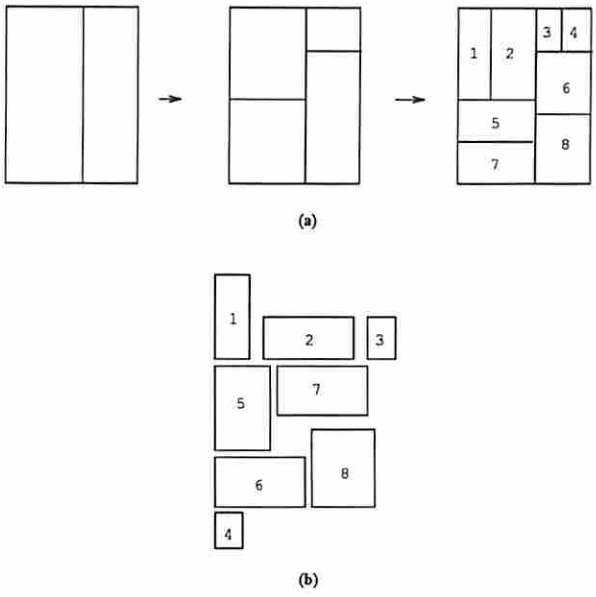

Figure 1.3: The Floorplanning problem. (a) shows how a slicing floorplan is obtained. (b) shows a possible floorplan to the same problem.

row. The aim of a placement procedure is to derive a placement that minimizes the amount of wiring.

Let $\mathcal{M} = \{M_1, M_2, \cdots, M_n\}$ denote a set of n modules and $C = [C_{ij}]$ denote the *connectivity matrix*. The element C_{ij} is the number of wiring connections between modules M_i and M_j. If the layout surface is a grid of dimension $p \times q$, there are $p.q$ "slots" into which the logic modules can be placed. Without loss of generality, it may be assumed that $p.q = n$. Let $\mathcal{S} = \{S_1, S_2, \cdots, S_n\}$ be the set of slots. Representing slots as point masses, let $\mathcal{D}_{ij}$ be the distance between slots S_i and S_j [HK72]. The form of the distance function depends on which routing model is adopted. For example, in the Manhattan routing model, wires run either horizontally or vertically and therefore, $\mathcal{D}_{ij}$ is the rectilinear distance between the slots S_i and S_j.

The placement problem is to arrive at a bijection $\mathcal{P}$

$$\mathcal{P} : \mathcal{M} \mapsto \mathcal{S} \tag{1.4}$$

which minimizes some objective function $\Theta(\mathcal{P})$. The most common objective functions are the total wire length and the total area of placement. Since the minimization of wire length indirectly minimizes the total area, it is usual to consider aggregate wire length $\omega(\mathcal{P})$ as the cost function. Consider two modules i and j, assigned to slots $\mathcal{P}_i$ and $\mathcal{P}_j$ respectively; the distance between the modules is $\mathcal{D}_{\mathcal{P}_i\mathcal{P}_j}$. Thus, the length of the wiring required to connect modules i and j is $C_{ij}\mathcal{D}_{\mathcal{P}_i\mathcal{P}_j}$. Hence,

$$\omega(\mathcal{P}) = \sum_{i=1}^{n}\sum_{j=1}^{i} C_{ij}\mathcal{D}_{\mathcal{P}_i\mathcal{P}_j} \tag{1.5}$$

Formulated as above, module placement is an instance of the *Quadratic Assignment problem*, well known to be NP-complete . To appreciate how big the solution space is for the above placement problem, observe that there are $n!$ different bijections $\mathcal{P}$. When n is large, Stirling's approximation yields

$$n! \approx n^n e^{-n}\sqrt{2\pi n} \tag{1.6}$$

In other words, the number of placements is exponential in n and an enumerative algorithm for placement requires exponential time to find the best placement. Since a typical VLSI chip contains 5000 cells or more, brute force enumeration is out of question.

1.6.4 Circuit Routing

After the cells have been positioned, the interconnections among cells are established using metallic wires. This is called *routing*. The principal aim of a router is to ensure that all the connections are correctly established. When gate array design style is used, the router does not have much flexibility. There are a fixed number of routing channels, and each channel has a fixed number of routing tracks. As a result, the *routability* of the chip is in question. With standard cell and general cell designs, the router has the option of trading chip area for routability. Thus, more tracks can be created if necessary by increasing the area of the chip. Routing procedures aim at 100% routability, at the same time trying to optimize on chip area. It may not

be possible to directly minimize the chip area; instead, routing algorithms try to find as short a route as possible for each connection. A good router also attempts to minimize the number of *vias*, since vias increase the resistance of the signal path and hence contribute to signal delays. Vias reduce the reliability of the circuit by introducing potential "open contacts."

The input to a router is a *netlist* i.e. a set of electrical nets. In turn, each net is a set of electrically equivalent *pins*. The aim of a routing program is to find a path to wire each net, subject to the given wiring model. A wiring model specifies the number of layers available for routing, what wiring directions are permitted in each layer, and what types of wires are permitted on each layer. For example, the Manhattan wiring model assumes two layers. Polysilicon wires are restricted to layer 1 and metal wires are restricted to layer 2. Only horizontal wires are permitted in layer 1 and only vertical wires in layer 2. If a wire route has to bend at some point, a change in direction is achieved by use of a via at that point.

As mentioned earlier, in addition to achieving 100% routing, a router also tries to optimize one or more objective functions such as the number of *vias* and the total wire length. It is hard to define a good objective function for layout problems because of conflicting requirements. For example, if total wiring length is defined as the objective function during placement, the resulting solution may have "hot spots." If a cut-line is imagined passing through the chip, the number of nets that cross the cut-line is known as the wiring density at the cut-line. The wiring density d is a lower bound on the number of wiring tracks required at the cut-line in order to successfully route all the nets that cross the cut-line. A point at which the wiring density exceeds the channel capacity (maximum number of tracks in the channel) is known as a "hot spot." If the design style is gate array, then the presence of a hot spot clearly implies that 100% routing cannot be achieved. Standard cell and general cell design styles are more flexible and allow 100% routing to be achieved by eliminating hot spots. However, eliminating a hot spot can only be done by increasing the channel capacity at the spot, which can only be done by widening the channel. Thus, the elimination of hot spots costs chip area.

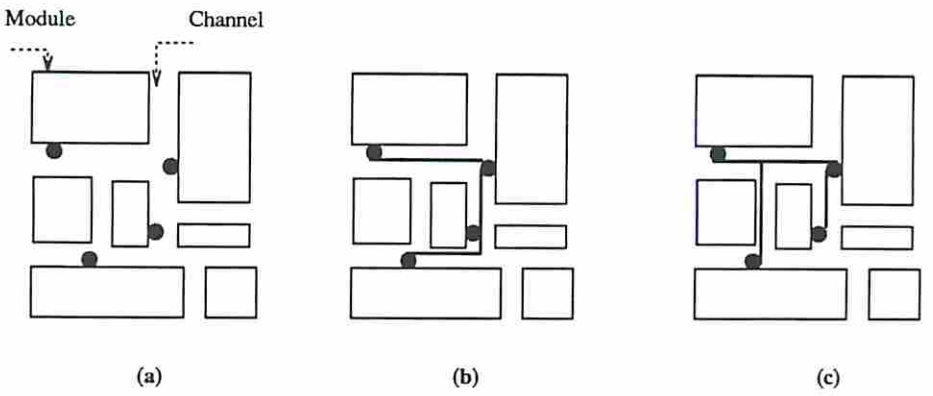

Figure 1.4: Two ways to route a 4-pin net. Both the routes are implemented as rectilinear Steiner trees.

1.6.4.1 Lee Routing

A net is a collection of pins that are electrically equivalent. For the moment, assume that each net has exactly two pins. The best way to route a net is to connect the two end points of the net by the shortest possible wire; a shortest-path algorithm can be used to find such a route. Lee's algorithm [Lee61] is one such algorithm. If there are N nets in the netlist, consider applying Lee's algorithm to the nets in some order $\pi_1, \pi_2, \cdots, \pi_N$. After routing net π_i, the area occupied by the net cannot be used by another net. Thus, nets $\pi_1, \pi_2, \cdots, \pi_i$ must be treated as obstacles when routing the subsequent nets $\pi_{i+1}, \pi_{i+2}, \cdots, \pi_N$. It is not hard to realize that the order in which nets are routed is crucial to the success of the router. Thus, a certain ordering $\pi = \pi_1, \pi_2, \cdots, \pi_N$ of the nets may be infeasible i.e. it is impossible to route nets $\pi_{i+1}, \cdots, \pi_N$ after the nets $\pi_1, \pi_2, \cdots, \pi_i$ have been routed. Since π is a permutation of $1, 2, \cdots, N$, there are $N!$ possible ways to order the nets. To date, there does not exist a polynomial-time algorithm to find a *feasible* ordering π.

1.6.4.2 Global Routing

A global router is so called because it looks at all the nets at once and determines rough paths for each net, such that routability is ensured. A rough path for a net specifies only the channels through which the net must be routed – it does not assign routing tracks to the net. A 4-point net is shown in Figure 1.4(a). Two different

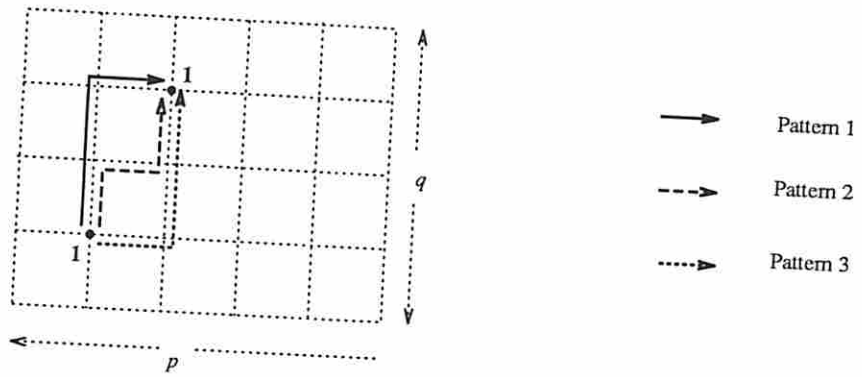

Figure 1.5: Integer Programming Approach to Global Routing. The figure illustrates three shortest-path routes for net 1.

global routes for this net are shown in Figures 1.4(b) and 1.4(c). The example illustrates that there is more than one way to route a single net. A global router considers all the minimum-length routes to implement a net. When a net has only two pins, it is possible to use a "shortest path" algorithm to find the best global route for the net. In general, for a p-point net, the minimum-length routes correspond to minimum-length rectilinear Steiner trees on p points [Han66]. Polynomial-time procedures are not known for constructing rectilinear Steiner trees for arbitrary values of p. Hanan has devised simple geometric procedures to construct rectilinear Steiner trees for $p = 3, 4, 5$.

For the present discussion, assume that $p = 2$. Superimpose a grid on the routing area as shown in Figure 1.5. Consider the possible ways to route net 1 shown in the figure. There are exactly three of these, as shown in the diagram, if routes are restricted to shortest path routes. In general, if the horizontal span of a net is m and the vertical span is n, then there are $\binom{m+n}{m}$ ways of completing the net. Let N_i be the number of routes for pattern i, $1 \leq i \leq nomathmcnm + nm$; then the following equation is true.

(1.7)

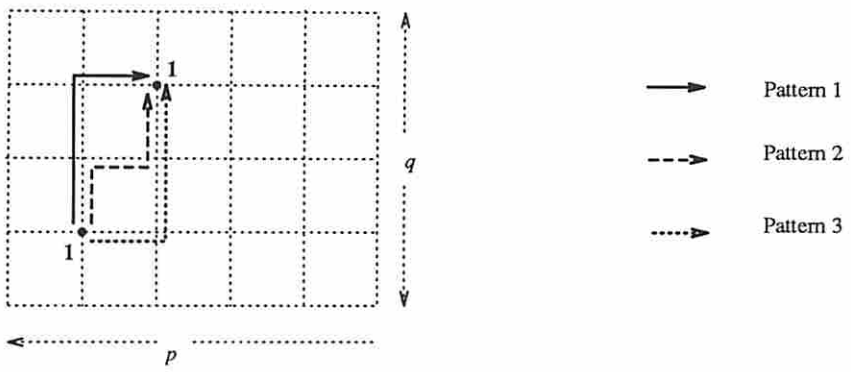

Figure 1.5: Integer Programming Approach to Global Routing. The figure illustrates three shortest-path routes for net 1.

global routes for this net are shown in Figures 1.4(b) and 1.4(c). The example illustrates that there is more than one way to route a single net. A global router considers all the minimum-length routes to implement a net. When a net has only two pins, it is possible to use a "shortest path" algorithm to find the best global route for the net. In general, for a p-point net, the minimum-length routes correspond to minimum-length rectilinear Steiner trees on p points [Han66]. Polynomial-time procedures are not known for constructing rectilinear Steiner trees for arbitrary values of p. Hanan has devised simple geometric procedures to construct rectilinear Steiner trees for $p = 3, 4, 5$.

For the present discussion, assume that $p = 2$. Superimpose a grid on the routing area as shown in Figure 1.5. Consider the possible ways to route net 1 shown in the diagram. There are exactly three of these, as shown in the diagram, if routes are restricted to shortest-path routes. In general, if the horizontal span of a net is m and the vertical span of the net is n, then there are $\binom{m+n}{m}$ ways of completing the net. Let $y_i = 1$ if the given net is routed using pattern i, $1 \leq i \leq nomathmcnm + nm$; otherwise let $y_i = 0$. For net 1 in the example, the following equation is true.

$$y_1 + y_2 + y_3 \leq 1 \tag{1.7}$$

The sum is equal to 1 if it is possible to route net 1 using one of the three patterns; the sum is 0 if the net is unroutable. Such inequalities can be written down for each

of the nets in the circuit. Then global routing reduces to solving for the y's, subject to some constraints. Constraints arise since wire congestion needs to be avoided. Each segment in the grid can be assigned a weight c_i which indicates the capacity of segment i. Such constraints arise in gate array chips, where the chip size is fixed and channels have limited capacity. If the grid is of dimensions $p \times q$, then there are $p(q-1)+q(p-1)$ segments in the grid; suppose that these segments are numbered $1, 2, \cdots$, in some order. Now define a binary variable a_{ij} as follows. a_{ij} is set to 1 if and only if pattern y_j for some net uses the segment i. Then the sum $\sum a_{ij}y_j$ counts the number of nets that pass through segment i. If the capacity of the segment i is c_i, then

$$\sum a_{ij}y_j \leq c_i \tag{1.8}$$

so that channel capacities are not exceeded.

The goal of the global router is to route as many nets as possible. Thus, the objective is to maximize the sum $\sum y_j$. With two more notations, the global routing problem can be summarized as a 0-1 integer linear programming problem. Let N_k indicate the set of patterns associated with net k. Let m be the number of nets and n be the total number of patterns.

$$\begin{aligned}
\text{maximize} \quad & \textstyle\sum_{j=1}^{n} y_j \quad j = 1, 2, \cdots, n \\
\text{subject to} \quad & \textstyle\sum_{j \in N_k} y_j \leq 1 \quad k = 1, 2, \cdots, m, \\
\text{and} \quad & \textstyle\sum a_{ij}\, y_j \leq c_i \; i = 1, 2, \cdots, 2pq - p - q \\
\text{where} \quad & y_j \in \{0,1\}.
\end{aligned}$$

In the above form, global routing is an instance of the 0-1 linear programming problem, well known to be NP-complete. The above formulation has been adapted from Hu and Shing's work in [HS85].

1.6.4.3 Channel Routing

The complexity of channel routing depends on the wiring model assumed. Consider the Manhattan wiring model, which permits two layers, metal and poly. Horizontal wire routes are implemented on metal and vertical wire routes in polysilicon. The input to a channel router is a set of N two-point nets. Figure 1.6 shows a channel

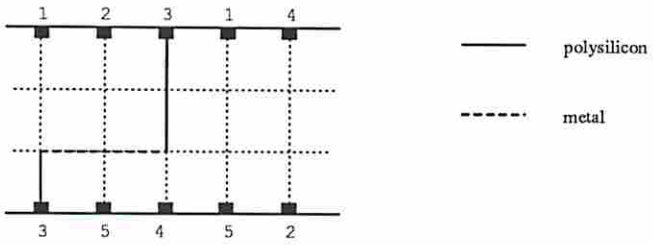

Figure 1.6: A channel with two-point nets

with five two-point nets, one of which is shown wired. The objective of the router is to assign the minimum number of routing tracks so that all the nets can be realized [YK82]. The *interval* or *span* of a two-point net (A, B) is $[x_1, x_2]$ where x_1 is the coordinate of the leftmost point of net (A, B) and x_2 is the coordinate of the rightmost point. Two nets cannot be placed on the same track if their intervals intersect. This is known as the *horizontal constraint* between the two nets. In general, when $k \geq 2$ nets are involved, they can be placed on the same track if the every pair of intervals does not intersect.

The above ideas can be captured by creating a simple graph model. This graph, known as an *horizontal constraint graph*, has one node to represent each interval (or net). An edge exists between nodes i and j if and only if the intervals i and j intersect. An *independent set* in the horizontal constraint graph (HCG) corresponds to a set of nodes that can be placed on a single track without horizontal conflicts. Alternately, consider the complement of the HCG, indicated by HCG′. The complement graph has the same number of nodes. An edge is added between nodes i and j of HCG′ if and only if the intervals i and j do *not* intersect. Both HCG and HCG′ belong to a special class of graphs known as *interval graphs*. A set of nets which can be placed on the same track will form a *clique* in HCG′. In terms of HCG′, the channel routing problem translates to the problem of finding the *minimum sized clique cover*. Finding a minimum clique cover is an NP-complete for general graphs [GJ79]. However, for interval graphs, the problem is solvable in polynomial time [Gol80]. Unfortunately, matters are complicated due to what are known as "vertical constraints."

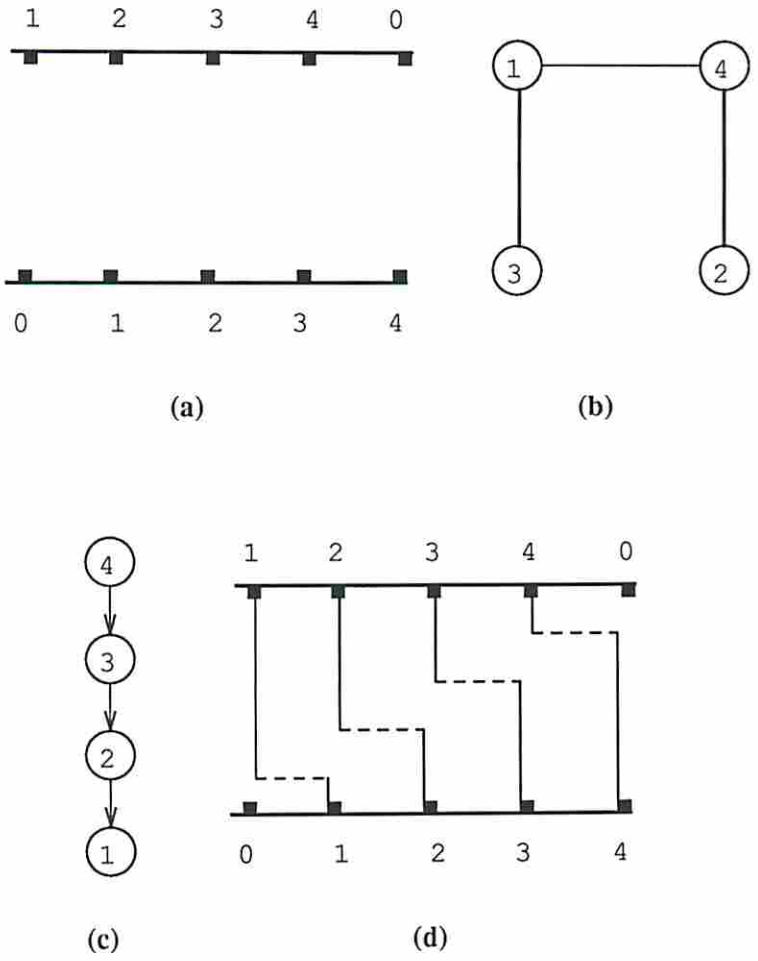

Figure 1.7: Horizontal and Vertical Constraints in Two-layer Channel Routing.

In the above discussion, only the horizontal intervals of nets were considered; the vertical portions of wiring required to complete the routing were ignored. Consider the channel shown in Figure 1.7(a). There are four nets in this example, marked $1, 2, 3$ and 4. The complement horizontal constraint graph for the example is shown in Figure 1.7(b). A clique cover of size 2 is {13,24} seemingly leads to a two track solution – with nets 1, 3 in the first track and nets $2, 4$ in the second track. However such is not the case. An attempt to complete the vertical wiring will lead to the following disappointing discovery – the vertical segment which connects net 2 to the bottom overlaps with the vertical segment which connects net 3 to the top of the channel. This is known as a vertical constraint between nets 2 and 3. Vertical constraints can be determined by looking at each column of the channel. For example, in the second column, pin 2 appears on the top and pin 1 on the bottom. This means that net 2 must be placed above net 1 in order to avoid their vertical portions from overlapping. Vertical constraints can be modeled by constructing a directed graph where each node corresponds to a net and a directed edge exists from node i to node j if net i must be placed above j due to a vertical constraint. The vertical constraint graph (VCG) for the example is shown in Figure 1.7(c). Due to vertical constraints, the channel in the example permits a 4 track solution shown in Figure 1.7(d).

Thus channel routing is more complex than the clique cover problem on an interval graph. The cliques of the cover must satisfy an additional property – no two nodes that belong to a clique must lie on a directed path in the vertical constraint graph. This constrained clique cover problem is NP-complete.

1.7 Organization of this dissertation

Chapter 2

Computations that arise as part of circuit design fall into three categories – numerical computation, symbolic computation and combinatorial optimization. Over the past two decades, a vast number of techniques have been developed for solving numerical

problems on parallel computers. The same cannot be said of symbolic computations and optimization problems. Most problems that arise in physical design of an integrated circuit are hard combinatorial optimization problems. Therefore, it is necessary to understand on a general plane how parallel processing can help solve NP-complete problems. Chapter 2 will address this issue. The concept of *parallel complexity* of a problem is introduced. Parallel heuristic algorithms and parallel approximation algorithms are presented as parallel optimization techniques. Parallel randomized algorithms are described for combinatorial search problems. These algorithms offer two advantages – better quality solutions and superlinear speedup.

Chapter 3

Chapter 3 considers parallel algorithms for the logic partition problem. Kernighan and Lin's heuristic algorithm for two-way circuit partition is known to generate high quality solutions. A PRAM version of this heuristic algorithm is presented. The PRAM algorithm is then extended to execute on two parallel machines – a SIMD computer known as the Orthogonal Array and the Alliant FX/80 supercomputer, which is a shared memory multiprocessor.

Chapter 4

In Chapter 4, parallel algorithms are described for the circuit placement problem. First a taxonomy is presented to classify the different algorithms available for parallel placement. Parallel iterative improvement, parallel min-cut placement, parallel placement by the divide-and-conquer paradigm, parallel linear placement based on eigensystem approach, and parallel placement by simulated annealing are described. In each case, several special-purpose and general-purpose parallel systems are described to solve the placement problem.

Chapter 5

Chapter 5 looks at parallel routing algorithms. A classification of existing parallel routers is presented. A parallel algorithm is described for hierarchical global routing. This approach divides the routing problem into two steps – global routing and channel routing. The algorithm is targeted for the Orthogonal Array SIMD architecture. Next, an MIMD algorithm is described for channel routing. The technique exploits the large-grain parallelism in the channel routing problem and schedules multiple channels for concurrent routing.

Chapter 6

Subproblems in physical design can be conveniently modeled using graphs. Each of the subproblems can then be stated as an optimization problem on the corresponding graph. For example, the channel routing problem can be reformulated as a clique cover problem on an interval graph. Graph optimization problems such as *maximum clique*, *minimum clique cover*, and *minimum coloring* are NP-complete for general graphs. However, for a class of graphs called "perfect graphs," these optimization problems are solvable in polynomial time. A perfect graph has some additional structure to it; in particular, the clique number of a perfect graph is equal to its chromatic number. Interval graphs, permutation graphs, triangulated graphs and transitive graphs are all examples of perfect graphs. Under restricted conditions, several physical design problems can be modeled as optimization problems on perfect graphs. Chapter 6 gives a number of examples of this phenomenon. Apart from physical design, several problems in *register level synthesis* can also be formulated in terms of perfect graphs. These are also considered in Chapter 6. Specifically, the chapter examines applications of interval graph coloring and permutation graph coloring. Parallel algorithms are discussed for both of these problems.

Chapter 7

Chapter 7 summarizes the contents of the dissertation by defining the concept of a *parallel framework* for electronic design automation. Developing a parallel framework for a design problem consists of several subtasks – identifying the core computations in the problem, analyzing the core computations for parallelism, choosing a suitable computing platform based on price/performance considerations, mapping the design algorithm to the target architecture, developing an abstract parallel algorithm and actual implementation.

Applying parallel processing to electronic design automation is a relatively new field of research and there are a number of opportunities for further work in this area. Several directions are indicated in Chapter 7 for future research. Perfect graphs have begun to attract the attention of researchers working in several areas such as operating systems, operations research and computer-aided design. There is a great deal to be gained by reconsidering the optimization problems that arise in design automation and studying their complexity under restricted conditions. Developing parallel algorithms for graph optimization problems is by itself a fruitful area of research. It is also of great interest to consider new approaches for solving optimization problems. For example, *artificial neural networks* have been recently applied to solve the traveling salesperson problem. Expert systems have been effectively used to solve difficult problems such as switchbox routing. Search techniques such as the A* algorithm, which originated from research in Artificial Intelligence, have been applied to PLA folding and gate matrix layout. Branch-and-bound algorithms have been used for circuit placement and logic minimization. Backtracking has been used in test generation and high level synthesis. All these new paradigms have the potential to yield high quality solutions to optimization problems. But their common drawback is their compute-intensive nature. Parallel processing holds great promise in overcoming this drawback.

1.8 Research Contribution

This thesis presents parallel algorithms for VLSI layout design. Implementations of these algorithms on Intel iPSC/2 and Alliant FX/80 are described. This section describes the research contributions in detail.

Combinatorial Search

General-purpose parallel techniques have been described in Chapter 2 to solve combinatorial optimization problems of the NP-complete type. These techniques are based on parallel randomization. Multiple processors are used to simultaneously explore the exponentially large state space of a discrete combinatorial optimization problem; randomization is used to ensure that two different processors search through different regions of the state space. These techniques have the merit that they have the potential to generate significantly better quality solutions compared to a sequential optimization algorithm. It is easy to implement these techniques on a multiprocessor architecture; in fact, an existing sequential implementation can be ported to a parallel machine without much effort when incorporating these search techniques. The parallel search techniques have the potential to yield *superlinear speedup*.

The search methods have been successfully tested for two optimization problems, the traveling salesperson problem and the two-way circuit partition problem. The Intel iPSC/2 and the Alliant FX/80 have been used for implementation.

Circuit Partition

A parallel algorithm for two-way circuit partition is presented in Chapter 3. A PRAM algorithm is presented first and it is extended for operation on an SIMD array processor known as *Orthogonal Array*. The asymptotic performance of the algorithm has been derived using theoretical means. The algorithm has been implemented on an Alliant FX/80 multiprocessor and has been tested against benchmark problems.

Circuit Placement

Several parallel algorithms have been developed for circuit placement (Chapter 4). A parallel version of the Min-cut placement algorithm has been developed for the Orthogonal Array parallel processor. A parallel algorithm is reported for placement by iterative improvement; this algorithm is suitable for implementation on a massively parallel SIMD computer. A new technique for placement based on the divide-and-conquer paradigm is proposed; this algorithm is inherently parallel and its parallelization is described on two platforms, a shared memory SIMD computer (Orthogonal Array) and a distributed memory MIMD computer (Hypercube). A placement algorithm based on a numerical optimization technique (of identifying a suitable eigenvector of a positive definite matrix) is reported; this algorithm has been implemented on an Alliant FX/80 multiprocessor. All the parallel algorithms reported above are processor-optimal and their asymptotic analysis has been carried out.

Circuit Routing

A parallel algorithm for global routing, based on the "cut and paste" technique is reported in Chapter 5. The algorithm is intended for execution on an Orthogonal array; the theoretical analysis of the algorithm's performance is carried out. A parallel algorithm for channel routing is reported which schedules multiple routing channels for simultaneous wiring. This algorithm is intended for an MIMD platform and its performance is predicted for a specific class of floor plans known as slicing structures. A three-layer channel routing algorithm based on the theory of interval graphs is reported in Chapter 6; this algorithm is intended for execution on an EREW PRAM.

Optimization Problems on Perfect Graphs

Interval graphs, permutation graphs, triangulated graphs, transitive graphs etc. are examples of *perfect graphs*; these are interesting since the otherwise hard optimization problems such as *maximum clique*, *optimum node coloring*, *maximum independent set*, etc. are solvable in polynomial time for perfect

graphs. In Chapter 6, a number of applications in the field of design automation are modeled as optimization problems on interval graphs and permutation graphs. Parallel algorithms are presented for interval graph coloring, permutation graph coloring, finding a maximum clique in an interval graph, and finding a maximum independent set in a permutation graph. The interval graph coloring algorithm has been implemented on an Intel iPSC/2.

Chapter 2

Parallel Processing and NP-completeness

Computational problems that arise in VLSI design may be grouped into three categories – numerical computations, symbolic computations, and optimization problems. Examples of numerical computation are matrix and vector operations, solving systems of linear equations, differential equations, partial differential equations, and so on. These problems come up during device-level circuit simulation. Typical symbolic computations include sorting, text processing, database operations, etc. Symbolic computations arise in applications such as design rule verification and CAD databases. Examples of optimization problems include linear programming, non-linear programming, integer programming, graph optimization problems such as optimum node coloring, maximum clique, maximum independent set, minimum clique cover, and so on. Optimization problems come up in every *synthesis* stage of hierarchical VLSI design – functional design, logic design, circuit design and physical design.

Parallel processing has been applied extensively to numerical applications over the past two decades. Symbolic computations have begun to attract parallel implementations much more recently. The study of parallel optimization techniques is still in its infancy.

As Section 1.6 has brought out, physical design involves a number of optimization problems. Furthermore, these optimization problems are computationally hard. Few studies have been conducted to understand the relationship between parallel processing and NP-completeness. This chapter addresses the issue by compiling together

the various parallel approaches to NP-complete problems. We also propose search techniques based on parallel randomization for solving hard optimization problems. Several processors are used to concurrently explore the state space of a combinatorial optimization problem. Randomization is used to ensure that two different processors search through different regions of the search space. The traveling salesperson problem is used as an example to illustrate our techniques. The results of applying our parallel search methods to large instances of these problems are described. The target machines used for these parallel implementations are the Intel iPSC/2 hypercube and the Alliant FX/80. The results obtained are encouraging both in terms of the solution quality and speedup. Our techniques are easy to implement and are easily extended to other optimization problems of design automation.

2.1 Parallel Optimization Techniques

Optimization problems are further classified into "hard" and "easy" optimization problems. A problem is said to be computationally easy if there exists a deterministic, polynomial-time algorithm to solve the problem. A hard optimization problem is one which belongs to the NP-complete class [GJ79]. Although no formal proof is available, it is strongly believed that a deterministic algorithm cannot solve an NP-complete problem in polynomial time. An example of an easy optimization problem is the construction of a minimum spanning tree (MST) of a graph. Using Prim's algorithm, $O(|V|^2)$ time suffices to find an MST of a graph with $|V|$ nodes [AHU83]. Parallel algorithms for easy optimization problems have been studied in the literature [Akl89, GR87]. Graph bisection, bin packing, traveling salesperson, graph coloring, circuit placement, and channel routing are all examples of problems that are known to be computationally hard. Section 1.6 has reviewed several hard optimization problems which arise in VLSI design. By employing parallel processing, it is not possible to change the computational complexity of an optimization problem. An exponential number of processors will be required to get exact solutions to NP-complete problems in polynomial time! Even for moderate problem sizes, using

an exponential number of processors is out of question. How can parallel processing help us attack NP-completeness? The most common approaches are discussed below.

2.1.1 Parallel Heuristic Algorithms

Many heuristic algorithms are known to perform exceedingly well in practice, although they can generate arbitrarily bad solutions in the worst case. Good examples for this phenomenon are the Kernighan-Lin algorithm for graph partition [KS70], Min-cut procedure for logic placement [Bre77], the "two-opt" procedure for the traveling salesperson problem [SDK83]. The success of these 'rule-of-the-thumb' procedures can be explained by probabilistic analysis, where it is assumed that problem instances are drawn from some probability distribution. For example, while analyzing the traveling salesperson problem, it is reasonable to assume that the locations of the cities are drawn from a uniform distribution. Probabilistic analysis was pioneered by Karp, who analyzed his divide-and-conquer heuristic for the traveling salesperson problem [Kar77]. Statistical methods to analyze heuristics are given in [L+85]. In the field of design automation, the popular technique to evaluate a heuristic algorithm is to test it against a set of benchmark problems. Based on benchmark evaluation, several heuristic algorithms for design problems have come to become stable algorithms – Kernighan and Lin's partition algorithm, Min-cut placement algorithm, force-directed placement, and iterative improvement to mention a few. In this dissertation, parallel algorithms will be examined for several stable heuristic algorithms.

2.1.2 Parallel Approximation Algorithms

An approximation algorithm for an optimization is a procedure which generates a near-optimal solution in polynomial time. An approximation differs from a heuristic in that it is possible to quantify the concept of near-optimality of an approximate solution. Sahni [SG76], Karp [Kar77], Johnson [Joh74] and others who pioneered

research in *approximation analysis.* Let P be a maximization problem and C^* be the maximum value of the cost function. Let C be the cost of an approximate solution generated by some algorithm A. The algorithm A is an *ϵ-approximation algorithm* if

$$\frac{C}{C^*} \geq 1 - \epsilon \textit{ for any } \epsilon,\ 0 < \epsilon < 1,\ \textit{for all instances of the problem.} \qquad (2.1)$$

ϵ-approximation algorithms are known for classic optimization problems such as the traveling salesperson, 0-1 Knapsack, and bin-packing (see [Gop86, Kar77, PS82]). The running time of an ϵ-approximation scheme is a polynomial in (n/ϵ), where n is the size of the problem. As ϵ approaches 0, the algorithm becomes more and more expensive in CPU-time.

Example 2.1 :

Given n points on a plane, the problem of constructing a minimal-length rectilinear Steiner tree (RST) passing through these points is NP-complete. On the other hand, constructing a minimum-length spanning tree (MST) to pass through the n points is an easy optimization problem. The n points are considered as n nodes in a graph. The edges in the graph are straight lines joining each pair of points. The weight on an edge (i, j) is the Euclidean distance between the two points. When the distance measure is Manhattan in stead of Euclidean, the MST is also known as a rectilinear MST, or RMST. Hwang [Hwa79] has shown that an RMST is a good approximate solution to the Steiner tree problem. In fact, he has proved that

$$\frac{length(RMST)}{length(RST)} \leq 1.5 \qquad (2.2)$$

To illustrate, consider the three points A, B, C shown in Figure 2.1. Figure 2.1(a) shows a minimum spanning tree on A,B, and C. The length of this MST is $\sqrt{5} + \sqrt{8}$. In Figure 2.1(b), an RMST is depicted. The length of the RMST is 7. The minimum-cost rectilinear Steiner tree on A,B, and C is

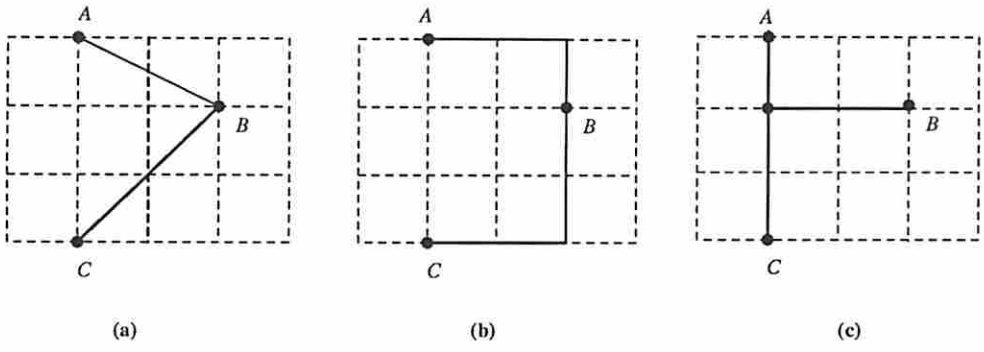

Figure 2.1: Approximating a Rectilinear Steiner Tree by a Rectilinear Minimum Spanning tree.

shown in Figure 2.1(c) and has a length of 5. Therefore the RMST is only a factor of 1.4 larger than the best RST. Hwang has designed an $O(n \log n)$ algorithm to construct a rectilinear MST through n points. This algorithm is an approximation algorithm for the Steiner tree problem.

Example 2.2 :

Consider Karp's partitioning algorithm for the traveling salesperson problem [Kar77]. Given n cities, the algorithm subdivides the set of cities into smaller groups, each of which has no more than t cities. t is a user-specified parameter and is small enough to allow an optimal tour to be constructed within each group. The sub-tours are then pasted together such that the final circuit is Hamiltonian. Karp showed that the worst case error of the algorithm is given by

$$T - T^* \leq O(\sqrt{n/t}) \tag{2.3}$$

where T is the length of the tour generated by the heuristic and T^* is the optimal tour of n cities. The running time of the algorithm is given by $T(n)$,

$$T(n) < 2\frac{n-1}{t-1}Dd^t + O(n \log n). \tag{2.4}$$

where D and d are positive constants larger than unity. The result is very powerful since it holds for all values of t, $1 \leq t \leq n$. In other words, the value of ϵ is controllable by varying t. However, when t approaches n, the algorithm degenerates to an exponential-time algorithm.

2.1.3 Polynomial-time solutions to Layout Problems

Mead and Conway suggested that advanced VLSI technology will bring radical changes to the way we solve problems. The conventional way to cope up with NP-completeness has been to design heuristics to solve intractable problems approximately in polynomial time. An aggressive and unconventional approach is to take parallel processing to its extreme and use an exponential number of processors to tame an intractable problem. For instance, consider the placement of n modules as formulated in Section 1.6.3; the search space of this problem consists of an exponential number of placements (Equation 1.6). How can this search space be explored in parallel? The placement of n modules can be visualized as a tree computation, where the slot for module i is selected at level i of the tree. Since there are n choices for module 1, the tree has n nodes at level 1. There are $n \cdot (n-1)$ nodes at level 2, and, in general, $n \cdot (n-1) \cdots (n-k+1)$ nodes at level k. There are $n!$ leaf nodes, each of which corresponds to a final placement. Figure 2.2 illustrates such a tree for $n = 3$. Associated with each leaf node is the cost of the corresponding placement. Theoretically, a systematic search of the solution space is possible and involve a worst case total computation of

$$T(n) = \sum_{k=1}^{n} (n)_k \tag{2.5}$$

where $(n)_k$ represents $n \cdot (n-1) \cdots (n-k+1)$. $T(n)$ may be simplified as

$$T(n) = n!(1 + \frac{1}{2} + \frac{1}{2 \cdot 3} + \cdots) = O(n!) \tag{2.6}$$

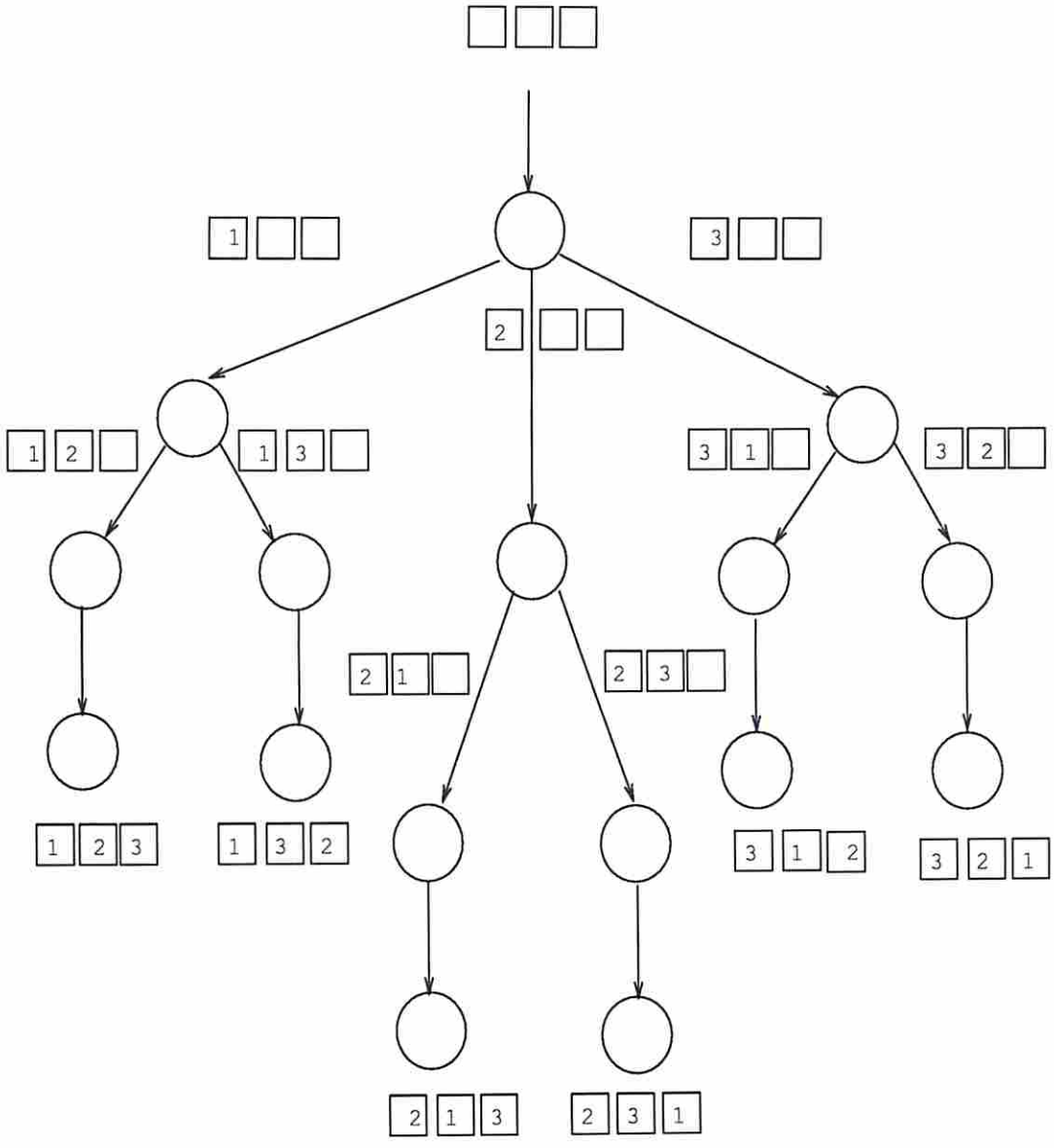

Figure 2.2: Decision Tree for a 3-module Placement

The search tree can be pruned by employing the *Branch and Bound* technique, which stops expanding a non-leaf node η as soon as it is evident that η gives inferior solutions compared to its siblings at the same level. The worst case complexity of the Branch-and-Bound technique, however, remains an exponential function of the problem size N. Assuming that there is an unlimited supply of processing elements, assign a single processor to each node in the search tree. Such a massively parallel machine can find an optimal placement. A *partial placement*, in which k modules have been assigned slot positions, can be associated with the level k of the search tree. The $k+1$st module may be placed in any of the remaining $(n-k)$ slots. Not all these possibilities need be explored if there are efficient ways to discard one partial placement in favor of another based on cost considerations. When total wire length is used as the cost function, observe that it can be written as a quadratic form (Equation 1.5). A simple lower bound for the total wire length is obtained as follows.

1. Sort the connectivities in the increasing order and form a vector c.
2. Sort the distance measures in the decreasing order and form vector d.
3. The inner product of vectors c and d gives a lower bound on wire length.

Using this idea, Gilmore has developed a way to compute lower bounds on wire lengths for partial placements (see [HK72]). Figure 2.3 describes the operation of the tree machine. Specifically, it shows the process executed by each processor in the tree.

Lemma 1 *Using $O(n!)$ processors, the placement of n modules needs $\Omega(n \log n)$ time.*

Proof : Computing the Gilmore bound at level k of the search tree involves a sorting operation and a dot product computation on a set of $(n-k+1)$ elements. There are additional sources of parallelism in the placement procedure. If parallel sorting is used within each processor, then the total computation time for computing the Gilmore lower bound is bounded below by $\log_2 n$. Since there are at most n levels

```
class processor;
begin
      i = mylevel();              /* Find out my level in tree */
      if (i ≠ 1)     /* I am not a root */
      begin
            receive (P, S, F);   /* Receive partial placement P from parent */
            if (F = TRUE) terminate; /* Do not expand, solution known to be inferior */
            j = myposition() /* I am the jth child of my parent */
            P(i) = S_j; /* Place the jth module in position i */
            S = S − {j};       /* Module j is now placed */
            Evaluate (P);    /* Lower Bound on the cost of Partial Placement */
            if P not satisfactory then F = true ;
            else F = false ;
            if (i ≠ N) then   /* I am not a leaf */
              broadcast (P, S, F);
      end

      if (i = 1)          /* I am a root processor */
      begin
            P = all blanks;              /* Initial Partial Placement */
            S = {1, 2, ···, N};
            F = FALSE;          /* All sub-nodes of root must be expanded */
            broadcast (P, S, F) to all children;
      end
end class ;
```

Figure 2.3: Placement on a Massively Parallel Machine

in the search tree, the optimal placement of n modules requires at least $\Omega(n \log n)$ time using $O(n!)$ processors ■.

2.2 How else can parallel processing help?

Although VLSI has made massively parallel processing a reality, the current state-of-the-art still does not allow us to build an exponential number of processors for meaningful values of problem size. In this sense, the results of the previous section are theoretical. The usefulness of these results is that they allow us to evaluate the

parallelism in a layout problem as opposed to parallelism in a layout algorithm. The practice is to look for the next best thing i.e. analyze the parallelism in a particular layout algorithm and attempt a parallel solution. Most of the current research is directed towards this end. The rest of this chapter indicates the various possibilities which parallel processing opens up for VLSI design.

2.2.1 Better Quality Solutions

It has been mentioned earlier that the only practical approach to solve an NP-complete problem is to use heuristic algorithms. Certain NP-complete problems have been studied for more than three decades and a number of different heuristic techniques have been developed. Even to this day, new heuristics and hybrids of existing heuristics are developed for NP-complete problems of practical significance. Each heuristic technique has its merits and demerits. Two heuristic algorithms can be compared based on the quality of their final solutions for a set of benchmark problems, and their speed performance. Selecting a heuristic technique is a tradeoff between desired solution quality and speed. A heuristic algorithm that gives better solutions is also more compute-intensive, since it searches through a larger subset of the solution space. Simulated Annealing is a good example of this phenomenon. Parallel computing can help improve this situation by speeding up expensive heuristic algorithms.

2.2.2 Superlinear Speedup

Speedup is defined as the ratio T_s/T_p, where T_s is the time taken by the best serial algorithm for the given problem and T_p is the time required by the parallel algorithm. It is believed by the parallel processing community that the maximum speedup obtainable through the use of P processors cannot be more than P; this is known as Amdahl's law. Let N be the number of statements in the serial algorithm. Each statement is assumed to require one unit of execution time; hence, $T_s = N$.

Suppose that only N_p statements of the algorithm exhibit parallelism. The remaining N_s statements are executed sequentially by the parallel algorithm. Under this assumption, $T_p = N_s + \frac{N_p}{P}$. Therefore, the speedup is given by

$$S = \frac{N}{N_s + \frac{N_p}{P}} \tag{2.7}$$

In the best case, $N_p = N$, resulting in a speedup of P. In reality, a speedup of P is never achieved due to overheads of parallel computation. The execution time of the parallel algorithm is correctly given by $T_p = N_o + N_s + \frac{N_p}{P}$, where N_o is the extra code in the parallel algorithm; typically, N_o corresponds to synchronization overheads, interprocessor communication, and task scheduling overheads. If the speedup S is plotted as a function of P, the resulting curve should be a 45° straight line in the ideal case; for most applications, however, the curve is below the 45° line. This is called *sublinear* speedup.

In sharp contrast to sublinear speedup, a phenomenon called *superlinear speedup* has been observed when using parallel search algorithms and *randomized* parallel algorithms. Combinatorial optimization algorithms involve a search through a large solution space. Suppose that a set of processors is used for conducting the combinatorial search in parallel. Each processor works independently and explores a part of the search space. When several processors simultaneously search through the same solution space, it is possible that some amount of work is duplicated among the processors; but, working together, the processors find the solution faster than any one of them working alone. Under certain conditions, the decrease in solution time is more than proportional to the number of processors added [MG85]. Some CAD algorithms have indeed exhibited superlinear speedup when implemented on multiprocessors (see [BB88], for example). Since many physical design algorithms indeed fall into the category of combinatorial search, it is useful to take a closer look at the phenomenon of superlinear speedup.

Combinatorial search can be understood using a "search tree" of the type shown in Figure 2.4. The root node r represents the start state, and the leaf nodes represent *goal states*. The aim of the search algorithm is to reach one of the goal states

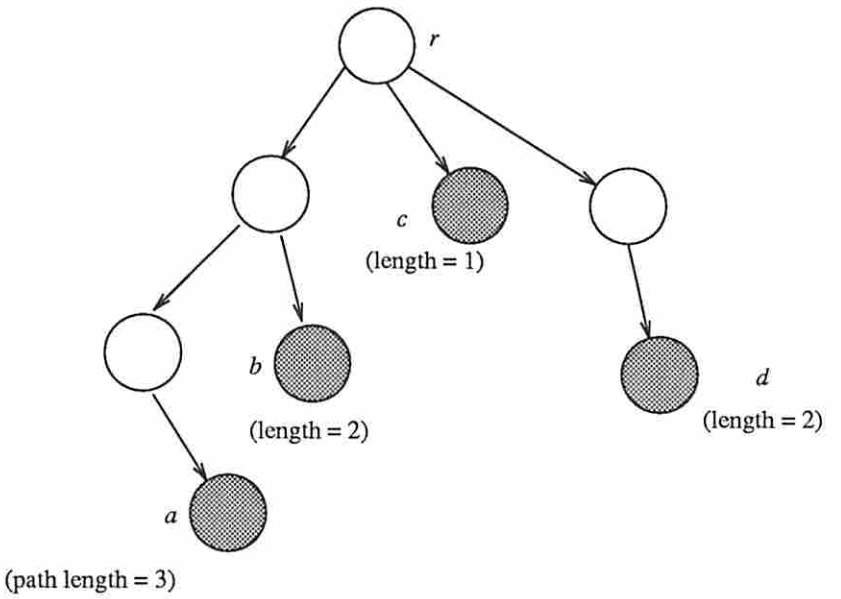

Figure 2.4: A search tree with different path lengths

starting from the root. A deterministic search procedure searches through the tree in a predefined way; two popular search algorithms are *depth first search* and *breadth first search*. Depth first search begins at the root node and visits the left most child of the root that has not already been explored. The newly visited node is now treated as the root and the search continues until a goal state is reached. Breadth first search visits the nodes in the tree level by level. The root is at level 1, the children of the root are at level 2, the grand children of the root are at level 3, and so on. The nodes at any level are visited in the left to right fashion.

The important point to note is that a deterministic search always ends up at the same goal node a. The number of nodes visited by the algorithm before it reaches the goal node is a measure of its execution time. Suppose that T_a nodes are visited before reaching the goal node a. Let b be another goal node that can be reached from the root in T_b steps. A pathological case of deterministic search occurs when T_b is much smaller than T_a. In other words, a deterministic search algorithm cannot recognize a short cut to an alternate goal node. This difficulty can be alleviated by using randomized search. Consider depth first search with the following randomization : beginning at the root node r, select one of the children nodes *randomly* and visit

the selected node. Whenever there is a choice of nodes to visit, the randomized procedure selects by tossing a coin. Consequently, the algorithm can terminate in any one of the goal states. If the randomized procedure is executed k times, it is highly unlikely that the algorithm will end up in the pathological goal state a in all the k runs.

Now consider the following parallel search algorithm. Beginning at the root node, the parallel algorithm visits k children nodes at once. An individual processor is dedicated to search through each of these k paths. The reader will agree that this is equivalent to k executions of the sequential randomized procedure. Let $T(k)$ be a random variable that represents the time taken by k processors to conduct the search. $E[T(k)]$ represents the expected value of $T(k)$. If P is the number of processors used in the parallel search algorithm, a measure of the resulting speedup is

$$S(P) = \frac{E[T(1)]}{E[T(P)]} \tag{2.8}$$

Mehrotra and Gehringer [MG85] showed that $S(P)$ can be greater than P.

Example 2.3 :

Consider the *iterative improvement* algorithm for the circuit placement problem. The basic idea behind the algorithm is to start with an initial placement of modules and attempt to improve the objective function through small changes to the placement. The details of the algorithm will be discussed in Chapter 4. Consider the following method to parallelize the iterative improvement algorithm. P processors are used to independently execute the iterative improvement algorithm starting with P different initial placements. The run time of the iterative improvement algorithm depends on the initial placement. There are pathological initial placements for which the algorithm will run for a very long time before converging to a local optimum. Let p_m be the probability that more than m iterations are required for an instance of the algorithm to converge. Then the probability that all the P executions of the algorithm will lead to more than m iterations is given by $(1 - (1 - p_m)^P)$; this probability

decreases quickly with an increase in P. Therefore, the expected execution time of the parallel implementation falls rapidly with an increase in P.

2.2.3 Searching a Larger Subspace of Solutions

Consider the following variation of the search problem of the previous section. Each goal node is associated with a cost function and the objective is to reach the goal node with the minimum cost function. In Example 2.3 above, each goal node represents a local optimum placement. It is the objective of the search procedure to locate the *best* possible local optimum placement. When a deterministic search procedure is used, it always reaches the same goal node. A randomized search procedure has the chance to visit other goal states. A parallel search procedure imitates multiple executions of the randomized search procedure and therefore has the potential to visit several goal states. Specifically, the procedure can visit P different goal states if there are P processors to search through the state space; the output of the parallel search procedure is the best of these goal states. The following example illustrates these concepts by considering the traveling salesperson problem.

Example 2.4 :

The Euclidean traveling salesperson problem is to find the shortest Hamiltonian tour in a weighted graph of n nodes; each node represents a city and the weight of an edge (i, j) represents the Euclidean distance from city i to city j. The weight of edge (i, j) is denoted c_{ij}. An $n \times n$ matrix $C = [c_{ij}]$ is used to store the weights; if an edge (p, q) does not exist in the graph, then c_{ij} is set to infinity. The "two-opt" procedure is an iterative improvement algorithm for TSP. The procedure takes a Hamiltonian tour T of n cities as input. It may be possible to reduce the length of the tour by swapping the positions of two adjacent cities in T. One iteration of the two-opt procedure consists of examining all possible adjacent pairwise swaps and identifying the pair that gives the highest reduction in tour length. If the highest reduction is negative, the procedure halts; the tour T is a local optimal tour. Otherwise, the tour T is modified by swapping the pair of cities found by the iteration. Figure

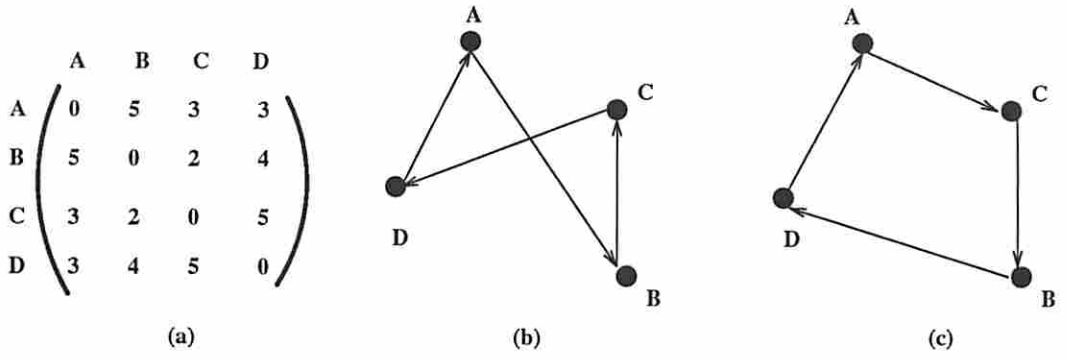

Figure 2.5: Illustration of the two-opt procedure for a 4-city TSP.

2.5 illustrates the algorithm with a small example of 4 cities. The cost matrix is shown in Figure 2.5(a). Suppose that the initial tour T is $A\ B\ C\ D$; the length of the initial tour is 15 (Figure 2.5(b)). There are two possible adjacent pairwise swaps – swap B and C, or swap C and D. The former swap results in the tour $A\ C\ B\ D$ with a cost improvement of 3; the latter results in the tour $A\ B\ D\ C$ with a cost improvement of -2. The algorithm selects the former pair and modifies the tour to $A\ C\ B\ D$; the improved cost function is 12 (Figure 2.5(c)). In the next iteration, the possible swap pairs are (C, B) and (D, B); the improvements associated with these pairs are -3 and -5 respectively. Thus the procedure stops and outputs $A\ C\ B\ D$ as the final tour.

In an n-city problem, there are $\frac{n(n-3)}{2}$ possible adjacent pairwise swaps to be examined per iteration. The number of iterations and the cost of the final tour depend on the initial tour. Suppose that the cities are labeled $1, 2, \cdots, n$. A *deterministic* two-opt procedure (D2OPT) always begins with a specific tour, such as $1\ 2\ \cdots\ n$; it is clear that such a deterministic procedure always terminates in the same local optimum. A *randomized* two-opt procedure (R2OPT) assigns a random permutation of $1, 2, \cdots, n$ as the initial tour. A *parallel* two-opt procedure (P2OPT) uses P processors and assigns a different random permutation of $1, 2, \cdots, n$, say T^i, to each processor i; the processors concurrently apply the two-opt procedure. Let F^i be the final tour generated by

processor i and let $c(F^i)$ be the cost of F^i. The output of the parallel two-opt procedure is F^j, if $c(F^j)$ corresponds to the minimum of $c(F^i)$, $i = 1, 2, \cdots, n$.

The results of parallel two-opt and several other parallel search techniques are described in [Rav90b]. These are briefly discussed below. A *constructive procedure* for TSP is one that builds up a tour T by adding one unvisited city to T at a time. *Farthest insertion* is such a procedure. It begins with a single city in the partial tour T and adds a new city to T in each iteration. It is clear that the procedure requires $n - 1$ iterations. Iteration i consists of two steps – selection and insertion. The selection step picks an unvisited city which is farthest from the partial tour T. Let v be an unvisited city. Define $d(v)$ as the distance between v and the node in T which is closest to v.

$$d(v) = min_{u \in T} \ C_{v,u} \tag{2.9}$$

Then the selection step chooses the city with the maximum value of $d(v)$ for insertion. If (i, j) is an edge in the partial tour, then inserting a city v between cities i and j will mean deleting edge (i, j) and inserting two new edges (i, v) and (v, j). Therefore the cost of inserting v in between i and j is

$$\delta_{ij} = C_{iv} + C_{vj} - C_{ij} \tag{2.10}$$

A greedy insertion procedure is to identify cities $t, h \in T$ such that δ_{th} is minimum, and insert v in between t and h.

Deterministic farthest insertion (DFI) is one in which the initial city is always chosen in the same way, say city 1. Randomized farthest insertion (RFI) chooses the initial city randomly; each city has probability $1/n$ of being selected as the initial city. Parallel farthest insertion (PFI) makes use of P processors, each of which executes RFI starting with a different initial city.

2.2.3.1 Implementation of Parallel Farthest Insertion

We have implemented the parallel farthest insertion algorithm for TSP on both Intel iPSC/2 and Alliant FX/80. The details of these machine architectures are found in

Appendix A. The implementation of PFI on Intel iPSC/2 is now described. Each processor reads in a copy the cost matrix C into its local memory. The file containing the cost matrix is stored on the concurrent file system (`/cfs`) so that all processors can have simultaneous access to the file. On a random basis, each processor selects a different city to start the tour construction. The procedure `lrand48` is used for this purpose; this procedure is provided as a built-in function by the UNIX operating system, and returns a pseudo-random integer. The cities are numbered $0, 1, \cdots, n-1$ and each processor begins its tour with the city `lrand48() modulo` n. The procedure `srand48` is used by each processor to initialize the pseudo-random sequence. In particular, processor i uses `srand48(i+time)` to initialize its sequence. The variable `time` represents the "seconds" portion of the Greenwich Mean time at the instance when `srand48` is invoked; it is obtained using the `gmtime` system call. Since it processor uses a personalized seed to initialize its pseudo-random sequence, it is highly unlikely that two or more processors start with the same city. It is easy to see that if two processors i and j use the farthest insertion heuristic starting with cities s_i and s_j, they will construct identical tours if $s_i = s_j$. Similarly, if $s_i \neq s_j$, the processors i and j will construct different tours. Let u be the number of unique starting cities when P processors are used to execute the parallel farthest insertion heuristic. Table 2.1 shows the results of running PFI on an iPSC/2 with 32 processors. Two problem instances were used and the experiment was repeated 5 times in each case. For each experiment, we show the minimum, maximum and average tour lengths found by the processors. As can be seen, u is sufficiently close to P and approaches P for larger values of n. This is important, because $P - u$ represents the amount of redundant work in the parallel farthest insertion algorithm. The redundancy can be further reduced by using better pseudo-random generators. The tour length obtained by a deterministic farthest insertion algorithm which always starts the tour with city 0 is shown as *dfi*. The PFI algorithm outperforms DFI in both average and best case. For example, in problem `p532`, the best tour found by the PFI algorithm is 29499, which is roughly 2% better than the tour length of 30051

Problem k24, $n = 100$, $P = 32$					
	Exp1	Exp 2	Exp 3	Exp 4	Exp 5
u	20	23	24	23	22
max	24006	24089	24089	24089	24006
min	22047	21628	21628	21628	21628
avg	22738	22821	22761	22755	22650
dfi	23144				

Problem p532, $n = 532$, $P = 32$					
	Exp1	Exp 2	Exp 3	Exp 4	Exp 5
u	27	32	31	32	32
max	30874	30849	30629	30874	30849
min	29499	29576	29576	29499	29576
avg	30186	30193	30246	30158	30215
dfi	30051				

Table 2.1: Results of Parallel Farthest Insertion Heuristic for TSP on Intel iPSC/2.

generated by DFI. This improvement can be further enhanced by making the following change to the PFI algorithm. Each processor executes the farthest insertion heuristic k times, starting with a random initial city. This amounts to Pk executions of the randomized farthest insertion procedure. We denote the modified PFI algorithm as Multiple Execution PFI, or MFI for short. The percentage improvements obtained by MFI for varying values of k are plotted in Figure 2.6.

2.2.3.2 Parallel k-opt

The two-opt procedure of Example 2.4 can be generalized to result in a k-opt procedure. The two-opt procedure uses adjacent pairwise interchange to modify a tour; this amounts to deleting two edges from the tour and adding two fresh set of edges that result in a different tour. The tour $A\ B\ C\ D$ of Figure 2.5(b) is modified to the tour $A\ C\ B\ D$ of Figure 2.5(c) by deleting edges AB and CD and adding the edges AC and BD. A k-opt procedure deletes k edges from the tour T and adds a fresh set of k edges so that the resulting graph is still a Hamiltonian tour. It was mentioned in Example 2.4 that $O(n^2)$ swaps must be examined for every iteration of two-opt.

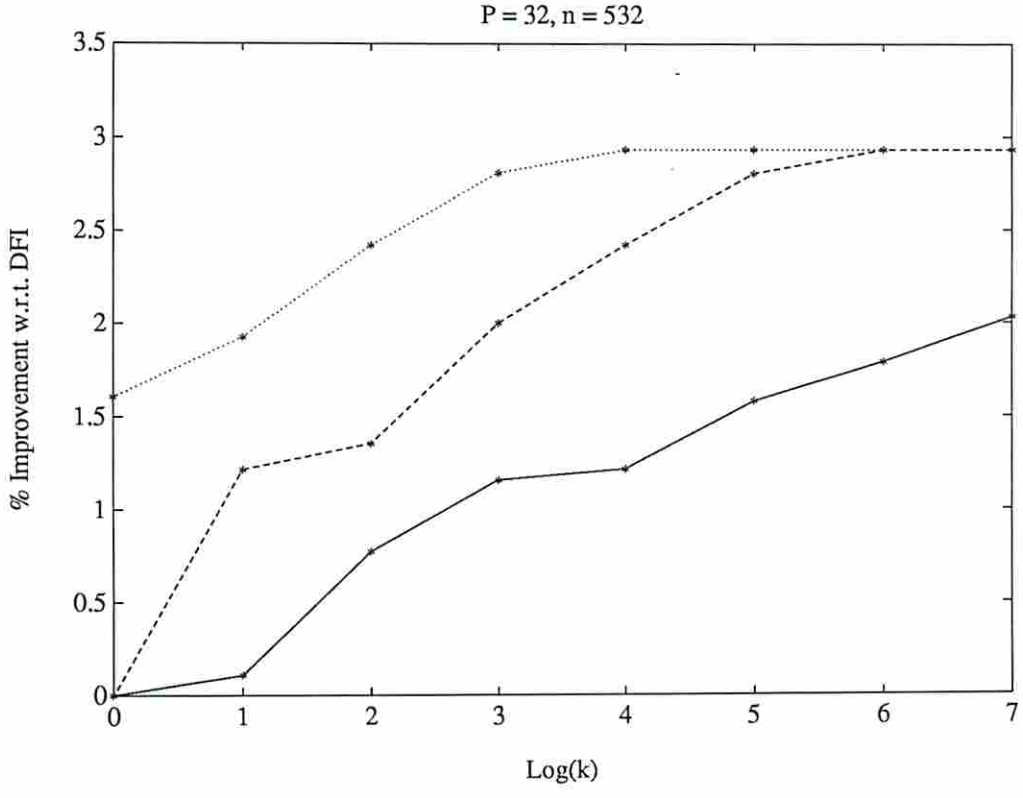

Figure 2.6: Results of Multiple Execution PFI for the 532-city problem. The dotted curve corresponds to $P = 32$. The dashed curve corresponds to $P = 8$. The solid curve corresponds to $P = 1$.

In general, $O(n^k)$ alternatives must be examined in a k-opt algorithm. For $k > 2$, the k-opt algorithm is complicated to implement, since it is non-trivial to maintain the Hamiltonian property of the tour while adding a fresh set of k edges. Even for $k = 3$, the k-opt algorithm is too compute-intensive to be affordable for large n. About 3 days of CPU time was required to run the three-opt procedure for a 532 city TSP on a 4 MIPS computer [Rav90b]. Nevertheless, the three-opt procedure does result in solutions that are superior to those generated by the two-opt procedure. It is possible to define the deterministic, randomized and parallel versions of a k-opt procedure just like it was done for two-opt. These are denoted DKOPT, RKOPT and PKOPT respectively.

P2OPT and P3OPT algorithms were implemented on iPSC/2. Several variations of P2OPT algorithm were implemented, as explained below.

1. FI+P2OPT : Each processor obtains an initial tour using the farthest insertion heuristic. The initial city for farthest insertion is obtained using a pseudo-random generator, as discussed in the previous section. Each processor executes the "two-opt" procedure starting with its initial tour; processors proceed independently of one another since each has a copy of the cost matrix. After all the processors converge to a local optimal tour, the shortest tour is identified using the global function `gilow`. This function applies a distributed algorithm to compute the `minimum` of P integers $x_1, x_2, \cdots, x_P$; as input to the function, processor i supplies the pointer to local variable x_i.

2. RAND+P2OPT : Each processor applies the two-opt procedure to a randomly generated initial tour. A random tour T is generated by constructing a random permutation of $0 \cdots n-1$ as shown below. The two-opt procedure is identical to (1) above.

```
RandomTour(n);
    for i = 0 to n − 1 do T[i] = i;
    for i = 0 to n − 1 do begin
        r1 = lrand48() mod n;   /* Random integer, 0 ··· n − 1 */
        r2 = lrand48() mod n;
```

```
            swap (T[r1], T[r2]);
        end
    procedure ;
```

3. MFI+P2OPT : Identical to (1), except that each processor executes the farthest insertion technique k times and selects the best of the k tours as the initial tour. The parameter k is provided by the user as input to the program.

4. MFI+C2OPT : Each processor uses k executions of farthest insertion heuristic to construct an initial tour. The best of these initial tours is identified using the `gilow` global function. One or more processors may have arrived at the best initial tour; among these, the processor whose **id** is the lowest broadcasts the best initial tour to all the remaining processors. The following code is used to achieve the broadcast operation. Each processor sets a flag variable to 1 (or 0) if it has (or has not) found the best tour in the MFI phase. These flag variables are bunched together into a vector of P flags using the **global collate** operation called `gcol`. If i is the first non-zero entry in the flag vector, then processor i is the lowest indexed processor with the best tour from MFI.

```
/* cost contains the length of the best of k tours constructed
   using the farthest insertion heuristic. bestcost contains the
   smallest of cost_1, ..., cost_P and is calculated using gilow function.
*/
if (cost = bestcost) then flag = 1 else flag = 0;
collate(flag, flagvector)
for i = 0 to P do
    if flagvector[i] = 1 then break;
if (mynode() = i) then
        broadcast(T)
else
        receive (T);
```

The C2OPT phase of the algorithm is a distributed version of the "two-opt" procedure. Recall that the two-opt procedure on an n city problem must evaluate $\frac{n(n-3)}{2}$ adjacent pairwise swaps during each iteration. The basic idea in parallelizing the procedure is to distribute these pairwise swaps equally among the P processors for concurrent evaluation. The following procedure shows the structure of the two-opt algorithm.

```
procedure two-opt;
begin
 /*
  let H = (x_0, x_1, ..., x_{n-1}) be the n edges in the initial tour.
  the notation C(x_i) is used to indicate the cost of edge x_i.
 */
  repeat
    δ_max = 0;
    for i = 0 to n - 3 do
        for j = i + 2 to n - 1 or n - 2 do
            /* Latter case for i = 0 only.
               Evaluate the cost improvement from deleting edges x_i, x_j
               and adding new edges y_p, y_q which complete the tour */
            δ_move = (C(x_i) + C(x_j)) - (C(y_p) + C(y_q));
            if δ_move > δ_max then begin
                  save i and j;
                  δ_max = δ_move
            end
    if δ_max > 0 then
          update tour by deleting x_i, x_j and adding y_p, y_q;
  until δ_max = 0;
end
```

The distributed two-ont

tation

The distributed two-opt algorithm is shown in Figure 2.7. The core computations are parallelized by distributing the $n-2$ iterations of the outer **for** loop among the P processors. Thus each processor examines $\frac{n(n-3)}{2P}$ pairwise interchanges and identifies a pair (i,j) which results in the largest reduction in tour length; this largest reduction is denoted δ_{local_max}. At the end of the **for** loop, the processors compute δ_{max}, the maximum of δ_{local_max}, in a distributed fashion using the `gihigh` global function. Several pairwise interchanges may result in a gain of δ_{max}; the smallest indexed processor which has such a pair broadcasts the interchange pair to all the processors. This enables all processors to update their copy of the tour. Therefore, the overhead of parallelization consists of

(a) computation of δ_{max} using `gihigh`

(b) broadcast operation explained above

Both these overheads require $O(\log P)$ time on a P-processor hypercube. When n is much larger than P, the core computations overshadow the overheads and a speedup close to P is realized.

The three-opt algorithm was parallelized in ways similar to the two-opt algorithm. Three-opt requires $O(n^3)$ computations per iteration and hence computationally much more expensive than the two-opt. The core of three-opt algorithm consists of three nested loops; the distributed version of three-opt, called C3OPT, distributes the n iterations of the outermost loop among P processors for concurrent execution in a manner similar to C2OPT. The different variations of two-opt algorithm were tested against standard benchmark problems. For the famous 532-city benchmark, the shortest tour of length 28285 was generated by the P3OPT algorithm. 32 processors were used in the experiment and the algorithm required more than 3 days to converge.

Table 2.2 shows the results of parallel search techniques when applied to several instances of TSP. 32 processors were used in parallel search. P3OPT generated the best solution in each case; it was also the most compute-intensive algorithm.

```
procedure distributed-two-opt;
begin
   repeat
      δ_local_max = 0; /* Best swap gain as seen by this processor */
      for i = mynode() to n − 3 step P do
                  ⋮ /* core identical to two-opt */
      end
      δ_max = max_{k=1}^{P} (δ_local_max)
      if δ_max > 0 then begin
          if mynode() is the smallest index such that
           (δ_local_max = δ_max) then
                broadcast (swap pair (i, j), (p, q))
          else
                receive (swap pair (i, j), (p, q))
          update tour by deleting x_i, x_j and adding y_p, y_q;
      end
   until δ_max = 0;
end
```

Figure 2.7: Distributed Two-opt Algorithm.

The tour length shown against an algorithm A corresponds to the minimum length tour generated using A i.e. the minimum taken over all experiments using A with different values for the number of processors P. Further, the same algorithm A may generate its best solution several times, for different values of P; the execution time shown in the table against A is the minimum over all such cases. Table 2.3 shows the performance of deterministic search procedures on the same instances of TSP; these procedures were executed on a single node of Intel iPSC/2. Appendix A gives an overview of the machine architecture and the programming environment of the iPSC/2. The search procedures discussed above were coded in C. Comparing the tables 2.2 and 2.3, it can be seen that parallel search always generates superior solutions compared to deterministic search. Superlinear speedups were observed with the parallel search procedures, but not consistently. For example, in problem k24, the DFI procedure required about 69 seconds; the PFI procedure takes 1.684 seconds using 32 processors. This represents a speedup of 40.

Problem Name	# Cities	PFI		FI+P2OPT		FI+P3OPT		MFI+C2OPT	
		Tour	Time	Tour	Time	Tour	Time	Tour	Time
k24	100	21628	1.68	21591	2.59	21247	9060	21628	0.98
k25	100	22926	1.86	22587	2.48	22312	7108	22640	3.25
k26	100	20878	1.67	20865	2.39	20703	5388	20878	0.99
k27	100	21868	1.99	21594	2.33	21347	8403	21731	1.26
k28	100	22498	1.99	22441	2.29	22062	9285	22456	1.41
p532	532	29576	32.9	29405	96.9	28285	3 days	29195	103.8

Table 2.2: Results of Parallel Search for TSP. All times are in seconds, except for the last entry in the P3OPT column

Problem Name	# Cities	DFITSP Tour	D2OPT Tour	D3OPT Tour
k24	100	23144	22621	21530
k25	100	23189	23789	22348
k26	100	21142	20965	20723
k27	100	23197	22495	21347
k28	100	22456	23796	22062
p532	532	30051	29606	28285

Table 2.3: Performance of Deterministic Algorithms on TSP Instances

The performance of FI+P2OPT as a function of P is shown in Figure 2.8 for two problems, the 532 city problem (p532) and a 100 city problem (k24). The performance of RAND+P2OPT technique is shown in Figure 2.9 for the 532-city problem. It is seen that with increase in number of processors, both algorithms give better results in terms of solution quality. However the synchronization overheads are also higher when there are more processors, resulting in an increase in execution time. For the same value of P, FI+P2OPT always performed faster than RAND+P2OPT. The solution qualities of the two algorithms were compared for the same values of P and $k = 1$. It was also observed that FI+P2OPT consistently generated better quality solutions. This confirms with the folklore that a constructive initial solution is better than a random initial solution. To test if MFI+P2OPT outperforms FI+P2OPT, the two algorithms were compared with identical inputs and were executed with the same number of processors. Figure 2.10 shows the results obtained using MFI+P2OPT. For a given number of processors, increasing k improves the solution quality up to a point; for larger values of k, a saturation effect sets in. For example, when $P = 16$, $k > 16$ does not lead to better quality solutions.

Figure 2.11 shows the performance of MFI+C2OPT. Four different plots are shown, corresponding to $P = 2, 8, 16, 32$. With increase in k, the speedup of the parallel algorithm is seen to improve. For $k \geq 128$, we observed that speedup reaches the optimum value of P.

2.2.4 New Ways of Solving Problems

New parallel architectures have inspired new algorithms for computer-aided design. For example, the parallel algorithms for Simulated Annealing are distinctly different from their sequential counterparts. In fact, one of the earliest works on parallel placement used an unconventional algorithm [U+83]. In an attempt to parallelize iterative improvement, the authors of [U+83] proposed making multiple perturbations to the placement and evaluating these perturbations independently. Although there is a small chance that the algorithm does not converge, its expected performance is

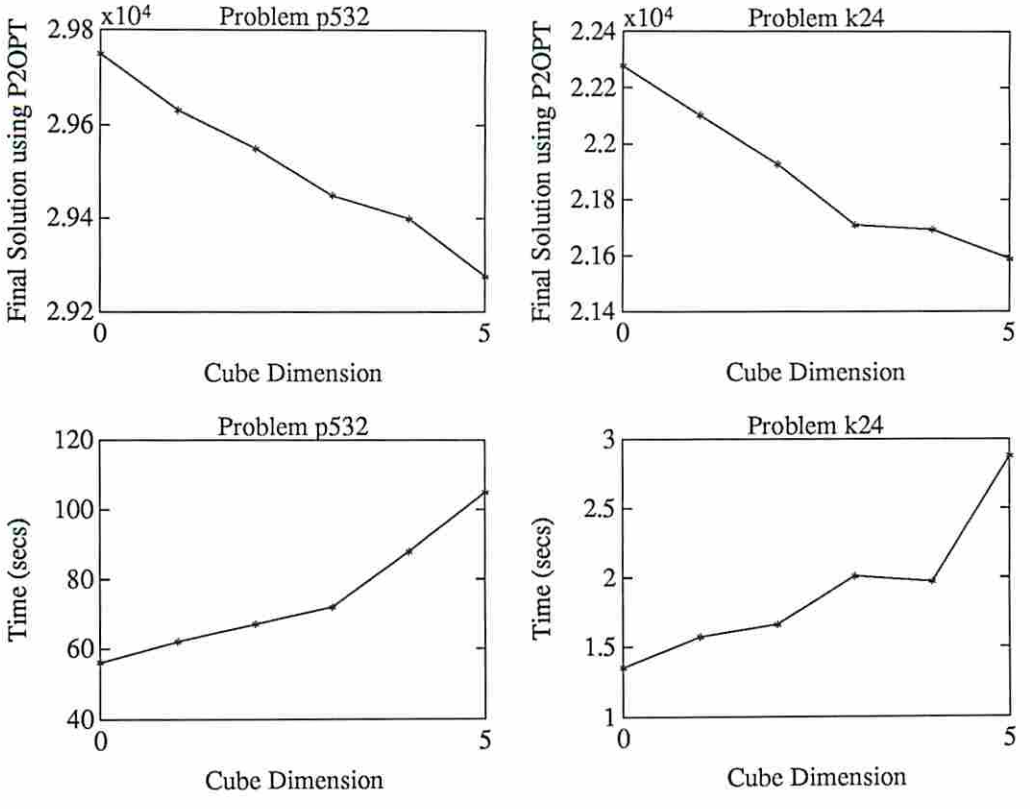

Figure 2.8: Performance of FI+P2OPT as a function of the number of processors.

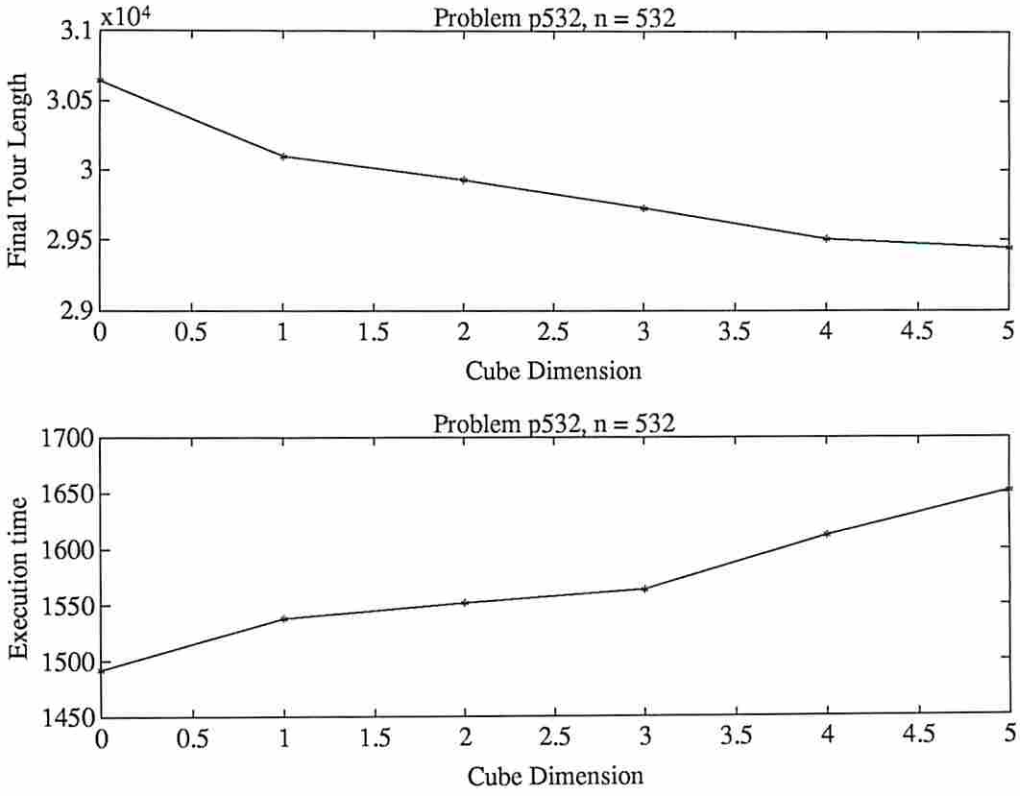

Figure 2.9: Performance of RAND+P2OPT for the 532 city problem.

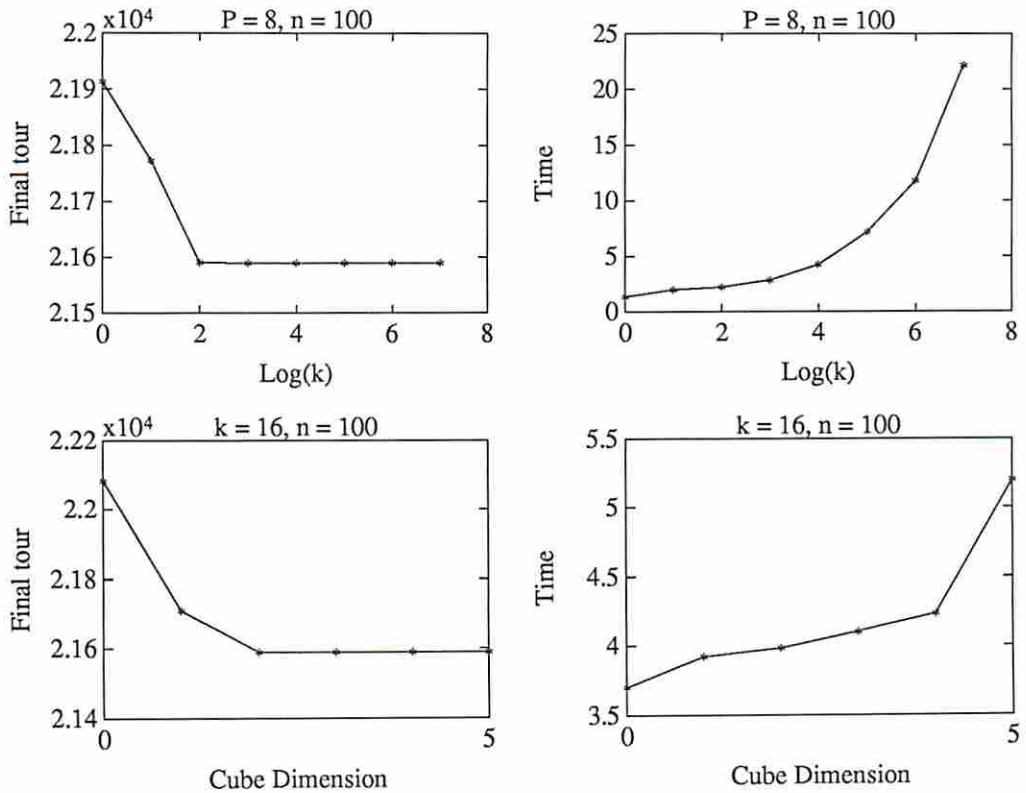

Figure 2.10: Performance of MFI+P2OPT for the 100 city problem.

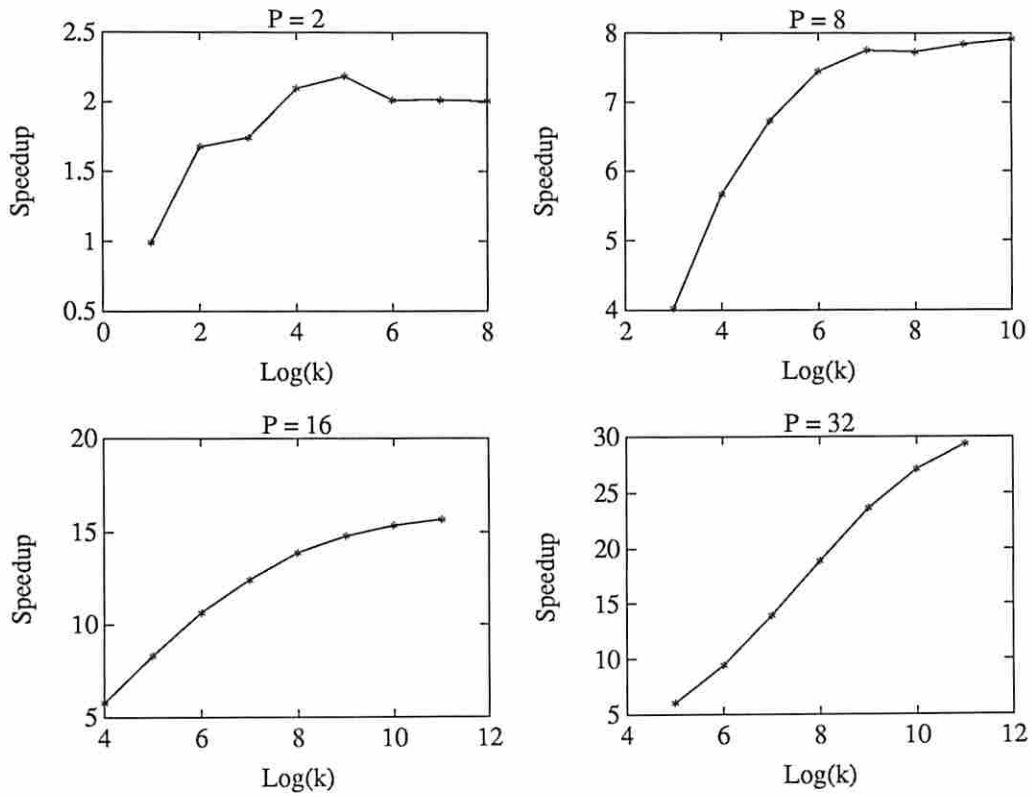

Figure 2.11: Performance of MFI+C2OPT for the 532 city problem.

good. Since several processors make modifications to the system configuration asynchronously, the objective function (wire length) does not decrease monotonically. As a result, the algorithm has a small chance of accepting inferior states and climbing out of a local minimum.

Completely new paradigms of computing, which are inherently parallel, are being explored today for various applications. Artificial Neural Networks (ANNs) are one such. Although their principal application seems to be in computer vision and pattern recognition, some authors have explored the use of ANNs for solving optimization problems [HL89, Yu89]. It is likely that a number of CAD applications will be investigated in the near future as possible candidates for ANN related research.

2.3 Summary

NP-complete problems have been studied for more than thirty years and a body of knowledge exists on techniques to find sufficiently good solutions to these problems. Selecting a solution method for an NP-complete problem is a tradeoff between solution quality and computational requirements. A good example of this phenomenon is found in the k-opt technique for solving the TSP; the larger the value of k, the closer the final solution to the optimum and larger the computation time. Simulated Annealing is another technique that yields solutions close to the optimum at the expense of enormous amounts of computation. The use of parallel processing can shift the slope of the quality-time curve in the favorable direction by making the expensive-but-good heuristics affordable.

Randomization is a good weapon against NP-completeness. In fact, an NP-complete problem is defined to be one that may be solved in polynomial time by a nondeterministic automaton. Coupling randomization and parallel processing leads naturally to parallel randomized algorithms; these algorithms hold two promises – better quality solutions and superlinear speedup. Since multiple processors are used to simultaneously explore the combinatorial search space starting with different initial states, it is intuitively clear that more of the search space is covered by

a parallel search than a sequential search in the same amount of time. Presently, there is no way to guarantee that the state subspaces searched by two different processors are indeed non-overlapping. This property is desirable since it would ensure that the parallel algorithm necessarily generates better quality solutions compared to its sequential counterpart. Improvements to this work will come from parallel optimization algorithms that ensure the non-overlap property.

The parallel algorithms described in this chapter are well suited for implementation on MIMD computers. The advantage of algorithms such as PFI, P2OPT and P3OPT is that their existing sequential implementation can be ported to a parallel machine without much effort. Distributed algorithms such as C2OPT and C3OPT require more elaborate parallel implementation. The techniques described here are also sufficiently general. For example, iterative improvement algorithms similar to "two-opt" are used for circuit placement and routing. Goto [Got81] describes a λ-opt algorithm for two-dimensional circuit placement which starts with a *good random* initial placement and improves it by shuffling λ cells at a time. Banerjee and Brouwer [BB88] have described a two-layer channel routing algorithm which starts with a track assignment using exactly α tracks, α being the channel density of the problem. The initial track assignment may violate vertical constraints. A move consists of altering the track assignment for two nets. The algorithm performs moves and accepts those that decrease the number of vertical constraint violations. The algorithms in [Got81], [BB88], and many others can both be adapted for MIMD implementation using our techniques.

Chapter 3

Parallel Circuit Partition

This chapter will consider parallel algorithms for two-way circuit partition. It was seen in Section 1.6.1 that this problem is closely related graph bisection, which is known to be NP-complete. Kernighan and Lin [KS70] proposed a heuristic algorithm for graph bisection. Now a classic, the Kernighan-Lin algorithm has been used in several applications – logic packaging, placement, program partition, load balancing, and so on. It is known to generate near-optimal partitions for dense graphs of large size. Specifically, if the average degree of the nodes in the graph is greater than 3, the Kernighan-Lin algorithm is known to give excellent solutions. This chapter presents parallel implementations of Kernighan and Lin's algorithm. First a PRAM algorithm is discussed for Kernighan-Lin partition scheme. This algorithm is extended to work on two different parallel architectures – a shared memory SIMD array processor and a shared memory multiprocessor.

3.1 Kernighan-Lin Algorithm

Let Z be a set of circuit modules. Given an arbitrary initial partition of Z into two sets X and Y, the Kernighan-Lin (KL) procedure repeatedly improves the partition obtaining a better partition in each iteration, until a local optimum is reached. The essential idea behind the KL heuristic is the following. There are some modules in the set X and an equal number of modules in set Y that are *out of place* in the

sense that if these modules are moved to the opposite set, the resulting partition is optimal. The *KL* procedure tries to identify these *out of place* modules.

Let $a \in$ X and $b \in$ Y. The *external cost* E_a and *internal cost* I_a associated with the module a are given by

$$E_a = \sum C_{ay} \; \forall y \in Y \tag{3.1}$$

$$I_a = \sum C_{ax} \; \forall x \in X \tag{3.2}$$

where C_{ay} gives the number of wires connecting modules a and y. E_a measures how strongly a is attracted to X, whereas I_a measures how strongly a is attracted to Y. The 'D value' associated with module a is defined as

$$D_a = E_a - I_a \tag{3.3}$$

The 'D value' of module a is a measure of how much *out of place* module a is. That is, the larger the value of D_a, the more likely that a belongs to Y rather than to X. D_b is defined similarly. The connections between modules a and b contribute to the external cost before and after the swap. Therefore, the gain g_{ab} of swapping the two modules is $(D_a - C_{ab}) + (D_b - C_{ab})$, and hence,

$$g_{ab} = D_a + D_b - 2 \cdot C_{ab} \tag{3.4}$$

Figure 3.1 shows the steps of the Kernighan-Lin algorithm for two-way partition. Given the initial partition (X, Y), step P1 computes the D values for all the modules. Step P2 identifies two modules, one in set X and another in set Y such that swapping them decreases the external cost by the maximum extent. These two modules are then 'locked' and are not considered further as candidates for swap. Since two modules are locked at a time, $n/2$ iterations are necessary before all the modules get locked. This explains why the **for** loop in the procedure is executed $n/2$ times. Suppose g_i is the gain of swapping two modules during the ith iteration of the **for** loop. The cumulative gain after iteration j of the **for** loop is given by $\sum_{i=1}^{j} g_i$. How

```
procedure PARTITION (X, Y) ;
begin
        repeat
                X1 = X ; Y1 = Y ;
P1:             Compute D values for all a in X1 and all b in Y1 ;
                for i := 1 to n/2 do begin
        P2:         Find a_i and b_i which maximizes g_i = D_{a_i} + D_{b_i} - 2.C_{a_i b_i} ;
        P3:         Move a_i to Y1 and b_i to X1 ;
        P4:         Lock a_i and b_i;
                    Update D values for the elements of X1 - {a_i} and Y1 - {b_i}
                end { for }
        P5:         Find k that maximizes g_max = sum_{i=1}^{k} g_i
        P6:         if g_max > 0 then
                      Exchange a_1, a_2, ..., a_k with b_1, b_2, ..., b_k.
        until g_max ≤ 0 ;
end procedure
```

Figure 3.1: Kernighan-Lin Procedure

does the cumulative gain behave as a function of j? When $j = 0$ i.e. before the execution of the **for** loop, the cumulative gain is 0 since no swaps have taken place. At the completion of the loop, when $j = n/2$, the cumulative gain should again be 0, since all the modules have been swapped and sets X and Y have been effectively interchanged. Let k be the iteration index at which the cumulative gain is maximum. The algorithm adopts a greedy strategy and retains only those swaps made during iterations $1, 2, \cdots, k$ of the **for** loop (step P6). All the modules are then 'unlocked' and the entire **for** loop is repeated. The **repeat** loop terminates if the cumulative gain of the **for** loop is not greater than 0.

The procedure requires $O(n^2 \log n)$ time when used with a circuit of n modules [KS70]. This can be seen by referring to Figure 3.1. The **repeat** loop is known to execute a constant number of times. The **for** loop is executed $n/2$ times, and the most expensive step in the body of the **for** loop is P2. Consider the ith iteration of the **for** loop. There are $n/2 - 2 \cdot i$ free modules in each of the two sets X and Y. Step P2 must pick one module from each set such that the gain of swap is maximum. A brute-force method to identify the best swap-pair is to examine all the $(n/2 - 2 \cdot i)^2$

possible pairs. Kernighan and Lin suggested a better way. Sort the free modules in X in the decreasing order, based on their D values. Similarly sort the modules in the set Y. Use two pointers to scan down the two sorted lists. Keep track of the maximum gain while scanning down the lists. If there exist two modules (say a and b) in the lists such that $(D_a + D_b)$ is no larger than the maximum gain so far, then there is no need to examine the modules further down in the two lists. This is because the gain of swapping modules a and b is $D_a + D_b - 2 \cdot C_{ab}$, which is no more than $(D_a + D_b)$. Again, since the lists are sorted in the descending order, the gain of the pairs further down in the lists are also less than $(D_a + D_b)$.

Soring $(n - 2 \cdot i)$ elements requires $O(n \log n)$ time using the heap sort algorithm. Scanning the lists should not require more than $O(n)$ time. Therefore, the body of the **for** loop requires $O(n \log n)$ time. This verifies the claim that the Kernighan-Lin algorithm takes $O(n^2 \log n)$ time. In this chapter, parallel algorithms are considered to obtain an *order of magnitude* improvement in performance. Section 3.2 will consider the parallel execution of the Kernighan-Lin procedure on a theoretical SIMD machine model known as EREW PRAM. Section 3.3 describes a data parallel partition algorithm which uses a novel array architecture.

3.2 Parallel Kernighan-Lin on an EREW PRAM

Consider a circuit with n logic modules. In this section, a parallel algorithm is described to partition the circuit into two equal-sized subcircuits. The target machine is an abstract model known as the Parallel Random Access Machine. Later the abstract algorithm is extended to operate on a more practical machine known as the Orthogonal Array.

It is assumed that the PRAM has n processing elements (PEs). The PEs are denoted $PE_1, PE_2, \cdots, PE_n$. The connectivity matrix $C = [C_{ij}]$ is stored in the shared memory of the PRAM. The PRAM operates in the Single Instruction, Multiple Data (SIMD) mode. Thus there is a control processor to fetch instructions and decode them. Scalar instructions are also executed by the control processor.

Vector instructions are broadcast to the PEs for simultaneous execution. Typical vector instructions are VCOPY, VMAX, PARTSUM, etc. Table 3.2 describes these vector instructions. For example, the VMAX instruction stands for "vector maximum." To execute this instruction, the name of an array X and two indices s and d are needed as parameters. VMAX computes the maximum of the the set $\{V(s), V(s+1), \cdots, V(d)\}$.

Apart from conventional vector instructions like VMAX, the vector instruction set of the machine is enriched with the macro-instructions PAR-P1, PAR-P2, $\cdots$, PAR-P6. For example, macro PAR-P1 corresponds to parallel execution of step P1 of the procedure *PARTITION* .

3.2.1 Computing the D values in Parallel

Step P1 of the *PARTITION* algorithm involves the computation of D values. One module is assigned to each PE in the SIMD machine. Let PE_i be assigned module i. Each PE has several registers:

- Register I holds the identification number of the PE.
- Register M contains the *membership information* of the module i; M contains 0 if the module belongs to set X, whereas M contains 1 if the module belongs to set Y. The membership information of all modules is maintained in the global array *MEMBER* . The k th element of the array *MEMBER* contains 0 if module k belongs to set X ; the element contains 1 otherwise. The M register and the array *MEMBER* are used in the computation of D values, as explained later.
- Register PD holds the *partial D value.* This register is initialized to zero before the machine begins the parallel execution of step P1. At the end of step P1, the PD register of PE_i holds the D value for module i.
- TEMP is a scratch register.

The execution of the vest

procedures

The execution of the vector instructions may be understood through the following procedures. For example, the procedure EXECUTE-PAR-P1 is executed by every PE when the control processor broadcasts the vector instruction PAR-P1 to the PEs.

```
procedure EXECUTE-PAR-P1 ;
begin { computation of D values }
    for j := 1 to n do
        if modules j and i belong to the same set then
            PD := PD − C_ij
        else
            PD := PD + C_ij
{ C_ij : element in the i th row and j th column of C
```

To test whether modules

The execution of the vector instructions may be understood through the following procedures. For example, the procedure *EXECUTE-PAR-P1* is executed by every PE when the control processor broadcasts the vector instruction PAR-P1 to the PEs.

procedure *EXECUTE-PAR-P1* ;
begin { computation of D values }
    for $j := 1$ **to** n **do**
        if modules j and i belong to the same set **then**
            $PD := PD - C_{ij}$
        else
            $PD := PD + C_{ij}$;
    { C_{ij} : element in the i th row and j th column of C }
end procedure

The procedure *EXECUTE-PAR-P1* is self explanatory. To test whether modules i and j belong to the same set, PE_i compares the value in M register and the value in $MEMBER(j)$. If the two values are identical, it means that the modules i and j belong to the same set. The computational complexity of *EXECUTE-PAR-P1* is $O(n)$. The complexity of the sequential execution of P1 is $O(n^2)$.

3.2.2 Selecting Modules for Interchange

Recall that step P2 of *PARTITION* finds the pair (a_i, b_i) such that $a_i \in X1$, $b_i \in Y1$, and the gain which results by swapping the modules a_i and b_i is maximized. The dominant term in the execution time of the *KL* algorithm is contributed by step P2. A brute-force implementation of the step P2 will evaluate the $n^2/4$ gains corresponding to all the pairs (a_i, b_j), and then compute the maximum gain. This leads to an $O(n^2)$ implementation of P2, and, since P2 is executed $O(n)$ times, to an $O(n^3)$ overall execution time of the procedure *PARTITION* . The method used by Kernighan and Lin consists of sorting the D values such that

$$D_{a_1} \geq D_{a_2} \geq \cdots \geq D_{a_{n/2}} \tag{3.5}$$

and

$$D_{b_1} \geq D_{b_2} \geq \cdots \geq D_{b_{n/2}} \ . \tag{3.6}$$

When sorting is used, only a few contenders for a maximum gain need to be considered. This is because, when scanning down the set of D_a's and D_b's, if a pair D_{a_i},D_{b_j} is found whose sum does not exceed the maximum improvement seen so far in this pass, then there cannot be another pair (a_k,b_l) such that $k \geq i, l \geq j$, with a greater gain, (assuming $C_{ij} \geq 0$) and so the scanning can be terminated. Although this implementation of P2 has a *worst case* execution time of $O(n^2)$, its average execution time is $O(n \log n)$. This is because sorting is an $n \log n$ operation on sequential computers. The overall average execution time of the KL procedure improves to $O(n^2 \log n)$ when the sorting scheme is employed in implementing P2 [KS70].

In parallelizing the KL partitioning algorithm on n processors, step P2 is indeed the bottle neck in obtaining the optimal speedup of $O(n)$. It has already been shown that step P1 can be parallelized with an $O(n)$ speedup on an SIMD-EREW computer. It is shown later that optimal speedups are obtained for the other compute-intensive steps of the KL algorithm as well. However, it is provably not possible to realize an $O(n)$ speedup of step P2 using n processors and an SIMD-EREW model of computation.

Lemma 2 *The speedup obtainable by executing step P2 on an SIMD-EREW computer with n processors is bounded above by $n/\log n$.*

The proof follows from the arguments given below.

- If the brute-force version of P2 is parallelized, then essentially n processors are used to parallelly compute the maximum of $n^2/4$ gains. The computation of the maximum of $O(n^2)$ elements on an SIMD-EREW machine with n processors is known to take $\Omega(\log^2 n)$ time. Therefore, the obtainable speedup cannot exceed $O(n \log n)/\ O(\log^2 n)$, which is equal to $O(n/\log n)$.

- The other alternative is to parallelize the *sorting implementation* of P2 described above. However, the best known *deterministic* sorting methods on an

SIMD-EREW computer with n processors require $\Omega(\log^2 n)$ time. By applying an argument similar to the preceding one, it can be seen that the speedup of step P2 cannot be greater than $O(n/\log n)$ ■.

In the light of the above result, the choice of a strategy for parallelizing step P2 is, by large, an engineering judgement. However, it must be noted that both the methods discussed above require complex control structures when implemented on an SIMD computer. In what follows, a *third* alternative is described for the implementation of step P2. This method is based on a heuristic procedure suggested by Kernighan and Lin to select the pair (a_i, b_i). The advantage of this heuristic is that it leads to a very simple parallel implementation, at the same time providing the maximal $O(n/\log n)$ speedup. The selection heuristic is the following. The module a_i corresponding to the maximum value of D_{a_i} is selected. Similarly, the module b_i is selected to maximize D_{b_i}. On a sequential computer, the heuristic procedure requires linear amount of time. On an SIMD-EREW machine with n processors, the computation of the maximum of n elements is known to take $O(\log n)$ time. Thus, by implementing this heuristic on the parallel version of the *KL* procedure, it is possible to obtain the dual advantages of simple control structure and maximal speedup. The disadvantage of using the heuristic is, of course, that it may not result in a pair (a_i, b_i) which maximizes the gain of swapping. However, it must be borne in mind that the entire partitioning procedure itself is a heuristic, and one does not gain significantly by choosing to obtain the optimal solution for a particular step of the heuristic. In fact, experience with optimization techniques such as Simulated Annealing [KGV83] has shown that optimization heuristics may perform *better* in terms of the quality of the final solution if one adopts a *non-greedy* strategy and introduce a small probability of choosing a *less than the best* solution during the course of iterative improvement. Guided by the above considerations, we choose to implement the *KL heuristic* for module selection. In our parallel implementation of step P2, the modules a_i and b_i are located in two steps as seen in the procedure *EXECUTE-PAR-P2* .

procedure *EXECUTE-PAR-P2* ;

begin

Step 1 : *Locate* a_i

if M register contains 0 **then**

$TEMP := PD$

else

$TEMP := -\infty$;

Use binary tree computation to locate

the maximum of the $TEMP$ registers ;

Step 2 : *Locate* b_i

if M register contains 1 **then**

$TEMP := PD$

else

$TEMP := -\infty$;

Use binary tree computation to locate

the maximum of the $TEMP$ registers ;

end procedure

Since the computation of the maximum of n numbers on an array processor with n PEs requires $O(\log n)$ time, the time complexity of *EXECUTE-PAR-P2* is $O(\log n)$. A uniprocessor requires $O(n)$ time to execute step P2 sequentially.

Referring to the procedure *PARTITION* , it is seen that modules a_i and b_i are removed from further consideration in the pass (current iteration of the **repeat** loop) after the modules have been swapped. In the parallel implementation of *PARTITION* , the same effect can be created as follows. The control processor swaps the modules a_i and b_i after all the PEs complete the execution of the vector instruction PAR-P2. The control processor deactivates the processors that are assigned to modules a_i and b_i . The deactivation can be achieved by the use of a mask register.

3.2.3 Updating the D values

The D values for all the modules must be updated after modules a_i and b_i have been swapped in step P3 of the *PARTITION* algorithm. The control processor broadcasts the module numbers of a_i and b_i to all the PEs. Each PE_k executes the following steps after receiving the module number for a_i (similarly for b_i).

procedure *PAR-UPDATE-D-VALUES* ;
begin { update D_i }
    if (*MEMBER* (a_i) = M) **then**
        { modules k and a_i are in the same set }
        $PD_k := PD_k + C_{a_i k}$
    else
        { modules k and a_i are in different sets }
        $PD_k := PD_k - C_{a_i k}$
end procedure

A uniprocessor requires $O(n)$ steps to update all the D values.

3.2.4 Executing Step P5 in Parallel

A global array *GAIN* is maintained in the shared memory of the SIMD machine. $GAIN(i)$ contains the maximum gain achievable in the i th iteration of the **for** loop in the Kernighan-Lin procedure. At the end of the **for** loop, it is required to calculate the index k which maximizes g_{max}, where g_{max} is defined as

$$g_{max} = \sum_{i=1}^{k} g_i \ . \tag{3.7}$$

Evaluation of partial sums on an SIMD architecture is very efficient [HB85]. All the sums g_{max} for $k = 1$ to $n/2$ can be evaluated in $(\log_2 n)$ steps using $n/2$ PEs. In the SIMD machine under consideration, there are n PEs. The controller deactivates $n/2$ PEs during the parallel execution of step P5. The remaining $n/2$ PEs are active

Step	Execution Count	Time for Parallel Execution	Time for Sequential Execution	Speedup
P1	α	$O(n)$	$O(n^2)$	$O(n)$
P2	$n/2$	$O(\log n)$	$O(n)$	$O(n/\log n)$
P3	$n/2$	$O(1)$	$O(1)$	$O(1)$
P4	$n/2$	$O(1)$	$O(1)$	$O(1)$
Update D values	$n/2$	$O(1)$	$O(n)$	$O(n)$
P5	α	$O(\log n)$	$O(n)$	$O(n/\log n)$
P6	α	$O(1)$	$O(n)$	$O(n)$

Table 3.1: Complexity of Parallel and Sequential Execution of *PARTITION*

during the parallel execution of step P5 ; they use the global array *GAIN* to evaluate the partial sums g_{max} and store the sums in another array *GMAX* . The maximum of the elements of *GMAX* is then calculated. A uniprocessor requires $O(n)$ time to compute the partial sums and $O(n)$ time to compute the maximum of the sums. It may be observed that the parallel implementation of step P5 results in a speedup of $O(n/\log n)$. It is easy to see that, with the EREW model of computation, this is indeed the best possible.

3.2.5 Executing P6 in Parallel

Step P6 is the *communication step* in the procedure *PARTITION* , involving movement of data. The parallel implementation of P6 depends entirely on the communication structure of a particular architecture. It must be noted that the cost of such a communication step can overshadow the speedup obtained in the other steps of the algorithm *PARTITION* . The execution of the communication step on the Orthogonal Array is described in Section 3.3. Due to the ease of data movement in the Orthogonal Array, step P6 does not become a bottleneck in the parallel implementation of the partitioning algorithm.

3.2.6 Comparison of Parallel and Sequential Partitioning

Table 3.1 gives a summary of the complexity results obtained in this section. The inner loop of the *KL* procedure is executed $n/2$ times. Table 3.1 shows that the

inner loop can be executed in $O(\log n)$ time on an SIMD architecture. As noted by the authors of [KS70], the number of times the outer loop is executed is independent of n, and is denoted by constant α. Thus, provided that the step P6 does not cost more than $O(n \log n)$, the complexity of the parallel execution of *PARTITION* on an SIMD machine is $O(n \log n)$. If the heuristic selection procedure is used in step P2, the time complexity of *PARTITION* is $O(n^2)$. This leads to the following theorem.

Theorem 1 If the communication structure of the target architecture allows the step P6 of the *PARTITION* algorithm to be implemented in $O(n \log n)$ time, the parallel partitioning scheme described above uses $O(n)$ processors and results in a speedup of $O(n/\log n)$ for step P2.

In the following section, this optimal speedup is realized on the Orthogonal Array architecture.

3.3 Kernighan-Lin on an Orthogonal Array

This section extends the parallel algorithm of Section 3.2 to execute on a VLSI architecture known as "Orthogonal Array." The details of the architecture are given in Appendix A.

3.3.1 Data Structures

The principal input to the *KL* procedure is the connectivity matrix C. There being $O(n)$ processors and $O(n^2)$ memory in the Orthogonal Array, the storage of the connectivity matrix in the memory is straight forward; element C_{ij} is stored in the memory module M_{ij} of the Array. As will be seen later, such a storage results in a very simple data movement step of P6. This storage strategy also allows maximum speedups to be obtained in the execution of the different steps of the procedure *PARTITION* . Each processor stores the identity of the module assigned to it in a *membership register M.* At the end of the partitioning procedure, the two circuit

partitions can be read off by looking at the membership registers of the processors. If the two partitions are named X and Y, then

$$module\ i\ \in\ X\ if\ membership\ register\ i \leq n/2\ . \tag{3.8}$$

$$module\ i\ \in\ Y\ if\ membership\ register\ i > n/2\ . \tag{3.9}$$

The algorithm begins with two $n/2$ partitions $\{1, 2, \ldots, n/2\}$, $\{n/2+1, \ldots, n\}$. The contents of the register M can be used by the processor PE_i to determine the membership information of module i. During the execution of procedure KL , it becomes necessary for a processor PE_i to obtain the membership information of a module $j \neq i$. This problem is resolved by having the control processor broadcast the value of the membership information of j to all the PEs. The controller maintains the membership information of all the modules in the array $MEMBER$. The other important data structures maintained by the controller are the arrays *GAIN* and *GMAX* which were described in Section 3.2.

3.3.2 Control Structures

The control structure of the parallel partitioning algorithm can be understood by looking at Figure 3.2, which shows the procedure executed by the control processor of Orthogonal Array. The instruction set of the control processor includes the vector instructions *VCOPY*, *VMAX*, *PARTSUM* , *INIT-PD* , *COMP-D* , *UPDATE-D* , and *EXCHANGE* . These instructions are explained in Table 3.2. The reader is encouraged to compare the sequential algorithm *PARTITION* with parallel algorithm of Figure 3.2. The algorithm executed by the PEs is shown in Figure 3.2.

Lemma 3 *Step P1 of PARTITION can be executed on Orthogonal Array in $O(n)$ time, with no memory conflicts.*

The vector instructions involved in parallel execution of step P1 are *INIT-PD* and *COMP-D* . Referring to the algorithms of Figure 3.2 and 3.3, the PEs zeroize their PD registers when the instruction *INIT-PD* is received. Following the broadcast

```
process CONTROL;
begin
declare array GAIN [1 .. n/2],GMAX [1 .. n/2], A [1 .. n/2], B [1 .. n/2];
declare array MEMBER [1 .. n], X1Y1 [1 .. n];
declare array MASK [1 .. n];
declare register K;
MASK ←(1, 1, ..., 1); {enable all PEs}
repeat
        bcast (VCOPY, MEMBER, X1Y1) ; { copy the partition information}
        bcast (INIT-PD ); {each PE sets PD register to 0 }
        bcast (COMP-D ); {each PE begins the computation of D values}
        for j := 1 to n do
            bcast ( MEMBER(j) ); { needed by PEs for D computation}
        for i := 1 to n/2 do begin
            A[i] ←bcast (VMAX, 1, n/2, PD);
                {PE_1...PE_{n/2} compute the maximum of their D values}
                {The Control Processor stores the maximum in A[i]}
            B[i] ←bcast (VMAX, n/2, n, PD);
                {PE_{n/2}...PE_n compute the maximum of their D values}
                {The Control Processor stores the maximum in B[i]}
            GAIN [i] ←PD[A[i]] + PD[B[i]] − 2.C[A[i], B[i]];
                    {Control Processor computes GAIN[i]}
            MEMBER [A[i]] ←1 − MEMBER [A[i]] ; {swap A [i] and B [i]}
            MEMBER [B[i]] ←1 − MEMBER [B[i]] ;
            bcast (UPDATE-D ); {PEs update the D values}
            bcast (A[i], MEMBER [ A[i] ]); {Required by PEs for UPDATED operation}
            bcast (B[i], MEMBER [ B[i] ]);
            MASK [A[i]] ←MASK [B[i]] ←0;
                    {Disable the PEs indexed by A[i] and B[i]}
        end {for}
        bcast (PARTSUM , GAIN , GMAX );{PEs compute the Partial Sums}
              {from GAIN and store these sums in GMAX }
        K ←bcast (VMAX, 1, n, GMAX );
          {PEs compute the maximum of array GMAX }
        if (GMAX[K] > 0) then {Begin Exchange operation}
          for j := 1 to K do begin
              bcast (EXCHANGE );
              bcast ( A[j], B[j] ); {Needed by PEs for EXCHANGE }
          end {for};
until (GMAX[K] ≤ 0);
end process ;
```

Figure 3.2: Control Algorithm for Parallel Partitioning

```
process PE (i);
declare register INST; {Instruction Register}
declare register I; {Identity Regster}
declare register J; {Index Register}
declare register M; {Membership Information}
begin
while (true) do
begin
        receive (INST); {Receive the Vector Instruction from CP}
        case (INST) of
            INIT-PD : PD := 0;
            COMP-D : for J := 1 to n do
                            receive (M'); {CP broadcasts MEMBER (J)}
                            if (M = M') then PD := PD + C[I, J];
                            else PD := PD - C[I, J];
            end {for}
            UPDATE-D :
                            receive (A); {CP broadcasts index of a swapped module}
                            receive (MEMB-A); {and its Membership information}
                            if (MEMB-A = M) then
                                                PD := PD + C[A, I];
                            else PD := PD - C[A, I];
                            receive (B); {CP broadcasts index of a swapped module}
                            receive (MEMB-B);{ and its Membership information}
                            if (MEMB-B = M) then
                            PD := PD + C[B, I];
                            else PD := PD - C[B, I];
            EXCHANGE : receive (A); {CP broadcasts two PE indices}
                            receive (B);
                            Participate in Row and Column Exchange; {See Text}
            VCOPY: receive ( SOURCE );receive ( DEST ); DEST [i] := SOURCE [i];
            VMAX: receive ( START-INDEX); receive ( END-INDEX );
                    receive ( SOURCE );
                    Participate in Binary Tree computation to find the Maximum of
                    SOURCE [START-INDEX] .. SOURCE [END-INDEX];
            PARTSUM : receive (SOURCE);
                            Participate in Binary Tree computation to
                            find the partial sums of array SOURCE
        end {case}
end {while}
end { process PE }
```

Figure 3.3: Process Executed by each PE of RAA

Instruction	Explanation
VCOPY	*Vector Copy.* Addresses of source and destination locations are broadcast following a *VCOPY*instruction. All processors simultaneously copy elements of source array to destination array.
VMAX	*Compute the Maximum.* The maximum of elements of an array is computed. The name of the array, the start and end indices are broadcast by the controller as inputs to this instruction.
PARTSUM	*Partial Sum.* Given an array s, it is required that processor i computes $\sum_{j=1}^{i} s_j$. The name of the array is broadcast as input to the instruction.
INIT-PD	*Initialize D-value.* Set register PD in each PE to zero.
COMP-D	*Compute D-value.* A Macro-instruction. See text for explanation.
UPDATE-D	*Update D-value.* A Macro-instruction. The inputs are the names of modules A and B selected for interchange, and their membership information. A Macro-instruction. See text for details.
EXCHANGE	*Data Movement after a Swap Operation.*

Table 3.2: Vector Instructions for Circuit-Partition

of the *COMP-D* instruction, the controller sequentially broadcasts the membership information of each of the n modules. Processor PE_i compares the received membership information of module j with the membership information of module i which is stored in register M of PE_i. The processor modifies the PD register based on the result of the comparison, as explained in Section 3.2.1.

Note that for any processor a, $1 \leq a \leq n$, the elements C_{ax} , $1 \leq x \leq n$, are all available in the row (column) a of the memory. Moreover, the memory organization of Orthogonal Array allows concurrent row-wise (column-wise) accesses by all the processors. As a consequence, the modification of the PD register requires $O(1)$ time. The modification step is executed n times for each *COMP-D* instruction. Thus, in $O(n)$ time all the D values can be computed. It may also be noted that all processors in the array can be kept busy during this step ■.

Lemma 4 *step P2 of PARTITION can be executed in* $O(\log_2 n)$ *time on the Orthogonal Array.*

As was pointed out in Section 3.2.2, a good heuristic to pick modules a_i and b_i of step P2 is to select the modules which maximize D_{a_i} and D_{b_i}. At the end of step P1, the D values are available in the individual processors. The maximum of the D values for either partition can be found in $\log_2 n$ time using the standard MAX algorithm on the Orthogonal Array [SK87]. As shown in Figure 3.2, the vector instruction $VMAX$is used for this purpose.

Lemma 5 *The D values can be updated in $O(1)$ time on the Orthogonal Array.*

After swapping module a_i and b_i in step P4, the D values are updated using the following equations:

$$D_x \leftarrow D_x + 2.C_{xa_i} - 2.C_{xb_i},\ x \in\ X1 - \{a_i\} \tag{3.10}$$

$$D_y \leftarrow D_y + 2.C_{yb_i} - 2.C_{ya_i},\ y \in\ Y1 - \{b_i\} \tag{3.11}$$

The vector instruction *UPDATE-D* is used by the Othogonal Array for updating the D values. Following the broadcast of *UPDATE-D* , the controller also broadcasts the module number and membership information of the two swapped modules a_i and b_i. After a PE has received all this information, the updating of the D value is affected by a modification of the register PD as explained in Section 3.2.3. Referring to Figure 3.3, the execution of the instruction *UPDATE-D* requires $O(1)$ time. It is important to note that no memory conflicts arise during this step ■.

Lemma 6 *Step P5 of PARTITION can be executed in $O(\log_2 n)$ time on Orthogonal Array.*

Recall that during step P5, the partial sums $\sum_{i=1}^{k} g_i$ are computed and the value of k which maximizes the partial sum is chosen. The gains g_i are evaluated by controller after step P2, and stored in the array *GAIN* . Given this set up, step P5 reduces to the following standard computations on the orthogonal array:

1. Evaluation of partial sums

2. Calculating of the maximum element

Each of these standard operations requires $O(\log n)$ time on the orthogonal array [AK87, SK87]. The vector instruction *PARTSUM* is used by the Orthogonal Array to evaluate the partial sums of the array *GAIN* . The partial sums are stored in the array *GMAX* . As parameters of the instruction *PARTSUM* , the controller broadcasts the addresses of *GAIN* and *GMAX* to the PEs. The instruction *VMAX*is used to compute the maximum of the array *GMAX* $[1..n]$ ■.

Lemma 7 *The parallel implementation of step P6 of PARTITION requires O(1) time on Orthogonal Array.*

Vector instruction *EXCHANGE* is used by Orthogonal Array to implement the data movement involved in step P6. The implementation details of the *EXCHANGE* instruction are now briefly reviewed, taking as example the case where modules a_i and b_i are exchanged, $1 \leq i \leq k$. It may be noted that these k exchanges can all be made concurrently. The controller broadcasts the module numbers of a_i and b_i immediately after the broadcast of *EXCHANGE* instruction. The exchange of a_i and b_i can be simply affected by using the memory module $M_{a_i b_i}$ as a mail box. The processor PE_{a_i} writes the contents of its membership register M into a prespecified location in $M_{a_i b_i}$. At the same time, PE_{b_i} writes the contents of its membership register into in $M_{b_i a_i}$. The processors use their identity register to find out which memory location they must use. At the end of this *send* step, each of the involved processors does a *receive* step, where it picks up the mail sent by the other processor. Thus an *exchange* requires $O(1)$ time ■.

3.3.3 Data Movement during *EXCHANGE* Operation

In what follows, the implementation of *EXCHANGE* is illustrated with an example. Consider a 6 module problem, where module i assigned to PE_i, $1 \leq i \leq 6$. For this assignment, the storage of the connectivity matrix in the memory of the Orthogonal Array is shown in Figure 3.4. Suppose that the the partition at the end of the first iteration of the outer loop of the *KL* procedure is {5, 2, 3} | {4, 1, 6}. The connectivity matrix needs to be reorganized as in Figure 3.6. It has been shown in

$$\begin{pmatrix} C_{11} & C_{12} & C_{13} & C_{14} & C_{15} & C_{16} \\ C_{21} & C_{22} & C_{23} & C_{24} & C_{25} & C_{26} \\ C_{31} & C_{32} & C_{33} & C_{34} & C_{35} & C_{36} \\ C_{41} & C_{42} & C_{43} & C_{44} & C_{45} & C_{46} \\ C_{51} & C_{52} & C_{53} & C_{54} & C_{55} & C_{56} \\ C_{61} & C_{62} & C_{63} & C_{64} & C_{65} & C_{66} \end{pmatrix}$$

Figure 3.4: Initial C-Matrix Storage in Memory Array

$$\begin{pmatrix} C_{51} & C_{52} & C_{53} & C_{54} & C_{55} & C_{56} \\ C_{21} & C_{22} & C_{23} & C_{24} & C_{25} & C_{26} \\ C_{31} & C_{32} & C_{33} & C_{34} & C_{35} & C_{36} \\ C_{41} & C_{42} & C_{43} & C_{44} & C_{45} & C_{46} \\ C_{11} & C_{12} & C_{13} & C_{14} & C_{15} & C_{16} \\ C_{61} & C_{62} & C_{63} & C_{64} & C_{65} & C_{66} \end{pmatrix}$$

Figure 3.5: C-Matrix after Row Exchange

[RS88] that such reorganization can be done in two well defined steps, namely, *row exchange* and *column exchange.*

The data movement after the *transposition* of modules 1 and 5 is explained with reference to Figures 3.4 - 3.6. There are two steps in which such a data movement is done:

1. Interchange the *rows* 1 and 5 in the memory (Row Exchange)

2. Interchange the *columns* 1 and 5 in the memory (Column Exchange).

Figures 3.5 and 3.6 show the contents of the memory after the execution of steps 1 and 2 above.

Row and column exchanges require $O(1)$ time each, and both steps have the property that they do not involve memory conflicts. The lemmas 3,4,5 and 6 can be used to state the following theorem.

Theorem 2 *With the exception of steps P2 and P5, a speedup of O(n) is obtained by executing the individual steps of the KL partitioning algorithm on the orthogonal*

$$\begin{pmatrix} C_{55} & C_{52} & C_{53} & C_{54} & C_{51} & C_{56} \\ C_{25} & C_{22} & C_{23} & C_{24} & C_{21} & C_{26} \\ C_{35} & C_{32} & C_{33} & C_{34} & C_{31} & C_{36} \\ C_{45} & C_{42} & C_{43} & C_{44} & C_{41} & C_{46} \\ C_{15} & C_{12} & C_{13} & C_{14} & C_{11} & C_{16} \\ C_{65} & C_{62} & C_{63} & C_{64} & C_{61} & C_{66} \end{pmatrix}$$

Figure 3.6: C-Matrix after Column Exchange

array machine. For the steps P2 and P5, a speedup of O $(n/\log n)$ is obtained, which is optimal.

3.4 Kernighan-Lin on Alliant FX/80

The Kernighan-Lin algorithm for circuit partition has been implemented on the Alliant FX/80 a shared-memory multiprocessor. Appendix A describes the salient features of the Alliant FX/80. The machine has up to 8 computational elements and 12 interfactive processors. Each computational element is a microprogrammed, pipelined processor and supports both scalar and vector instructions. The computational elements are connected by means of a high speed bus known as the *concurrency control bus.* The computational elements share a common bank of memory and a high-speed cache. A computation can be executed in a mixed concurrent-vector mode on the Alliant machine. Thus, the job can be split into concurrent processes and executed on the computational elements. Each of these concurrent processes can further exploit the parallelism in the computation and execute in the vector mode. The interactive processors are intended for handling I/O – user terminals, disks, tapes and network traffic. However, each interactive processor supports the full instruction set of the Motorola 68020 processor and can execute user jobs in a scalar mode. The "execute" command of the Concentrix Operating System on the Alliant has an option to specify which type of processor the user job should be run on. The user can also specify the number of computational elements to be used to

run the job. The operating system takes the responsibility of scheduling tasks and balancing the load.

The partition program was implemented in FX/Fortran [All87]. FX/Fortran is an extension of Fortran 77 and permits concurrency and vector operations. In addition, the FX/Fortran compiler has the capability to parallelize programs that are written for a scalar machine. The important details of the FX/Fortran compiler are summarized in Appendix A.

The implementation of the Kernighan-Lin algorithm closely follows the algorithm shown in Figure 3.1. The program can be used for graph bisection in general and two-way circuit partition in particular. The connectivity matrix is stored in a two-dimensional array C. The user has the option to specify an initial partition. Otherwise, the program generates a random initial partition using the following technique. A random permutation of $1 \cdots n$ is generated in the array *perm*. The circuit modules numbered $perm(1)$, $\cdots$, $perm(n/2)$ are placed in the set X. The remaining modules are assigned to set Y. The output of the program is the pair of sets X and Y, both of which are implemented as integer arrays of dimension $n/2$. The parallelization of various steps in the algorithm follows the ideas illustrated in Section 3.2, except that a fixed number of processors are available in the target machine. Instead, advantage is taken of the vector capabilities of each processor to achieve parallelism.

The program was tested against circuits provided by Saab and Rao [SR90]. In these circuits, problem sizes range from 20 cells up to 350 cells. The results are shown in Table 3.3. We observed that for the same problem instance, the execution time varied considerably from one run to another. This is because the random initial solution is different for every execution. The initial solution affects the execution time in two principal ways. First, the number of iterations of the **repeat** loop are determined by the initial solution. Secondly, the memory access patterns depend on the initial solution. For example, consider the calculation of $E(a)$ for a module a. This involves the access of all connectivity terms of the form $C(a, b)$, where b is a module in the opposite block. Since the initial partition determines the opposite

Circuit Name	n	Solution	Sequential Time (sec)	Time on 1 CE (sec)	Time on 2 CEs (sec)	Time on 3 CEs (sec)	Time on 4 CEs (sec)
inp20.1.1	20	75	.0493	.0361	.0345	.0365	.0373
inp50.1.1	50	453	.5520	.2260	.2079	.2066	.2066
in100.1.1	100	1866	4.4225	1.9711	1.6284	1.5914	1.059
in150.1.1	150	4255	22.345	6.0619	5.0037	4.5286	4.5122
in200.1.1	200	7626	44.532	18.164	14.263	13.605	9.886
in300.1.1	300	17221	125.69	56.274	34.642	42.388	31.397
in350.1.1	350	23469	313.77	133.50	75.79	67.07	63.98

Table 3.3: Results of Kernighan-Lin on Alliant FX/80

block, it is clear that different $C(a, b)$ are requested for computing $E(a)$ during different executions of the program. The connectivity matrix is stored in the shared memory and a cache is used to access elements from the shared memory. The access sequence plays a significant role in determining the performance of cache as well as interleaved memory. In order to overcome this problem while reporting results, each experiment was repeated several times and the average execution time was used. Figure 3.7 shows the variation of speedup and execution time of the parallel partition algorithm as a function of the P, the number of computational elements. The case $P = 0$ corresponds to a sequential scalar implementation. The speedup is seen to deteriorate for $P = 3$. This is chiefly due to the fact that associative computations are inefficient on an odd number of processors. Figure 3.8 plots the variation of execution time as a function of the problem size n, using P as a parameter.

3.5 Summary

When a circuit is too large to be built into one chip, logic partition is used to subdivide the circuit into smaller subcircuits. The partition must satisfy the *constraint* that the subcircuits do not contain more logic modules than a specified number. The *objective* of the problem is to minimize the external wiring in the partition. Two-way partition is the simplest case of the problem. if a circuit must be divided into k parts, then two-way partition must be used recursively until k subcircuits are obtained. The size constraint translates to the *balance constraint* in the two-way partition problem – the two blocks of the partition must be nearly of the same size.

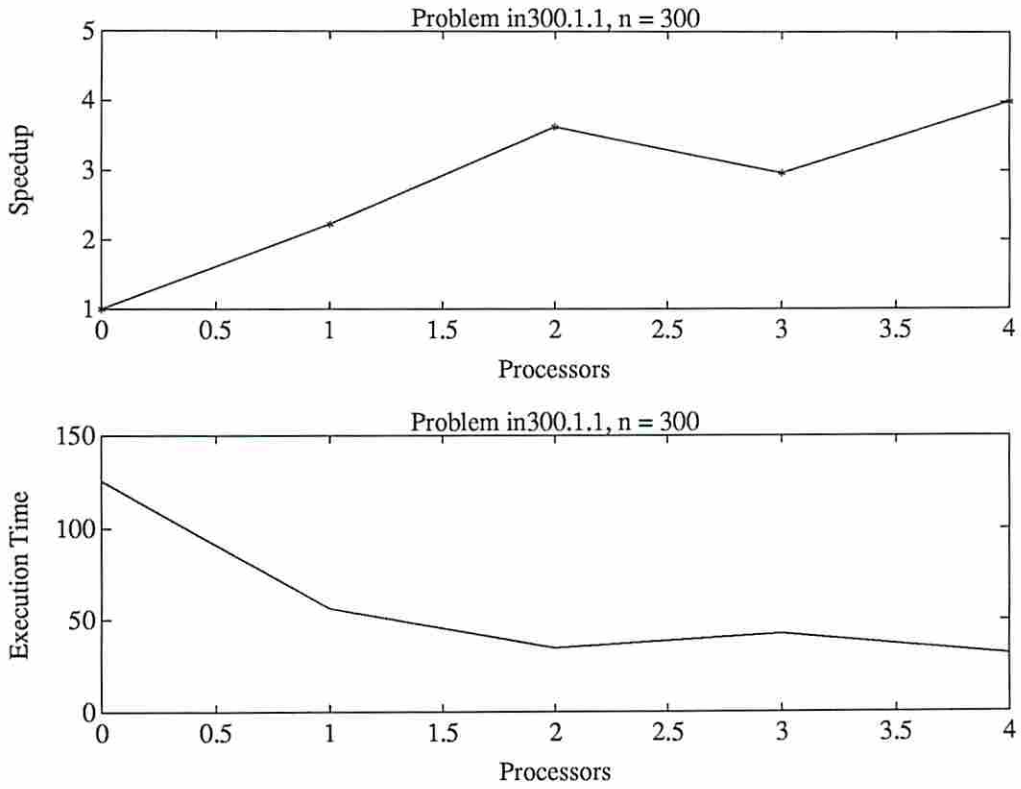

Figure 3.7: Speedup and Execution Time of the Parallel Partition Algorithm as a function of number of Computational Elements.

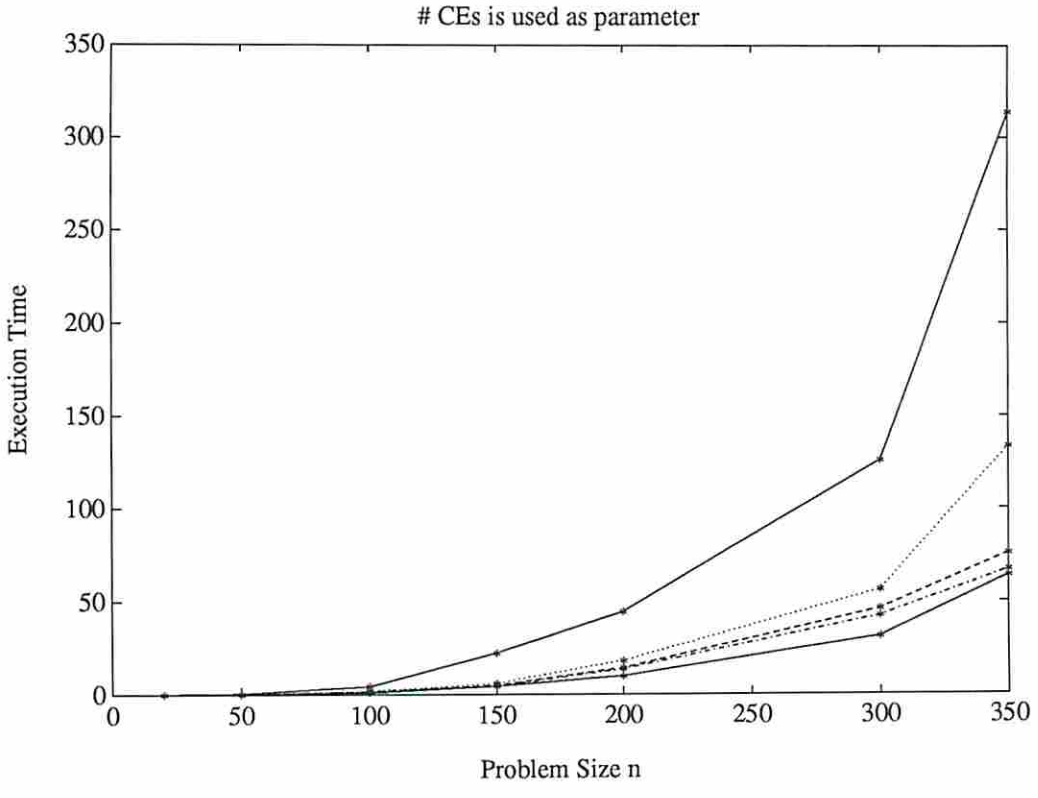

Figure 3.8: Execution Time of the Parallel Partition algorithm as a function of Problem Size. The continuous curve on top corresponds to scalar processing on 1 CE. The dotted curve corresponds to vector processing on 1 CE. The dashed curve, the curve with mixed dot and dash symbols, and the solid curve on the bottom correspond to 2,3 and 4 computational elements respectively.

The Kernighan-Lin algorithm is a well known heuristic algorithm for two-way circuit partition. This chapter has examined parallel implementations of Kernighan-Lin algorithm. A PRAM algorithm was developed for Kernighan-Lin, which was then extended to execute on an Orthogonal Array architecture. An Alliant FX/80 implementation of Kernighan-Lin was reported.

The partition problem has several other applications other than circuit packaging. Two-way circuit partition arises as a subproblem in min-cut placement, which is discussed in the next chapter. The *task allocation* problem, which must be solved by a multi-tasking operating system, is a variation of the circuit-partition problem; the input to the task allocation problem is a set of tasks which communicate among one another. The tasks must be assigned to processors such that the loads on the processors are balanced and the inter-processor communication is minimized. In conclusion, the partition problem is a problem of great practical significance and parallel algorithms to solve the problem find applications in several domains. Recently, the partition problem has been approached using several other approaches such as artificial neural networks (Chapter 7), Simulated Annealing [GS84], and Simulated Evolution [SR90]; a fruitful direction for future research is to use parallel computation for these techniques.

Chapter 4

Parallel Circuit Placement

Chapter 1 has introduced the problem of circuit placement. Section 1.6.3 discussed the computational complexity of the problem. In this chapter, we shall consider parallel algorithms for circuit placement. Since there are many parallel algorithms to be considered, it is useful to classify them and study them under broad categories. Section 4.1 provides a taxonomy of parallel placement algorithms based on two factors, namely, the parallel architecture used and the placement algorithm employed. The latter classification is used in the rest of the chapter.

Section 4.2 considers parallel placement by *iterative improvement*. Iterative improvement, also referred to as local search or greedy search, is often used to solve combinatorial optimization problems that arise in VLSI design. The technique consists of starting with some initial solution to the problem and improving it iteratively until no further improvements can be made. Section 4.3 considers parallel Min-cut placement. The Min-cut technique is the best known heuristic for minimizing the maximum wiring density in a VLSI chip. The technique is popularly used in the layout of gate array chips. The procedure begins by partitioning the circuit into two subcircuits such that the interconnect wiring between the subcircuits is minimized. If the two subcircuits are imagined to be placed in two adjacent regions of the VLSI chip and a cut-line that divides the two regions, then the Min-cut technique has essentially minimized the wiring density across the imaginary cut-line. The procedure is then recursively applied to the two subcircuits, until each subcircuit is an individual logic module.

In Section 4.4, we shall consider a parallel placement technique based on the divide-and-conquer paradigm. The technique is useful for the layout of gate array chips as well as standard cells. It consists of partitioning a large circuit into several small subcircuits and then solving the placement problem on each of the subcircuits. Clustering techniques are useful in the division phase of the algorithm. The motivation for dividing the circuit into small subcircuits is the fact that small placement problems can be solved more accurately than large ones. For example, if the problem size is less than 20, the placement problem can be solved exactly by exhaustive search or semi-enumerative techniques such as branch-and-bound. After the subcircuits have been placed, the divide-and-conquer algorithm attempts to further improve the solution through global movement of logic modules across subcircuits. In Section 4.5, we describe a parallel implementation of a placement algorithm based on a numerical optimization technique. The linear placement problem is formulated as a minimization of a quadratic form z which can be expressed in the form $z = X'BX$. Here B is a positive semi-definite matrix which is derived from the connectivity matrix C. X is a row vector of the coordinates $x_1, x_2, \cdots, x_n$ of the n modules to be placed in a row. The minimization of z is reduced to the problem of finding the eigenvector X of matrix B which minimizes z. The minimum eigenvalue represents the minimum cost function. We present an implementation of the algorithm on the Alliant FX/80.

Finally, in Section 4.6, we consider parallel placement by Simulated Annealing. In the recent past, perhaps no other algorithm for combinatorial optimization has generated as much interest as has Simulated Annealing. Annealing is the process of slowly cooling a molten substance (e.g. metal) so as to condense the substance into fine crystalline structure. The Simulated Annealing algorithm is based on the analogy between the slow cooling of a substance and the process of discrete combinatorial optimization. The solution to a discrete optimization problem is often described as a state. For example, in the circuit placement problem, the solution consists of assigning a slot position to each of the n logic modules. A frequently used technique for optimization is to start with some initial state and make changes

to the state in the hope of improving the cost function. For example, the iterative improvement technique considers a small subset of the state transitions, finds the transition which brings about the best improvement to the cost function and applies the transition to the current state. Such a greedy technique has been compared to rapid cooling of substances. Rapid cooling results in brittle structures, just like iterative improvement results in local optima. The key to obtaining crystalline structure is, therefore, slow cooling. The concept of temperature plays a major role in the Simulated Annealing algorithm. During the annealing of a substance, the energy possessed by any molecule is directly proportional to its temperature. At high temperatures, molecules are free to move about randomly. On the other hand, at temperatures close to absolute zero, the movement of the molecules is highly restricted. An analogy can be drawn between movement of molecules and a state transition. At high temperatures, the Simulated Annealing algorithm permits state transitions which are essentially random. In other words, the algorithm does not insist on transitions that necessarily improve the cost function. As the temperature is slowly reduced, the algorithm becomes more and more unfavorable towards transitions that do not improve the cost function. Perhaps the biggest advantage of Simulated Annealing is that it is easy to implement. Experience has shown that the solutions obtained using the technique are of superior quality. The biggest disadvantage of the technique is that it is extremely CPU-bound. As a result, using parallel processing to enhance the performance of Simulated Annealing algorithms is of significant practical interest.

4.1 A Taxonomy of Parallel Placement Systems

Over the period of last ten years, a number of parallel placement systems have been proposed and several have been actually built. It is useful to classify these systems based on two criteria:

1. the particular architecture involved, and
2. the placement algorithm implemented.

Architecture	Interconnection Network	Placement Algorithm	S/P or G/P?	Ref.
MIMD	Shared Memory	Simulated Annealing	G/P	[C+87]
			G/P	[Kra87]
	Distributed Memory (Hypercube)	Simulated Annealing	G/P	[Jay87]
		Divide-And-Conquer	G/P	[RS89b]
SIMD	Orthogonal Array	Min-cut	S/P	[RSP91]
		Divide-And-Conquer	S/P	[RS88]
	Adaptive Array 2-D Mesh	Iterative Improvement	G/P	[SW84]
			S/P	[CB83]
			S/P	[U+83]
	Linear Array		G/P	[RSP88]
PIPELINE	Large Grain	Simulated Annealing	S/P	[RP87, RP90]
	Small Grain	Iterative Improvement	S/P	[I+83]

Table 4.1: Classifying Placement Accelerators on Target Architecture. G/P and S/P are used to indicate General-Purpose and Special-Purpose, respectively.

Table 4.1 presents a classification based on the target architecture of the accelerator : SIMD, MIMD, or pipeline. Under each of these categories, further classification is possible based on other architectural considerations such as the interconnection network. For example, under the category MIMD, two broad groups of machines exist, namely, shared memory multiprocessors and distributed memory multiprocessors. Table 4.1 reflects this finer classification. For each entry in the table, a bibliographical citation is provided to a publication which describes the particular accelerator in detail.

Table 4.2 illustrates an alternate classification of placement accelerators based on the underlying placement algorithm. This classification, based on the placement paradigm employed by the parallel placer, is used in the rest of this chapter. The remaining sections discuss salient features of placement accelerators based on four different algorithms : iterative improvement, Min-cut, Divide-and-Conquer, and Simulated Annealing. Instead of machine details, emphasis will be placed on the study of parallelism in different placement algorithms as well as the techniques used to map these algorithms onto different parallel architectures.

Placement Algorithm	Architecture	Interconnection Network	S/P or G/P?	Ref
Local Search	SIMD	Adaptive Array	G/P	[SW84]
		2-D Mesh	S/P	[CB83]
			S/P	[U+83]
		Linear Array	G/P	[RSP88]
	PIPELINE	Small Grain	S/P	[I+83]
Min-cut	SIMD	Orthogonal Array	S/P	[RSP91]
Divide-And-Conquer	SIMD	Orthogonal Array	S/P	[RS88]
	MIMD	Hypercube	G/P	[RS89b]
Simulated Annealing	MIMD	Shared Memory	G/P	[C+87]
			G/P	[Kra87]
		Hypercube	G/P	[Jay87]
	PIPELINE	Large Grain	S/P	[RP87, RP90]

Table 4.2: Classifying Placement Accelerators on Underlying Algorithm. G/P and S/P are used to indicate General-Purpose and Special-Purpose, respectively.

4.2 Parallel Iterative Improvement

Iterative Improvement is a popular heuristic for many combinatorial optimization problems. Figure 4.1 illustrates the algorithm as applied to a minimization problem. S is the "current solution" and is initialized to S_0. The initial solution S_0 is obtained using a constructive procedure or by randomization. A local neighborhood of the current solution S is then searched for better solutions. The concept of local neighborhood is best understood with the example below. The **for** loop in Figure 4.1 considers all the solutions in the local neighborhood of S. If the best of these solutions gives a positive improvement, it is accepted by the algorithm as the "current solution." The search is then repeated. The procedure halts if none of the solutions in the local neighborhood is better than the current solution. The cost of the current solution improves over each iteration of the **repeat** loop – hence the name "iterative improvement." Since the procedure always moves to the best of the local neighborhood, it is also known as *greedy search*. Again, the procedure is known as *local search* since it searches in the local neighborhood of the current solution. The final solution obtained by the procedure is known as the local optimum solution.

Example 4.1 :

Consider the linear placement problem, where a set of n logic modules must be placed into n slots in a row. For this example, let $n = 5$. An initial placement S_0 can be easily found by assigning module i to slot i. Thus the initial solution is 1 2 3 4 5. *Pairwise interchange* is defined as the operation of picking two modules and interchanging their positions. For example, interchanging modules 1 and 3 in the initial placement generates the new solution 3 2 1 4 5. $\binom{5}{2} = 10$ pairs can be formed from 5 logic modules. A local neighborhood of S_0 consists of the 10 solutions that can be generated by the 10 possible pairwise interchange operations. A 3-way interchange, defined as the round-robin rotation of three modules, generates a different local neighborhood.

A variation of the iterative improvement procedure is illustrated in Figure 4.2. This version of the procedure is used when the size of the local neighborhood is very large and therefore it is impractical to enumerate the entire neighborhood. For example, the pairwise interchange operation of Example 4.1 generates a local neighborhood of size $\binom{n}{2}$ – prohibitively large when n is of the order of 1000. Since n is indeed large for VLSI problems, unless mentioned otherwise, our reference to iterative improvement will mean the latter algorithm (Figure 4.2).

The procedure *Local-Search* begins with an initial solution S_0. The current solution is called S. The **repeat** loop of the procedure consists of three important steps. Step R_1 applies a local change to the current solution S. This step is also known as a "move" in the literature and results in a local neighborhood solution S'. Step R_2 of computes the resulting change in cost function from S to S'. The new solution S' is immediately accepted if it is better than the current solution (Step R_3). If S' is inferior to S, more of the local neighborhood of S is searched. There is a chance that the same neighborhood solution is examined more than once. However, this probability is small since the local-neighborhood is large in size. The variable *reject* counts the number of successive rejections. The **repeat** loop is executed until a predefined number of rejections occur (constant β in Figure 4.2). The algorithm

```
procedure Iterative-improvement;
begin
   S = S0              /* Initial solution */
   repeat
      BestImprovement = −∞;
      BestNextsolution = S;
      for each S' in the local neighborhood of S do
         if (cost(S) − cost(S') > BestImprovement) then
          begin
              BestImprovement = (cost(S) - cost(S'));
              BestNextsolution = S'
          end ;
      if (BestImprovement > 0) then
          S = BestNextsolution;
    until (BestImprovement ≤ 0);
end procedure ;
```

Figure 4.1: Iterative improvement algorithm for combinatorial minimization.

is said to have arrived at a local optimal solution when the **repeat** loop terminates. The larger the value of β, the more likely that a local optimum has been reached.

It may be advantageous to start with a good solution before the local search is initiated. There is no theoretical result to support this statement. However, for many problems such as placement, traveling salesperson, and circuit partition, it has been empirically observed that a good starting solution helps in improving the final quality of the solution as well the running time of the procedure [Got81, HK72].

The procedure *Local-Search* of Figure 4.2 is simple to implement and has modest memory requirements. On the other hand, the computational requirement of the procedure is quite large. The **repeat** loop is executed $\alpha + \beta$ times, where α is the number of iterations required to move from S_0 to the local optimal solution. When pairwise interchange is used, β should be at least equal to $\binom{n}{2}$. α depends on several factors : the initial solution itself, the perturbation method, and the final solution. Assuming that $\alpha \leq \beta$, the sequential running time of the procedure *Local-Search* is given approximately by Equation 4.1.

$$T_S(n) \leq 2 \cdot \beta \cdot (T_1 + T_2(n) + T_3) \tag{4.1}$$

```
procedure Local-Search;
begin
        S := S_0;      /* S_0 is the initial solution */
        reject := 0;
        stop := false ;
        repeat
R_1:            S' := perturb(S);     /* Local Modification */
R_2:            δ_C := C(S') − C(S);
R_3:            if (δ_C < 0) then begin
                   S := S' ;              /* accept the new configuration */
                   reject := 0;
                end
                else begin
                   reject := reject+1;
                   if (reject = β) stop := true ; /* stop the repeat loop */
                end
        until (stop = true ) ;
end procedure
```

Figure 4.2: A Variation of Iterative Improvement Algorithm as applied to a minimization problem.

In the above equation, n is the problem size and T_i is the time requirement of step R_i, $1 \leq i \leq 3$. Notice that T_1 does not depend on n, since step R_1 makes a local modification to the current solution. Similarly, T_3 is also independent of the problem size. The running time T_S may be further approximated to obtain

$$T_S(n) = \beta \cdot T_2(n) \tag{4.2}$$

As mentioned earlier, the constant β should be chosen large enough such that there is a high degree of confidence that the local search has indeed terminated at a local optimum. This shows why iterative improvement is a compute intensive algorithm. There is another variation of iterative improvement. Since local search can get stuck at a local optimum, it is possible to run the iterative improvement phase several times starting with different initial solutions. The best of the local optima is taken as the the final solution. If ω initial solutions are attempted, it is easy to see that the running time of the algorithm increases by a factor of ω.

$$T_S(n) = \omega \cdot \beta \cdot T_2(n) \tag{4.3}$$

Looking at Equation 4.3, it may be concluded that the following techniques are useful in obtaining execution efficiency.

1. A straight forward technique is to concurrently execute all the ω instances of the iterative improvement algorithm. The time required to compare the ω solutions is trivial when compared to the factor $\beta \cdot T_2(n)$. Therefore, this scheme results in a speedup close to ω. The number of processors required in this scheme is ω. Since each processing element is an instruction set processor, the resulting architecture is MIMD.

2. Speedup techniques may be employed to reduce $T_2(n)$, the time to evaluate the change of cost after a state transition. This approach has been implemented in [I+83], where Iosupovici *et al* use iterative improvement algorithm for PCB placement. A pipelined hardware module is used to evaluate the change in

wire length after a pairwise interchange has been carried out. Section 4.2.4 describes the technique in detail.

3. It may be possible to decrease the number of iterations taken by the local search algorithm to converge to a local optimum. This scheme is called *Parallel Iterative Improvement.* There are many variations of parallel iterative improvement, as discussed in Section 4.2.1.

4.2.1 Placement by Parallel Iterative Improvement

The constant β in Equation 4.3 represents the expected number of perturbations required by the local search procedure to "make sure" that a local optimum has been reached. Given a state S and a perturbation function, there are a finite number of states which can be reached starting from S and applying the perturbation function. For example, consider that the placement problem requires that n logic modules be arranged in a linear order such that some cost function is minimized. There are $n!$ possible permutations of the n modules. Starting at one of these permutations, there are precisely $\binom{n}{2}$ permutations which can be reached by applying a pairwise interchange. This shows that $\beta = \Omega(n^2)$. Larger values are needed in practice to "ensure" convergence.

In order to obtain faster convergence, an iterative improvement algorithm must parallelize the loop consisting of *perturb*, *evaluate*, and *decide* operations. Accordingly, the existing iterative placement methods are classified as

1. SPSD - Single Perturbation, Single Decision

2. MPSD - Multiple Perturbations, Single Decision

3. MPMD - Multiple Perturbations, Multiple Decisions.

The SPSD scheme is suitable for sequential implementation, where a single processor perturbs the placement and decides whether or not to retain the new placement. In the MPSD scheme, several processors concurrently perturb the placement, but a

single processor evaluates the cost of the new placement and carries out the decision step. The MPMD approach uses multiple processors to asynchronously perform both perturbation and decision steps. It is clear that the MPMD approach offers the largest degree of parallelism, and is suitable for a multiprocessor implementation. The MPSD scheme offers partial parallelism, and lends itself to an SIMD implementation.

There are disadvantages in using the MPMD scheme. When two different processors simultaneously make moves, it may happen that the individual moves are both good, but the combined effect of the two moves is detrimental. An *erroneous decision* is said to have been made by the two processors. The iterative improvement algorithm is essentially a greedy heuristic, where only better solutions are accepted. Thus, if the cost value of the "current" placement is observed as the algorithm progresses, the resulting curve must show a monotonic decrease. But due to erroneous decisions, the cost curve may exhibit a non-monotonic behavior. A more serious problem is the *oscillation problem*, a situation in which a number of processors perform endless exchange of module positions. The parallel placement approaches in [CB83] and [U+83] are MPMD.

At the expense of partial loss of parallelism, the problems of erroneous decisions and oscillations can be overcome by adopting the MPSD approach. Since the accept/reject decision is *serialized* i.e., made by a single processor rather than several individual processors, there is no erroneous decision, and hence, no oscillation. The placement scheme in [RSP88] and [SW84] are MPSD.

4.2.2 Concurrent Placement on Array Processors

Two of the early attempts to parallelize iterative improvement algorithms for module placement are reported in [CB83] and [U+83]. Since the spirit of these two algorithms is essentially the same, only one of them, namely [CB83], is outlined. Chyan and Breuer [CB83] use a mesh-connected array of N processors for the placement of N modules (see Figure 4.3). One module is assigned to each processor in the array. The following steps are executed in parallel on the mesh of processors.

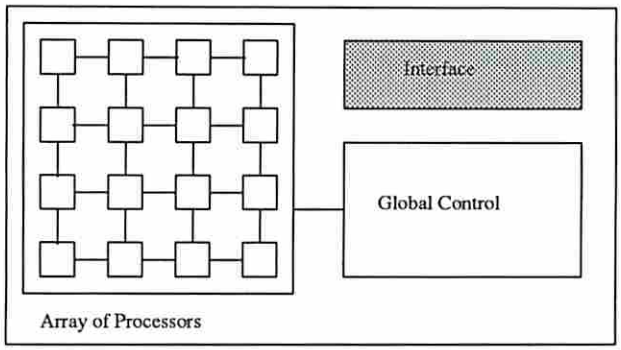

Figure 4.3: Array Processor for Local Search Placement

1. Examining all pairs of adjacent modules for possibility of interchange. This is done in four cycles. A module placed at coordinates (i, j) in the two-dimensional plane has four neighbors positioned at $(i, j+1)$ (North) $(i+1, j)$ (East) $(i, j-1)$ (West), and $(i-1, j)$ (South). Four different cycles are reserved to examine module-pairs of the form $\{(i, j), (i, j+1)\}$ (North) $\{(i, j), (i+1, j)\}$ (East) $\{(i, j), (i, j-1)\}$ (West), and $\{(i, j), (i-1, j)\}$ (South). The near-neighbor mesh interconnection suffices to form these pairs.

2. A processor which examines a pair of modules (p, q) decides whether or not to swap the modules independently of the other processors. Thus, in our classification, the parallel optimization technique is MPMD. A pair of modules is marked for interchange if the potential gain of the interchange is positive i.e. the total wire length decreases if the modules are interchanged.

3. Actual interchanges. For each pair chosen for interchange, the data corresponding to the two modules is exchanged among the corresponding processors.

After steps 1,2, and 3, a global update phase is performed, where all the processors are informed of the interchanges. A broadcast bus is used for this purpose. A processor which contains a swapped module p broadcasts the new location of p over the bus. All processors whose modules are connected to p use this information to update their local storage. Since there can be several swapped modules in one iteration, the broadcast phase must be repeated several times. Together with the global update phase, steps 1,2, and 3 form one iteration of the algorithm. The total

wire length is expected to decrease over every iteration. The algorithm is terminated after a predefined number of iterations have been executed. Except for the global update phase which is carried out sequentially, all the other steps in an iteration can be done in parallel. The complexity of a global update phase is proportional to the number of interchanges performed the iteration. The number of interchanges decreases over iterations. It is therefore complicated to analyze the speedup of the parallel algorithm. A rough analysis presented in [CB83] shows that the speedup achieved by the parallel algorithm is close to P, the number of processors.

4.2.3 Placement on Adaptive Array

An interesting implementation of a *steepest-descent directed pairwise interchange* placement (SDDPI) is reported in [SW84]. The SDDPI method is an improvisation of the local search algorithm, and has been mapped onto an adaptive array processor. An adaptive array is a two-dimensional array processor intended as a general-purpose parallel machine. It consists of a 256 $\times$ 256 array of 1-bit processors. The array is *reconfigurable* e.g., it can be used as a smaller array of 2048 processors each of which has a 32-bit data path.

During any iteration of the SDDPI algorithm, one of the N modules is selected by the controller as a *reference* module M. An individual processor assigned to module M_i computes the expected reduction in wire length D_i when modules M and M_i are interchanged. The module P with the maximum value of D_i is interchanged with M. To be able to compute D_i, processor i requires the connectivity information of the modules connected to M_i. This information is stored in the local memory of processor i.

The authors of [SW84] experimentally observed a linear growth of speedup as the problem size N is increased.

4.2.4 Pipelined Cost Evaluation

Iosupovici et al. [I+83] have described a special-purpose architecture to compute the incremental wire length Δ_L (Figure 4.4). This machine uses a 4-stage pipeline to speed up the evaluation of Δ_L. Module interchange is used in the perturbation of a placement. A central controller is responsible for both module interchange as well as accept/reject decision. Consider a move which consists of interchanging two modules i_1 and i_2. Let $\mathcal{P}$ and $\mathcal{Q}$ be the placement vectors before and after the perturbation. It can be shown that the incremental change in the aggregate wire length is given by

$$\Delta_L = \sum_{j=1}^{N} C_{i_1 j} \times (\mathcal{D}_{\mathcal{Q}_{i_1}\mathcal{Q}_j} - \mathcal{D}_{\mathcal{P}_{i_1}\mathcal{P}_j}) + \sum_{j=1}^{N} C_{i_2 j} \times (\mathcal{D}_{\mathcal{Q}_{i_2}\mathcal{Q}_j} - \mathcal{D}_{\mathcal{P}_{i_2}\mathcal{P}_j}) \tag{4.4}$$

The key idea in [I+83] is to recognize that this sequence of arithmetic operations (subtraction, multiplication, and summation) can be efficiently pipelined. The salient features of the pipeline are summarized below.

The placement information is stored as a table in stage (1), from which it supplies stage (2) with a stream of module numbers. These are the identification numbers of modules connected to i_1 and i_2. Stage (1) also supplies the connectivity information $C_{i_1 j}$ and $C_{i_2 j}$ to stage (3). Stage (2) stores the *distance matrix* D. Given slot positions $\mathcal{P}_i$, $\mathcal{P}_j$, $\mathcal{Q}_i$, and $\mathcal{Q}_j$, stage (3) computes the value $\Delta_d = (\mathcal{D}_{\mathcal{P}_i\mathcal{P}_j} - \mathcal{D}_{\mathcal{Q}_i\mathcal{Q}_j})$ and passes it as input to stage (3). Stage (3) multiplies Δ_d and the connectivity value received from stage (1). The output of stage (3) is passed on to by stage (4), which accumulates its inputs to form the final value of Δ_L.

4.2.5 A Data Parallel Approach

The special-purpose engines (SPEs) discussed above are designed as back-end attachments to uniprocessor hosts. SPEs are limited in power by the number of processors built into them at the time of design. When the problem size increases above this limit, there is no obvious way to adapt an SPE to cater for the new situation. An

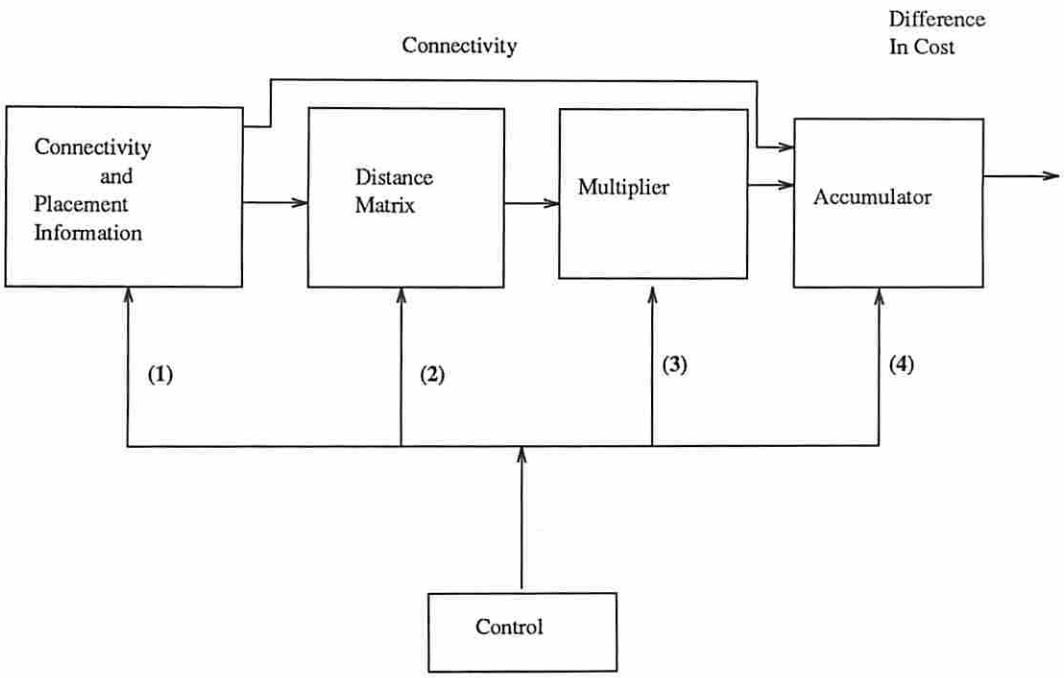

Figure 4.4: Pipelined Cost Evaluation

SPE also suffers from inflexibility. If a design algorithm becomes out of date due to change in design style, so does the SPE built for the algorithm. With the availability of massively parallel general-purpose computers, is possible to overcome these drawbacks. This feature has been exploited in [RSP88], where a *data parallel* approach is proposed to speed up local search placement algorithms.

The parallel algorithm of [RSP88] is suitable for a class of highly parallel general-purpose computers such as the Connection Machine. Since the processor count in such machines is of the order of tens of thousands, it is practical to assign an individual processor to each module. The connectivity information pertaining to module i is stored in the local memory of processor i, $1 \leq i \leq N$. The authors assume an SIMD control and make a minimal set of assumptions regarding the target machine.

- The control processor (CP) can broadcast an instruction or a piece of data to all the processing elements in $O(1)$ time.

```
procedure EvaluatePartialWirelength ; /* Executed by Processor i */
begin
        PWL_i ← 0; /* Reset the partial wire length */
        X ← S_i; /* Slot to which my module is assigned */
        for J := 0 to N − 1 do begin
            RECEIVE (Y) ; /* CP broadcasts the slot position for module J */
            PWL_i ← PWL_i + C(i, J) * DIST(X, Y)
        end { for }
end procedure
```

- An associative operation on N items stored in a distributed fashion among the processors is assumed to require $O(\log N)$ time.
- Near neighbor communication among processors is assumed to require $O(1)$ time.

Under these assumptions, an $O(N)$ speedup is estimated for large values of N. Table 4.3 shows the set of vector instructions used in the data parallel algorithm of [RSP88]. Unlike the previous placement accelerators, the initial placement is also selected by the parallel algorithm. A randomized algorithm is used for this purpose. The CP broadcasts a randomly selected integer r, $0 \leq r < N$ to all the processors. The initial slot for module i is then given by S_i,

$$S_i = (r + i) \mathbf{mod} N \tag{4.5}$$

Thus, $O(1)$ time is sufficient to generate a random initial placement. The cost function (total wire length) can be evaluated in $O(N)$ time as follows. Processor i executes the following loop to compute the *partial wire length* PWL_i, which is the length of wires originating at module i.

It is clear that $\frac{1}{2} \cdot \sum_i PWL_i$ is the total wire length. Since addition is an associative operation, the partial wire lengths can be added in $O(\log N)$ time. The connectivity terms $C(i, J)$ are available locally to module i. Therefore, no interprocessor communication is necessary to compute partial wire lengths. A similar

Step	Serial Complexity	Parallel Complexity
Random Place	$O(N)$	$O(1)$
Evaluate cost	$O(N^2)$	$O(N)$
Perturb	$O(\lambda)$	$O(1)$
Save Current Placement	$O(N)$	$O(1)$
Unsave Old Placement	$O(N)$	$O(1)$
Cost Difference Computation	$O(N)$	$O(\log N)$

Table 4.3: Steps in a Data-Parallel Iterative Improvement Algorithm

procedure is described in [RSP88] to compute the incremental change in wire length when the placement is subjected to perturbation. The latter requires only $O(\log N)$ time.

The perturbation scheme used in [RSP88] is known as λ-module interchange. It consists of selecting λ adjacent modules and shuffling them in a round-robin fashion. The number λ is chosen to be small, usually 2 to 5. The implementation of this scheme requires a *perturbation vector* (PV) to be broadcast to all the processors. PV is an N-bit vector whose ith bit is set to 1 or 0 depending on whether module i must participate in the perturbation or not. The control processor randomly selects λ adjacent bits in the PV and sets them to 1. Upon receiving the perturbation vector, processor i looks at the bits i, $(i+1)\mathbf{mod}N$, and $(i-1)\mathbf{mod}N$. If one or more of these bits is set to 1, the processor i participates in a data exchange operation with its neighbor(s). The details of the algorithm may be found in [RSP88]. At most two near-neighbor communication steps are involved in the parallel version of λ-module interchange.

The performance of the parallel algorithm presented in [RSP88] has been studied by means of simulation on a uniprocessor. The results of simulation verify the claim of $O(N)$ speedup. Since Simulated Annealing [KGV83] has an algorithmic structure similar to Iterative improvement, the techniques of [RSP88] can be extended to handle annealing algorithms.

4.3 Parallel Min-cut Placement

The Min-cut technique is a well known heuristic for PCB and semi-custom VLSI placement. Its chief merit is in reducing the routing-track usage, thereby improving the routability of the chip. Min-cut placement also gives good results in terms of total wire length. The essential idea in the Min-cut algorithm is to partition the given circuit into two sub-circuits such that the cross wiring between the two sub-circuits is minimal. The partitioning is then recursively applied to the two sub-circuits. The procedure stops when the sub-circuits are individual cells. The Min-cut algorithm is computationally expensive since it generates two-way circuit-partitioning as a subprocedure. Let $T(N)$ be the time taken by a partitioning algorithm to generate a two-way partition of N cells. Let the time required by the Min-cut algorithm to place N cells be denoted by $M(N)$. Then,

$$M(N) = T(N) + 2 \cdot T(N/2) + 2^2 \cdot T(N/2^2) + \cdots + 2^k \cdot T(N/2^k) \qquad (4.6)$$

where k is the depth of recursion. Since the Min-cut procedure terminates when $(N/2^k) = 1$, we have $k = \log_2 N$. The heuristic partitioning procedure proposed by Kernighan and Lin is normally used with the Min-cut algorithm. Kernighan and Lin's procedure requires $O(N^2 \log N)$ time to produce a near-optimal two-way partition of N modules. Using this result in Equation 4.6,

$$\begin{aligned} M(N) &= N^2 \log_2 N + 2 \cdot (N/2)^2 \log_2(N/2) + 2^2 \cdot (N/4)^2 \log_2(N/4) \\ &\quad + \cdots + 2^k \cdot (N/2^k)^2 \log_2(N/2^k) \end{aligned}$$

Simplifying, $M(N) = O(N^2 \log N)$. In this section, a parallel Min-cut placement procedure is described which runs on an SIMD architecture.

4.3.1 Min-cut Placement on Array Architecture

The Quadrature Min-cut placement algorithm (QPA) proposed by Breuer [Bre77] can be conveniently described by means of a *Min-cut Process Tree* as shown in Figure 4.5. Each process in this tree corresponds to a *cut-line* in QPA. The root process receives N randomly placed logic modules as input. The algorithm begins with a vertical bisection of the chip carried out by the root process (cut-line C_1). The root uses a two-way partition procedure to bisect the chip into approximately two equal halves. The modules which lie to the left (right) of the cut-line C_1 are passed on as input to the left (right) child of the root node. In general, a process at level i of tree, $0 \leq i < \log_2 N$, receives $N/2^i$ modules as input, and generates an optimal two-way partition of these modules (see Figure 4.6).

Since the QPA is a tree structured algorithm, it can be mapped onto a tree machine by assigning one processor to each process in the Min-cut Process Tree (MPT). However, since processes at only one level of MPT are active at any time, a tree architecture for this application suffers from poor processor utilization. Analytical studies reveal that the efficiency of such a scheme is $O(1/N)$ [Rav87]. We show that the orthogonal array can be used to overcome this drawback [RSP91]. The following observations lead to an efficient mapping of the Min-cut algorithm onto an orthogonal array with N processors.

- The performance of the parallel Min-cut algorithm can be enhanced if the computation of each process in MPT is parallelized. This computation consists of a two-way partition procedure. In other words, it is required to execute the partitioning procedure in parallel on the array architecture. A parallel version of the Kernighan-Lin algorithm Orthogonal Array was described in the previous chapter. The principal result of Chapter 3 is to speed up the Kernighan-Lin algorithm by a factor of $O(N/\log N)$ using an orthogonal array of size N. on the

- There are 2^i processes at level i of the MPT, each of which operates on a set of $N/2^i$ modules. Using the technique of [RSL89], $N/2^i$ processors are required for

each process at level i of MPT. Clearly, $(N/2^i) \cdot 2^i = N$ processors are necessary at each level of MPT (see Figure 4.7). Therefore, an N-processor orthogonal array can be used efficiently to implement the entire QPA in parallel, keeping all the processors busy all the time.

The parallel Min-cut algorithm is outlined in Figure 4.8. The details of the algorithm and its performance analysis are now presented. In the parallel Kernighan-Lin algorithm of the previous chapter, the array *MEMBER* contains the partition information at the end of execution. We recall that if *MEMBER* (i) contains 0 (1), it implies that module i belongs to the partition X (Y). The array *MEMBER* may be conveniently implemented as a *bit vector* of length N. Looking at the Min-cut placement algorithm, we observe that it consists of $\log N$ applications of the KL procedure. With each application of the partitioning algorithm, the position of the module becomes more and more well defined. In the parallel Min-cut placement algorithm, the slot position of a module can be traced unambiguously if a bit vector B of length $\log N$ is associated for the module. The bit vector B, which we shall denote as the *Module Position Vector*, encodes the final slot position of the module. Formally, during the k th call of the KL algorithm, $1 \leq k \leq \log N$, the k th bit of B is set to

- 1 if the corresponding module lies to the right of a vertical cutline C_k or to the bottom of a horizontal cutline C_k
- 0 if the module lies to the left of a vertical cutline C_k or to the top of a horizontal cutline C_k

In what follows, we shall explain how the module position vectors can be recovered from the available data structures. We have remarked that an N-bit vector can be used to maintain the array *MEMBER* . We shall denote this vector as a *membership vector*. Since there are $\log N$ calls to the KL procedure, we require an array of $\log N$ membership vectors to be maintained by the CP of the reduced array. The k th membership vector, $MEMBER_k$ is updated by the CP during the kth call of KL . The bit $MEMBER_k(i)$ represents the partition to which module i belongs after the

k th iteration of KL . Thus, the bit column $MEMBER_k(i)$, $1 \leq i \leq \log N$, forms the position vector for module i. Thus, data structure $MEMBER$ is sufficient for the parallel Min-cut algorithm.

4.3.1.1 Data Movement during Min-Cut Placement

At the end of each iteration of the parallel Min-cut placement algorithm, there is a need to reorganize the connectivity matrix among the memory banks of the Orthogonal Array. It has been shown in the previous chapter (Section 3.3.3) that such reorganization can be done in two well defined steps, namely, *row exchange* and *column exchange*. Row and column exchanges both require constant time, neither of the steps involves any memory access conflicts. After the first iteration of the Min-cut procedure, up to $\frac{N}{2}2$ exchange operations can become necessary in the worst case. Similarly, at the end of the ith iteration, the amount of time spent in redistribution of data does not exceed $\frac{N}{2^i}$. This observation leads us to the following lemma.

Lemma 8 *The overhead of data reorganization in the Min-cut placement procedure is $O(N)$.*

The lemma is easily proved by summing the reorganization costs i.e $\sum_{i=0}^{\log_2 N} \frac{N}{2^i} = O(N)$.

We use the next lemma to compare the computation costs and the communication overheads of our scheme.

Lemma 9 *The total computation time of the parallel Min-cut procedure on the Orthogonal Array is $O(N \log N)$.*

The proof of the lemma is based on the fact that the execution of Kernighan-Lin at ith iteration of the Min-cut placement procedure requires $\frac{N}{2^i} \log(\frac{N}{2^i})$ time. Thus, the total computational needs evaluate to $\sum_{i=0}^{\log_2 N} \frac{N}{2^i} \log(\frac{N}{2^i})$, which is $O(N \log N)$.

4.3.1.2 Speedup Analysis of Parallel Min-cut Placement

In this section, we show that the speedup obtained by using an Orthogonal Array for Min-cut placement is $O(N/\log N)$. With $O(N)$ processors in the Orthogonal Array this speedup is the best obtainable.

Lemma 10 *When compared to single processor execution of Min-cut placement, the speedup obtained by the algorithm PARALLEL-MIN-CUT is $O(N/\log N)$.*

Proof : The running time for sequential Min-cut placement is found by computing the time required by each level of processes in MPT and then taking the sum. The time T_i consumed by a process at level i to sequentially execute the Kernighan-Lin procedure is given by $T_i = (\frac{N}{2^i})^2$. The time taken by a uniprocessor to execute the QPA, denoted T_{sp}, is given by

$$T_{sp} = \sum_{i=0}^{\alpha}(T_i.2^i) = 2.N^2 - N = O(N^2). \tag{4.7}$$

The time for parallel Min-cut placement on the Orthogonal Array is denoted T_{pp}, and, by lemma 9, is $O(N \log N)$. Combining this with equation 4.7, the speedup S obtained by employing the Orthogonal Array for Min-cut placement is

$$S = (\frac{T_{sp}}{T_{pp}}) = O(N/\log N)\blacksquare. \tag{4.8}$$

4.4 A Divide-and-Conquer approach

A placement algorithm based on the *Divide-and-Conquer* paradigm has been proposed in [RS88]. This algorithm divides the given set of logic modules into small *clusters* and generates an optimal placement for each cluster. Finally, in a pasting step, the algorithm combines the optimal solutions for the smaller problems into a near-optimal solution for the original placement problem. Details of the procedure are given in Section 4.4.1. A merit of such a Divide-and-Conquer procedure is that

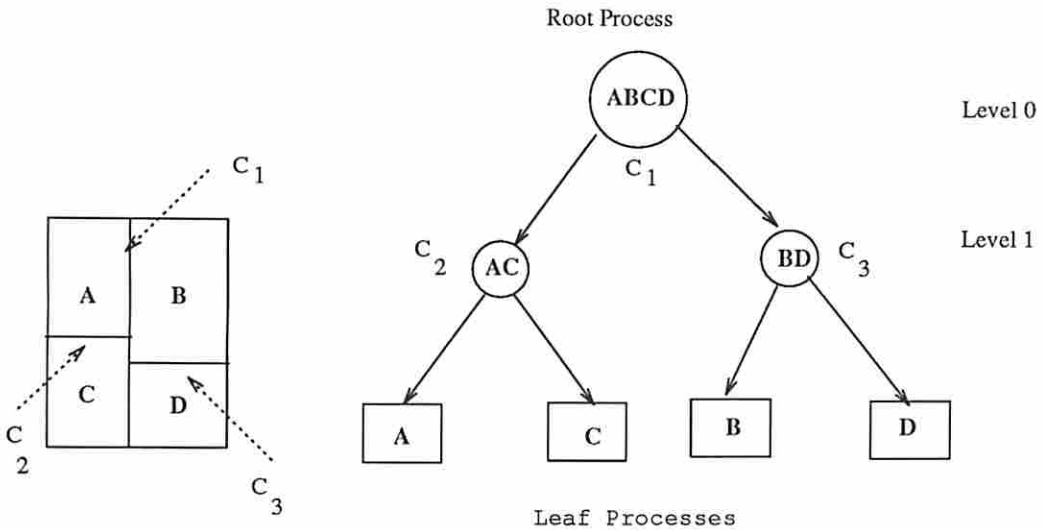

Figure 4.5: Min-cut Process Tree

```
procedure ParallelMinCut;
comment :  process code for each node in Min-cut Process Tree
begin
        if ( not the root node ) then
        begin
                Receive the set of modules from parent node ;
                Bisect your block by a cut-line c;
                Use Kernighan-Lin procedure to arrive at an optimal partition;
                Send the modules that lie to the left of c to the left child;
                Send the modules that lie to the right of c to the right child;
        end ;
end procedure
```

Figure 4.6: Process Code for a node in the Min-cut Process Tree

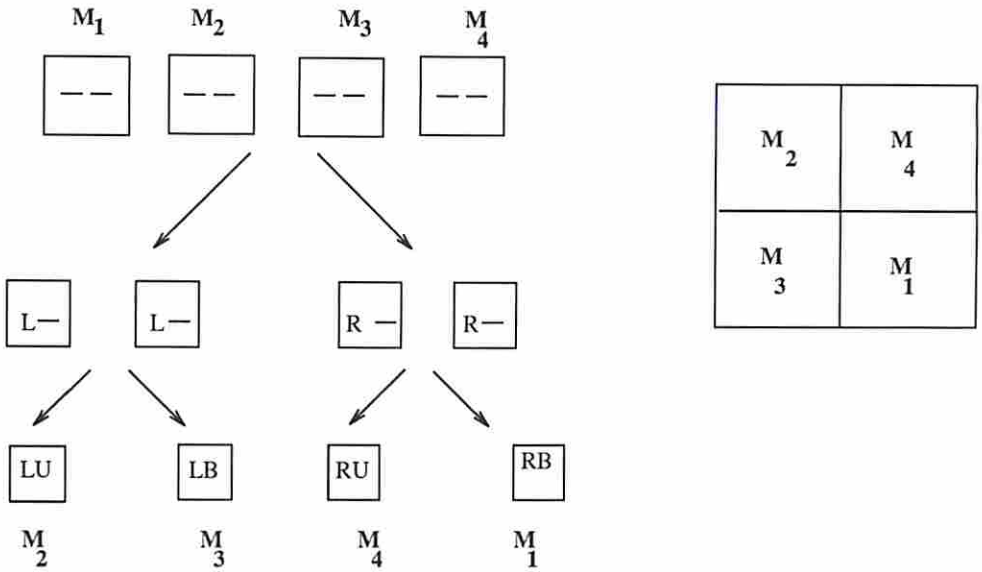

Figure 4.7: Parallelizing Min-cut Placement

```
procedure Parallel-Min-cut;
begin
      for i := 0 to log2 N do begin
          if (i = log2 N) then
             { each processor has a single module
              Each processor executes the following step :
              Assign your module to the slot k, where k is your processor id}
          else
                {A Cut Step : }
                logically partition the Orthogonal Array into 2^i subarrays;
                assign N/2^i modules to each subarray;
                { each subarray executes the parallel Kernighan-Lin procedure
                 on its input }
      end {for}
end procedure
```

Figure 4.8: Min-cut Algorithm on Orthogonal Array

it lends itself naturally to a parallel realization. Two parallel implementations are known for the Divide-and-Conquer placement procedure. The first is an attempt to use a special-purpose array architecture. This is discussed in Section 4.4.2. The second approach is to map the algorithm onto a general-purpose hypercube multiprocessor, as discussed in Section 4.4.3.

4.4.1 The DAC Placement Procedure

The Divide-and-Conquer (DAC) placement procedure proceeds by breaking up the connectivity matrix C into smaller pieces. Each piece corresponds to a *cluster* of modules. A cluster represents a placement problem of size small enough to be placed *optimally* using purely enumerative techniques. Semi-enumerative techniques such as Branch and Bound, which are known to produce near-optimal solutions, may also be used to place the modules which belong to a cluster. Hereafter, the term *Local Placement* is used to refer to the placement of modules within a cluster. Following the local placement phase, each cluster is *shrunk* into a hypothetical module. The hypothetical modules form a placement problem whose size is the same as the number of clusters, denoted by ℓ. If ℓ is small enough, the hypothetical modules can be placed optimally using brute force enumeration. Each hypothetical module i may then be replaced by the placement of modules that belong to the cluster i. When ℓ is large, the divide-and-conquer procedure can be recursively applied to the problem of size ℓ. Figure 4.9 illustrates the procedure DAC for $N = 16$ and $m = 4$.

There are $Q(N, m)$ ways of dividing a set of N modules into clusters of size m:

$$Q(N,m) = \frac{N!}{(m!)^{\lceil \frac{N}{m} \rceil} \times (\lceil \frac{N}{m} \rceil)!} \tag{4.9}$$

When m is a fraction of N, say $m = N/a$, $Q(N, m)$ can be exponentially large for $1 < a < N$. Hence it is impractical to try all possible partitions; instead, [RS88] resorts to a local search mechanism. In this method, only a fraction of $Q(N, m)$ partitions are investigated before arriving at a suboptimal solution. Refer to the description of the DAC procedure in Figure 4.10. The constant ξ used in

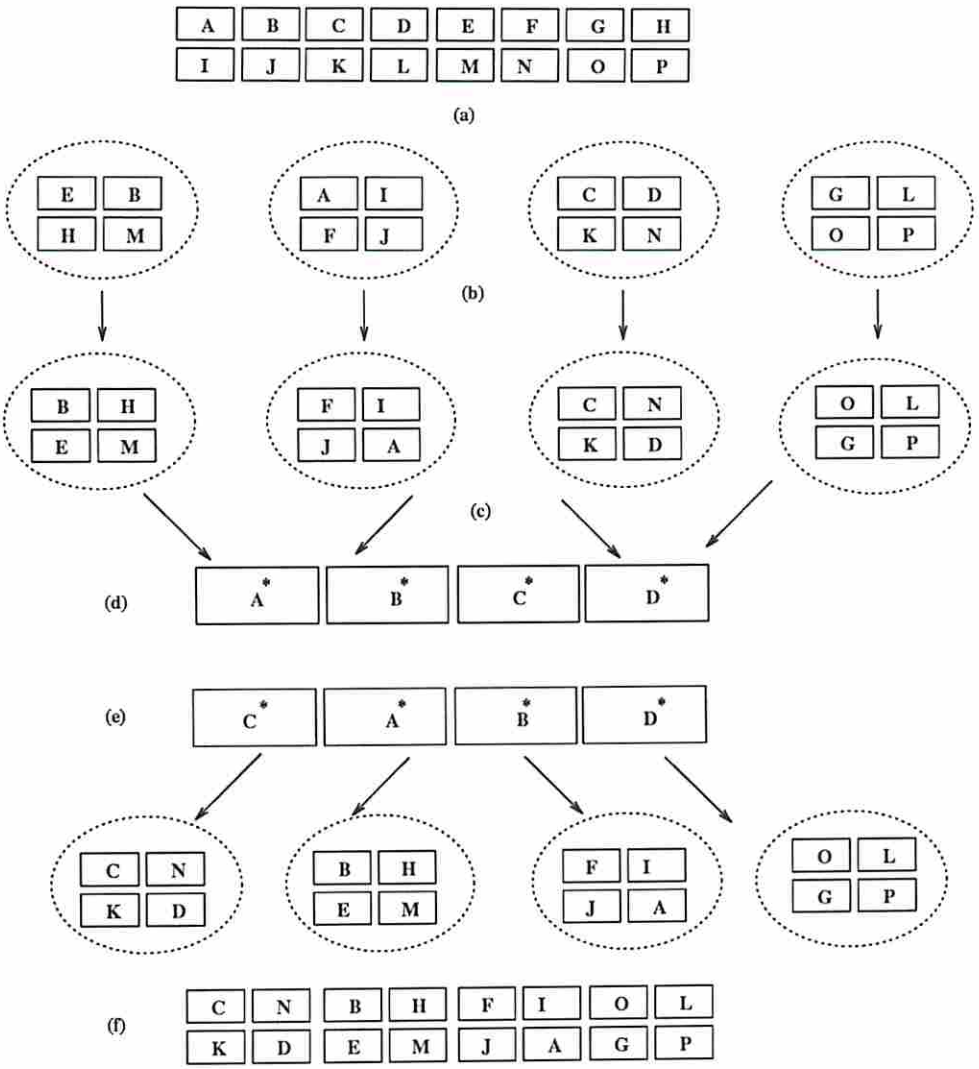

Figure 4.9: Basic Idea in Divide-and-Conquer Placement : $N = 16, m = 4$

```
procedure DAC(N, m);
const ξ;
begin
        Randomly Partition N logic modules into
        ⌈N/m⌉ clusters of size m;
        count = 0;
        repeat
                for every cluster i ∈ {1 .. ⌈N/m⌉} do
                    LocalPlace (m, i);
                PlaceClusters (⌈N/m⌉);
                Compute the cost of new placement;
                if cost(new placement) < previous cost then
                begin
                  accept new placement;
                  previous cost = cost (new placement);
                end
                Generate a new partition;
                count = count+1 ;
        until (count = ξ);
end procedure
```

Figure 4.10: Divide-and-Conquer Placement

the procedure is chosen to be a small percentage of $Q(N, m)$. The ξ parameter dictates the fraction of the search space considered by the placement algorithm before arriving at the suboptimal solution. The procedures *LocalPlace* and *PlaceClusters* use enumerative techniques to generate optimal placements. Heuristics such as local search can also be used in these steps.

4.4.2 DAC algorithm on Orthogonal Array

The orthogonal array discussed in Section 4.3 has been used in [RS88] to execute the DAC algorithm. The array has $\lceil \frac{N}{m} \rceil$ processors which concurrently execute all the invocations of *LocalPlace*. There are $\lceil \frac{N}{m} \rceil^2$ blocks of memory elements, each block containing m^2 memory cells.

Step 1 The control processor (CP) generates an initial partition of the modules. The set of modules received by processor PE_i is denoted by M_i. The connectivity matrix is stored in the memory of the orthogonal array in the following manner. If $M_i = \{q_1, q_2, \ldots, q_m\}$ the m^2 elements of the connectivity matrix C which are of the form $C_{q_j q_k}$ are stored in the memory module (i, i). This enables each PE_i to use the elements in module (i, i) to perform a local placement *without causing memory conflicts.*

Step 2 The local placement phase is then initiated by the control processor, during which each processor PE_i executes an enumerative algorithm to place the set of m modules in M_i. The cost of the local placement generated by PE_i is denoted by LC_i.

Step 3 The *collapsed connectivity matrix* C' is computed using the elements of the original connectivity matrix C. The entry C'_{ij} is given by

$$C'_{ij} = \sum_{p \in M_i} \sum_{q \in M_j} C_{pq} \tag{4.10}$$

The collapsed connectivity matrix has $\lceil \frac{N}{m} \rceil^2$ elements. By assigning PE_i to compute an entire column of the matrix C', a speedup of $P = \lceil \frac{N}{m} \rceil$ is achieved.

Step 4 The matrix C' represents a placement problem of size $\lceil \frac{N}{m} \rceil$. The modules in this problem are entire clusters. The CP places the hypothetical modules, also using an enumerative technique. The cost of this placement is known as the Interconnection Cost (IC).

Step 5 The total cost of the placement is now evaluated as

$$GC = \sum_{i=1}^{P} (LC_i) + IC \tag{4.11}$$

Step 6 The CP decides to accept or reject the current placement by comparing the cost of the current placement with that of the *previous* placement. For the first iteration, the cost of the "previous configuration" is taken as $+\infty$.

Step 7 (Generate a new partition)

λ modules are randomly selected from λ different clusters, $2 \leq \lambda \leq P$. These λ modules are permuted among the clusters in a round robin fashion. The details of this *shuffle step* are provided later.

Step 8 (Repeat steps 2 through 7)

The CP decides to reiterate steps 2 through 7 depending on some predefined termination criterion e.g., the maximum time allowed to be spent on the placement.

4.4.2.1 Shuffle Operation

The contents of an N-bit register R are broadcast by the control processor to all the PEs of the array. The bit R_i is set to 1 if and only if the module i is displaced from its partition. Let T_k indicate the cluster to which module k belongs. The shuffle operation moves module j from T_j to T_i if **(1)** bits R_i and R_j are both set to 1, **(2)** $j > i$, and **(3)** $R_k = 0, \forall\ i < k < j$.

A λ-shuffle of a partition can be realized by using at most $(\lambda - 1)$ pairwise-shuffles. The data movement associated with a pairwise shuffle is easy to implement on the orthogonal array. For example, consider the operation of shuffling the modules q_1 and q_2. The effect of this operation on the storage of the connectivity matrix consists of

1. renumbering all entries of the form (q_1, x) to (q_2, x)
2. renumbering all entries of the form (q_2, y) to (q_1, y)
3. renumbering all entries of the form (x, q_1) to (x, q_2)
4. renumbering all entries of the form (y, q_2) to (y, q_1)

In the language of hardware, this means *exchange the contents of rows and columns corresponding to* q_1 *and* q_2. Since the orthogonal array allows simultaneous access to different rows and columns of the memory blocks, the above mentioned data transfer can be made concurrently.

4.4.2.2 Performance

Since all the local placements are executed concurrently, a speedup of P is achieved in Step 2. The computation of the collapsed connectivity matrix (Step 3) also offers a speedup factor of P as explained earlier. The placement of clusters (Step 4) is carried out sequentially by the control processor. Step 5 achieves a speedup of $P/\log_2 P$, since the summation is an associative and can be carried out using a tree computation. Step 7 carries an overhead of data reorganization.

The other factors which contribute to the performance of the parallel algorithm are the algorithm used for local placement, the value selected for λ, the cluster size m, and the iteration count ξ. In a simulation study, the selection of cluster size m was observed crucial in deciding the quality of final placement [RS88]. Choosing $m = N^{2/3}$ yielded best results. A performance metric γ known as the *rate of convergence* is defined as the number of *improvement cycles* γ required by a placement algorithm to arrive at a placement whose cost is a certain fraction of the final solution. The simulation study in [RS88] indicates that in a given amount of time τ, the parallel *DAC* algorithm yields far better quality solutions than a sequential iterative improvement.

4.4.3 DAC on Hypercube Architecture

Of late, the *Hypercube* interconnection has gained great popularity in constructing both SIMD and MIMD computers. Its fault tolerance characteristics, routing efficiency, and expandability are the chief advantages of the hypercube interconnection network. As an added attraction, it is possible to embed into the hypercube, other architectures such as the tree, ring, rectangular mesh and linear array. A hypercube of d dimensions has 2^d nodes. The nodes are numbered $0, 1, \cdots, 2^d - 1$ using their binary representation. A node i is connected to a node $j \neq i$ iff the binary representations of i and j differ in exactly one bit position. Below, a hypercubic computer with a *distributed memory* architecture is assumed, where each processor has private access to its local memory. Processors exchange messages for synchronization and

mutual cooperation. At most d communication steps are necessary to route data from one processor to another.

In [RS89b], an MIMD hypercube computer with P nodes is used as the target architecture. The node whose address is 00...0 is used as the "master" node. The master initializes the computations to run as tasks on the different nodes of the hypercube (including the master itself). After the completion of a task, each node sends a synchronization signal to the master. The master can then initiate the next computation.

As with the orthogonal array, the basic idea in parallelizing the *DAC* placement procedure is to carry out all invocations of the *LocalPlace* procedure concurrently on the nodes of the hypercube architecture. However, as an improvement, the procedure *PlaceClusters* is also executed in a distributed fashion. This is achieved by means of an efficient data allocation scheme and an efficient task partition.

With each module i, a list known as the *connectivity list* is associated, which consists of tuples of the form (j, C_{ij}), $C_{ij} > 0$. Since modules have a maximum number of pins, say p, there are at most p elements in the connectivity list. Therefore an array of p tuples can be used to represent the connectivity list for a module. The *number* of elements in the connectivity list of module i, $1 \leq i \leq N$, is stored in an array $W[1..N]$. Let M_i denote the set of modules assigned to a node i. The connectivity list for each module $j \in M_i$ is stored in the local memory of node i. Assuming $\lceil \frac{N}{m} \rceil = 2^d$, where d is the dimension of the Hypercube, it is possible to assign one individual node to carry out one local placement. After the completion of local placement, processor i computes the elements C'_{ik} of the collapsed connectivity matrix $1 \leq i, k \leq \lceil \frac{N}{m} \rceil$. The data allocation scheme described above allows P local placements to be performed concurrently on individual processors without communication overheads. Again, no inter-processor communication is necessary in computing the collapsed connectivity matrix.

A parallel version of the procedure *PlaceClusters* is the following *distributed placement* problem : *Place* $\lceil \frac{N}{m} \rceil$ *clusters (hypothetical modules) on* $\lceil \frac{N}{m} \rceil$ *slots, given* P

processors. The connectivity information among the clusters is stored as the collapsed connectivity matrix of dimension $\lceil \frac{N}{m} \rceil \times \lceil \frac{N}{m} \rceil$. The processor i, $1 \leq i \leq \lceil \frac{N}{m} \rceil$ has local access to the ith row of the collapsed connectivity matrix C'. The *data parallel* local search algorithm (Section 4.3) may be conveniently used to solve the distributed placement problem.

The structure of *HyperPlace*, the Hypercube placement algorithm, is similar to the parallel DAC algorithm of Section 4.4.2, except for some improvements. For example, the implementation of the *shuffle* step is interesting. Two types of perturbation schemes are identified in [RS89b].

- *intra-cluster perturbation* , where the modules within a cluster are subjected to a random perturbation. The λ-interchange may be conveniently used. Since a processor node has local access to $\lceil \frac{N}{mP} \rceil$ clusters, λ-interchange may be applied to each of these clusters.

- *inter-cluster perturbation* , where modules belonging to different clusters participate in a pairwise-interchange. An inter-cluster perturbation may or may not involve communication overhead. If the source and destination clusters have been assigned to the two different processor nodes, inter-processor communication is necessary. This situation is dealt in the following way. The processors in the Hypercube are divided into two half-cubes of $P/2$ processors each. The identifiers of two processors i and j which belong to different half-cubes differ by exactly one bit. By the property of the Hypercube interconnection network, each processor in a given half-cube has a direct communication link to exactly one processor in the other half-cube. Inter-cluster perturbation must be performed across two nodes which belong to two different half-cubes and which are connected by a direct link. As an example, consider the case when $d = 3$. A possible partition into two half-cubes is {000, 001, 100, 101}, {010, 011, 110, 111}. Restrict that clusters assigned to processors x0y and x1y must participate in an inter-cluster perturbation (xy = 00,01,10,11). There are d ways of partitioning a d-dimensional Hypercube into two half-cubes. The

master node randomly selects one particular partition and broadcasts this selection information to all the processor nodes. Each processor can then identify itself as a source node or a destination node depending on whether its selection bit is 0 or 1. In the above example, the selection bit is at the second position, {000, 001, 100, 101} are the source nodes, and {010, 011, 110, 111} are the destination nodes. The source and destination nodes then randomly select two modules assigned to them and interchange the modules.

Table 4.4 summarizes the timing information on the important steps of the *HyperPlace* algorithm. From this table, it may be verified that the speedup ratio is the optimal value of P for most compute-intensive steps. There is a communication overhead proportional to $\log_2 P$ for certain steps, affecting the performance of the algorithm. An approximate expression for the speedup ratio is

$$S = \frac{1}{1/P + (\log_2 P)/N} \tag{4.12}$$

which converges to P for large values of N.

4.5 Hall's r-dimensional Placement Algorithm

Consider the linear placement problem, where a set of modules must be placed in a single row such that the amount of wiring is minimized. Let $x_1, x_2, \cdots, x_n$ be the X-coordinates of the n modules after placement. Let C_{ij} represent the connectivity between modules i and j. A possible cost function which represents the total amount of wiring is

$$z = \frac{1}{2} \sum_i \sum_j (x_i - x_j)^2 C_{ij} \tag{4.13}$$

Hall [Hal70] pointed out that z is a *quadratic form* and could be expressed as

$$z = X'BX \tag{4.14}$$

Step	Sequential Time	Parallel Time	Communication Overhead?
Local Placement	$\lceil \frac{N}{m} \rceil \cdot T(m)$	$\lceil \frac{N}{mP} \rceil \cdot T(m)$	No
Collapsing the Connectivity multilist	$O(N)$	$O(\frac{N}{P})$	No
Cluster Placement	$\mu\ O(N)$	$\mu\ O(\frac{N \log P}{P})$	Yes
Global Cost	$\lceil \frac{N}{m} \rceil \cdot O(m^2)$	$O(\lceil \frac{N}{mP} \rceil) + O(\log P)$	Yes
Intra-cluster Placement Perturbation	$O(\lceil \frac{N}{m} \rceil)$	$O(\lceil \frac{N}{mP} \rceil)$	No
Inter-cluster Placement Perturbation	$O(\lceil \frac{N}{m} \rceil)$	$O(\lceil \frac{N}{mP} \rceil) + O(\log P)$	Yes

Table 4.4: Performance Evaluation of HyperPlace Algorithm

where B is a positive semi-definite $n \times n$ matrix and X is the row vector of the coordinates x_i. [1] Hall also showed that B could be computed using the connectivity matrix C as follows.

$$B = D - C \tag{4.15}$$

Here $D = [d_{ij}]$ is a 'diagonal matrix' i.e. all the non-diagonal elements of D are zero. Furthermore, d_{ii} is given by

$$d_{ii} = \sum_{j=1}^{n} C_{ij} \tag{4.16}$$

The linear placement problem can be expressed as the optimization of $z = X'BX$, subject to the constraints that the determinant $|B| \geq 0$, and $X'X = 1$. The constraint $X'X = 1$ avoids the trivial solution $x_i = 0$. Using the method of Lagrangean multipliers, Hall obtained

$$(B - \lambda I)X = 0 \tag{4.17}$$

[1] A matrix A is said to be positive semi-definite if, for every vector Y, the value $Y'AY \geq 0$.

where λ is a Lagrange multiplier and I is the identity matrix. Equation 4.17 yields a non-trivial solution to X if and only if λ *is an eigenvalue of* B *and* X *is the corresponding eigenvector.* Since $X'X = 1$, equation 4.17 can be further written as

$$\lambda = X'BX. \tag{4.18}$$

Therefore, the minimization of cost z boils down to finding the eigenvector X of matrix B which minimizes z. The minimum eigenvalue represents the minimum cost function. To be exact, the first minimum eigenvalue does not give the minimum cost function, since it corresponds to the trivial solution $x_1 = x_2 = \cdots = x_n = \frac{1}{\sqrt{2}}$. The second minimum eigenvalue corresponds to the optimum solution. Hall also gave a generalization of the above method to r-dimensions, $r = 1, 2, \cdots, n-1$. The linear placement problem corresponds to the case $r = 1$. Hall's method has been only of theoretical interest, since it is a computationally expensive task to find the eigenvalues and eigenvectors of a large matrix. With the advent of vector supercomputers, the above formulation of the placement problem is a practical approach. In this section, we describe an Alliant FX/80 implementation of a placement algorithm based on Hall's technique.

The first step in Hall's r-dimensional placement procedure is to use the connectivity matrix C and compute the B matrix (see Equation 4.15). B is a positive semi-definite, symmetric matrix. The second step is to compute the eigenvalues and eigenvectors of B. Sort the eigenvalues in the ascending order of magnitude. Let $e(j)$ denote the jth smallest eigenvector. Further, let $e(j,k)$ denote the kth element of $e(j)$. In an r-dimensional plane, the coordinates of module k are given by $\{e(2,k), e(3,k), \cdots, e(r+1,k)\}$.

Example 4.2 :

Consider 7 circuit modules which are connected to form a binary tree with 4 leaves (Figure 4.11(a)). The connectivity matrix for the binary tree is shown in Figure 4.11(b). The B matrix is shown in Figure 4.11(c). The eigenvalues for this example are, in the ascending order of magnitude, {0, .268, 1, 1, 1.586,

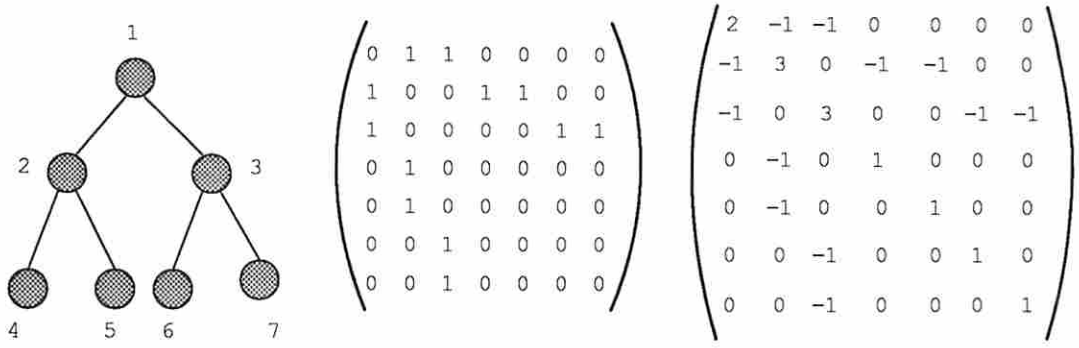

Figure 4.11: Hall's placement algorithm for a tree-structured circuit.

3.732, 4.414 }. Suppose that we are interested in a two-dimensional placement of the 7 modules. Two appropriate eigenvectors must be selected to define the coordinate system for the modules. The minimum eigenvalue, namely 0, is ignored. The next two eigenvalues are 0.268 and 1; the corresponding eigenvectors are

$$e(2) = (0.0000, 0.3251, -0.3251, 0.4440, 0.4440, -0.4440, -0.4440) \quad (4.19)$$

$$e(3) = (0.0000, 0.0000, 0.0000, 0.7011, -0.7011, 0.0918, -0.0918) \quad (4.20)$$

From this, it is evident that module 1 must be placed at (0.0,0.0) in the two-dimensional plane. Similarly, modules 2 and 3 are placed at (0.3251,0) and (-.3251,0) respectively. The complete placement of the cell is shown in Figure 4.12.

4.5.1 Implementation on Alliant

We have implemented Hall's algorithm on an Alliant FX/80 in FX/Fortran. The program was optimized for vector concurrent mode of execution. Details of Alliant FX/80 architecture and FX/Fortran can be found in Appendix A. The procedure *rsm* was used to compute the eigenvectors of the B matrix. This Fortran subroutine was obtained from the eigensystem subroutine package 'eispack' [S+76] and then

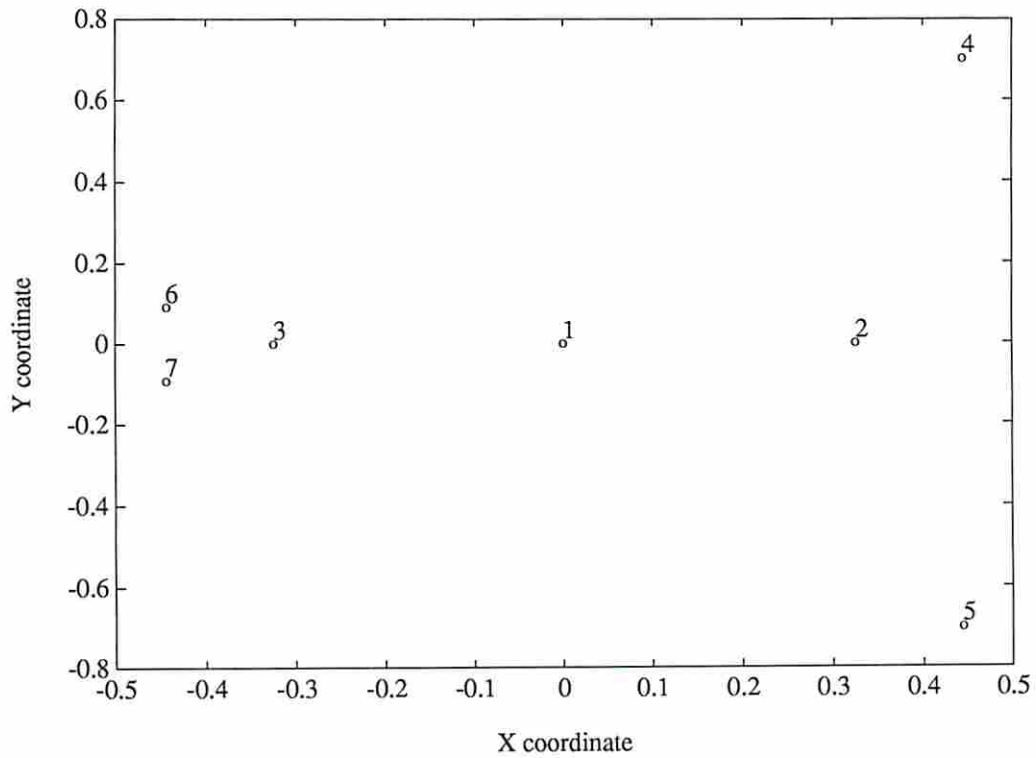

Figure 4.12: Placement of a tree-structured circuit with 7 modules.

Circuit Name	n	Sequential Time (sec)	Time on 1 CE (sec)	Time on 2 CEs (sec)	Time on 3 CEs (sec)	Time on 4 CEs (sec)
inp20.1.1	20	0.1555	0.0833	0.0806	0.0789	0.0779
inp50.1.1	50	1.1689	0.4554	0.4202	0.4062	0.4034
inp100.1.1	100	6.4364	1.9093	1.6783	1.5991	1.5695
inp150.1.1	150	18.652	4.6687	3.9408	3.7026	3.6075
inp200.1.1	200	41.017	9.0886	7.4171	6.8583	6.5906
inp300.1.1	300	125.58	24.504	18.909	17.074	16.218
inp350.1.1	350	193.29	28.218	27.583	24.517	22.912

Table 4.5: Results of Hall's Placement Algorithm on Alliant FX/80

vectorized. *rsm* is customized to operate on real symmetric matrices. The main inputs to this subroutine are the matrix B, its dimensionality n and the number m of eigenvectors desired. As output, the procedure generates m eigenvectors that correspond to the m smallest eigenvalues. $m = 3$ was used and the eigenvector $e(1)$ was ignored. The program was tested against several benchmark circuits. These circuits were used by Saab and Rao to test their circuit partition algorithm [SR90]. The results of our experiments are shown in Table 4.5. It can be observed that vectorization is more efficient than concurrent processing in the following sense. Increasing the number of computing elements (CEs) from one to two or more does not bring a proportional improvement in the performance of the algorithm.

4.6 Placement by Simulated Annealing

In Section 4.2, it was pointed out that an Iterative Improvement algorithm can provide a local optimal solution to a combinatorial optimization problem. Scientists have drawn an analogy between the optimization process and the process of cooling molten metal. If the optimization problem involves minimizing a function of N variables $x_1, x_2, \cdots, x_N$, an assignment of feasible values of these variables constitutes a *configuration*. The iterative improvement technique generates a configuration r_j by making small changes to the existing configuration r_i. The new configuration r_j is accepted if

$$\Delta_{ij} = cost(r_j) - cost(r_i) < 0 \tag{4.21}$$

Such an optimization technique has been compared to rapid quenching of molten metal. Just like rapid cooling results in meta-stable states, iterative improvement gets stuck in local optimal solutions. To avoid meta-stable states, annealing (or slow cooling) is necessary. This is the guiding principle behind a simulated annealing optimization algorithm. An annealing algorithm attempts to escape from local optima by *probabilistically* accepting bad moves. The acceptance criterion adopted by an annealing algorithm is known as *Metropolis Criterion* [KGV83]. Using the terminology of Equation 4.21, the configuration r_j is acceptable if one of the following conditions is satisfied.

1. $\Delta_{ij} < 0$
2. $e^{-\Delta_{ij}/T} > rand(0,1)$

In the Metropolis Criterion, the variable T is known as the temperature. The temperature is gradually lowered during the course of the algorithm. In the high temperature zone, the second condition is almost always satisfied, since $e^{-\Delta_{ij}/T} \approx 1$ for large T. Therefore, most moves are accepted at high temperatures. This is analogous to the fact that molecules of the molten substance possess greater energy at high temperatures and can move about freely. As the temperature is reduced, the molecular movement is constrained; accordingly, the algorithm tends more to reject than accept bad moves. At $T = 0$, the algorithm behaves exactly like a greedy algorithm.

Figure 4.13 shows the complete simulated annealing algorithm, as applied to a minimization problem. The algorithm begins with an initial temperature T_i and simulates the process of cooling by reducing the temperature by a factor of $\alpha < 1$. The constant α is chosen to be close to unity, say 0.95, so that "slow cooling" is effectively simulated. The code within the **while** loop is nothing but a local search with one important difference. The acceptance criterion is replaced by the Metropolis criterion. The number of moves accepted at temperature T is denoted by $\theta(T)$; this is also known as the annealing schedule.

```
procedure SA;
begin
        T := T_i            /* T_i is the initial temperature */
        S := S_0            /* S_0 is the initial system state */
        while (T > T_f) do begin          /* Do until final temperature*/
                accept := 0;
                stop := false ;
                repeat              /* Metropolis Procedure */
                R_1:    S' := perturb(S);
                R_2:    Δ_C := C(S') − C(S);
                R_3:    if (Δ_C < 0) or (random < e^{−Δ_C/T}) then begin
                          S := S' ;          /* accept the new configuration */
                          accept := accept + 1;
                          if (accept = θ(T) ) stop := true ; /* stop the repeat loop */
                        end
                until (stop = true ) ;
                T := T · α ;              /* cool the system */
        end /* while */
end procedure
```

Figure 4.13: Simulated Annealing Procedure

Simulated Annealing (SA) has generated significant interest as a general-purpose heuristic to solve optimization problems. Annealing algorithms are easy to implement, and have been used in a number of applications such as logic partitioning, module placement, and global routing [RP90].

4.6.1 Parallelism in Simulated Annealing

The following are the sources of parallelism in a Simulated Annealing algorithm.

- *System Perturbation :* An individual processor can be assigned to a subset of system variables x_i. Each processor makes changes to its private set of variables. This is also known as *Parallel Moves* in the literature. Alternately, a set of processors can share the work involved in making a single move. This is called *move decomposition.*

- *Cost Evaluation :* The change in system cost is generally expensive to compute, and may be evaluated in parallel.

- *Accept/Reject Decision :* If multiple processors are used to carry out multiple perturbations, each processor must decide to accept or reject its contribution to the change in system state. These decisions can be concurrently made by the processors independently of one another.

- *Pipelining :* The temporal parallelism in the execution of perturb-evaluate-decide cycle can be exploited to design a pipelined version of the Simulated Annealing algorithm.

4.6.1.1 Parallel Moves

Kravitz and Rutenbar [Kra87] report a standard cell placement algorithm based on Simulated Annealing, implemented on a VAX-11/784. Casotto et al. have implemented a macro-cell placement algorithm based on Simulated Annealing on a Sequent Balance 8000. Both the above machines are shared memory multiprocessors. The authors of [Kra87] have proposed a *Parallel Moves* strategy, which involves simultaneous evaluation of a number of moves, and a *move decomposition* strategy, where a single move is subdivided into several tasks for concurrent execution. When moves are made in parallel, there is a possibility of erroneous decisions , as discussed in Section 4.2. Erroneous decisions pose a serious threat to the convergence of the annealing algorithm. There are two solutions to this problem.

1. Allow interacting parallel moves, but minimize the uncertainty of convergence.

2. Guarantee convergence by prohibiting interacting parallel moves.

The latter solution is adopted in [Kra87]. The authors define a *serializable subset of moves* S^{ni}, as an ordered subset of moves, which, when evaluated in parallel, would produce the same accept/reject decisions as their serial evaluation. Convergence is not affected by parallel execution of moves that belong to S^{ni}. The simplest serializable subset S^{ss} consists of a sequence of rejected moves and a single accepted

move. Thus, to identify the set S^{ss} at each time step, the authors make several parallel moves and retain all the rejected moves as well as a single accepted move. If the size of the set S^{ss} is treated as a random variable, then the expected value of this variable is a measure of the resulting speedup. It has been shown that

$$E[\mid S^{ss} \mid] = P(1 - a(T)) + 1 - (1 - a(T))^P \tag{4.22}$$

where P is the number of processors, and $a(T)$ is the acceptance probability at temperature T. Since $a(T)$ is nearly unity at high temperatures, the speedup performance of *Parallel moves* strategy is poor. The functional decomposition of a move into multiple tasks is also not an attractive scheme, since it involves a considerable synchronization overhead.

The algorithm due to Casotto et al. also uses the *Parallel moves* strategy, but it differs from [Kra87] by allowing parallel interacting moves. There is thus an uncertainty in the value of the objective function. Empirical evidence is used to show that this uncertainty does not affect the convergence of Simulated Annealing. An attempt is made to reduce the uncertainty of the objective function by means of *clustering*. Modules which are likely to interact strongly are assigned to the same processor.

Jayaraman [Jay87] has used a Hypercube multiprocessor to execute an annealing algorithm for floorplanning. This scheme also uses *Parallel moves*, but attempts to solve the problem of erroneous decisions in a slightly different manner. Periodically, a *global update* is performed to correct the state information of all the processors. Each processor sends a copy of its state to a synchronizing processor. The synchronizing processor computes the true global state and broadcasts this information to all the processors. Since inter-processor communication is expensive in multiprocessors, a global update involves a considerable overhead.

4.6.2 Pipelined Execution of Simulated Annealing

We have used the MPSD approach to design a parallel placement algorithm based on Simulated Annealing [RP87]. The steps R_1, R_2, and R_3 of the procedure *SA* exhibit temporal parallelism, and may therefore be executed on different stages of a pipeline as explained below. When a given system configuration P_i is being perturbed, the cost of P_{i-1} can be simultaneously evaluated. Concurrently, the Metropolis criterion can be applied to P_{i-2} and P_{i-3}. The pipeline may be thus constructed using three stages, *Pert, Eval*, and *Decide* (see Figure 4.14). The functions of the three stages are, respectively, system perturbation, cost evaluation, and acceptance decision. Since the time requirements of the three stages to complete one operation are different, buffering is necessary between the stages. A circular queue, known as *Configuration Queue*, is maintained between the stages *Pert* and *Eval*. A second queue, called *Cost Queue* is maintained between *Eval* and *Decide*. The configuration queue is filled by *Pert* and emptied by the *Eval* stage. Whenever the *Pert* stage performs a move and changes the state P_i to P_{i+1}, it encodes the move into an entry on the configuration queue. If the perturbation technique is *Pairwise Interchange*, a simple way to encode a move is by means of a bit pattern B of length N. The bit vector is set up such that

$$B_i = \begin{cases} 1 & \text{if module } i \text{ must be displaced} \\ 0 & \text{if the module } i \text{ must be stationary} \end{cases}$$

An item in the cost queue is a binary encoded form of the *incremental cost* associated with a move. The *Eval* stage reads off move information from the configuration queue, computes the change in cost function resulting from the move, and stores the incremental cost on the cost queue. The *Decide* stage deletes an item from the cost queue and applies the Metropolis criterion. Acceptable moves are applied to the system state.

A timing diagram of the pipeline appears in Figure 4.15, where stage *Pert* is shown to generate a configuration P_j from P_i by the symbolic notation "$P_i \rightarrow P_j$." C_i, C_j are the costs of P_i, P_j. The notation $C_i \propto C_j$ is used to mean that C_i *is*

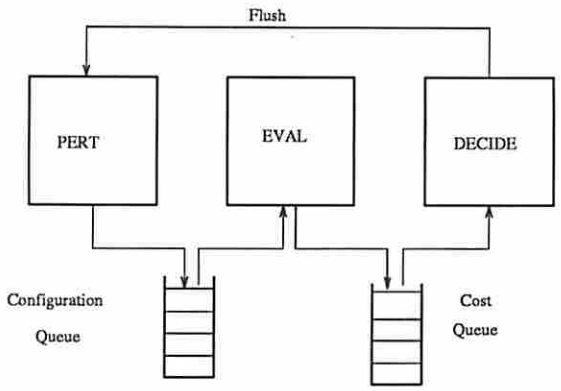

Figure 4.14: Organization of a Pipeline for Simulated Annealing

DECIDE			$c_1 < \infty$	$c_2 < c_1$	$c_3 < c_2$	$c_4 < c_3$	$\cdots$
EVAL		Δc_{01}	Δc_{12}	Δc_{23}	Δc_{34}	Δc_{45}	$\cdots$
PERT	$P_1 \rightarrow P_2$	$P_2 \rightarrow P_3$	$P_3 \rightarrow P_4$	$P_4 \rightarrow P_5$	$\cdots$	$\cdots$	$\cdots$

Figure 4.15: Timing Diagram for the Pipelined Version of Simulated Annealing

better than C_j based on the Metropolis criterion. The *Decide* stage validates if the configuration P_j is better than P_i based on Metropolis criterion. Clearly, it takes 3 steps to fill the pipeline.

The pipelined execution of *SA* continues in an uninterrupted fashion when there are no rejections of configurations. However, if a configuration P is rejected by the decision stage, all the configurations that have been derived from P must be flushed out from the configuration queue. This situation is analogous to the flushing of an *instruction pipeline* upon encountering a *branch* instruction in a pipelined CPU. The pipelined execution strategy may also be applied to the iterative improvement algorithm for combinatorial optimization, where a new configuration P is accepted if and only if the cost of P is strictly lower than the current cost. As a result, the probability of flushing the pipeline is much greater. Simulated Annealing, however,

also accepts bad moves on a probabilistic basis, hence reducing the probability of a *pipeline flush*. This is especially true in the high temperature zone, where almost every move is accepted.

4.7 Summary

A good placement of circuit components is an essential part of integrated circuit layout, since it determines the both area and time performance of the chip. In fixed-area layout styles such as gate array, placement determines the routability of the chip. In standard cell layout and general cell layout, a good placement reduces the interconnect area by positioning strongly connected components close together, hence avoiding long wires. Reducing long wires also reduces switching delays, hence improving the time performance of the circuit. In this chapter, the *objective function* of circuit placement was the total wire length, which is estimated by totaling the expected length of each net i, assuming that i is routed optimally by the wiring stage which follows the placement. Total wire length is a satisfactory objective function, since it is easy to compute and is directly related to the interconnect area and the average switching delay of the circuit; but it is only an approximate estimate of the actual wiring requirements of the chip, since the wiring program cannot generally route each net optimally. Recently, objective functions that measure the time performance of the chip more directly have been sought. The maximum wire length ω_{max} and the minimum wire length ω_{min} are two such measures. It is important to maintain ω_{max} below a maximum value and ω_{min} above a minimum value so that the switching delays of the slowest and the fastest signals in the circuit can be kept under control; this is important for the circuit to operate correctly. Placement of circuit components in order to constrain ω_{max} and ω_{min} is known as *performance directed placement*, and is the topic of further research. The existing placement methods need to be reexamined for the new objective function. Performance directed placement algorithms can be expected to be as compute-intensive, if not more, as existing placement tools and are good candidates for parallel processing.

This chapter has presented parallel algorithms for several traditional approaches to circuit placement, such as Iterative Improvement, Simulated Annealing, Min-cut, and Eigenvector-based placement techniques. Other authors have reported parallel algorithms for placement based on Simulated Annealing and Iterative Improvement; all these algorithms fall into the "multiple perturbation multiple decision" category in our classification. MPMD algorithms are well suited for multiprocessor implementation. But their main disadvantage is that they are not guaranteed to converge. We adopt the MPSD approach, where multiple perturbations are followed by a single centralized decision step. The MPSD approach is suitable for implementation on both SIMD and MIMD computers. It also eliminates the problems of erroneous decisions and oscillation.

Parallel algorithms based on Min-cut and Hall's eigenvector method are contributions of this dissertation. We have also presented a new algorithm for placement based on the divide-and-conquer paradigm, which is suitable for regular design styles such as gate array and standard cell. This algorithm is inherently parallel and can be efficiently implemented on both SIMD and MIMD platforms. We have presented two parallel implementations, one on an Orthogonal Array SIMD architecture and another on a Hypercube MIMD architecture.

There are many other placement techniques which were not discussed in this chapter; these include expert systems, branch-and-bound methods, learning automata and artificial neural networks. Each of these techniques is known to result in good solutions. However, these techniques have not received much recognition due to their compute-intensive nature. Parallel computing can change this scenario. Future research in this area must focus on these unconventional techniques. This point is further elaborated in Chapter 7.

Chapter 5

Parallel Routing

As seen in Chapter 1, routing a VLSI circuit calls for a hierarchical approach. Small circuits can be routed using a direct approach such as Lee routing. Lee's algorithm considers the nets one at a time and finds the shortest possible route for each net. The difficulty with this approach is that nets routed earlier may create blockages and make it impossible to route the remaining nets. When this happens, some of the routed nets are ripped up in order to remove the blockages. The blocked nets are now routed, and an attempt is made to reroute the nets that were ripped up. There is no guarantee that the rip-up-and-reroute technique will yield 100% routability. A better approach is to examine all the nets at once and plan a rough path for each net so that 100 % routability is ensured even before tracks are assigned to any net; then, in a second phase, the rough path for each net is refined to a detailed route consisting of actual wiring tracks. This two-phase approach is called hierarchical routing. The first phase of hierarchical routing is called global routing and the second phase is known as channel routing.

Global routing essentially decomposes the routing problem into a number of channel routing problems. A global router can be designed using the integer linear programming approach introduced in Section 1.6.4. But the resulting integer program has a large number of variables and constraints. Therefore, a hierarchical approach is more appropriate; this proceeds by dividing a large circuit into smaller regions and solves the global routing problem in each of the regions. If a net is completely with a region, it need not be considered further. If a net does

cross the boundary of two regions, the pieces of its global path must be suitably pasted together. This approach is called *hierarchical global routing*, and must be distinguished from *hierarchical routing*. Since a channel represents a smaller routing problem, Lee's algorithm can be used to route a channel. However, the problem of channel routing has some additional structure to it; the terminals lie along the boundary of a rectangle. Several algorithms have been designed to take advantage of this structure. Lee's algorithm still finds applications in custom routing, especially in completing the routes for some difficult nets which could not be manually routed. Similarly, when other autorouters fail to route a chip completely, Lee's algorithm with its rip-up-and-reroute feature is used as the last resort.

Recognizing the fact that Lee routing, global routing and channel routing are all compute-intensive problems, parallel processing has been applied to solve these problems. This chapter will begin with a review of earlier work in parallel routing. Next, we present a parallel algorithm for global routing based on a "cut-and-paste" technique. This algorithm uses the Orthogonal Array (Appendix A) as the target architecture. Olukotun and Mudge have recently reported a similar global routing algorithm which runs on a Hypercube [OM89]. Compared to [OM89], which results in a speedup of $O(\log P)$ using P processors, the advantage of our algorithm is that it yields $O(P)$ speedup by utilizing the processors better. We also present a parallelization technique for channel routing. Two previously reported parallel algorithms for channel routing make use of the small-grain parallelism in the problem and attempt to use multiple processors to route a single channel. On the contrary, our algorithm exploits the large-grain parallelism in the channel routing problem, and multiple processors are used to route several channels concurrently. We show that our algorithm performs exceedingly well for floor plans that do not lead to cyclic constraints in channel ordering; sliced floor plans, which are used popularly, are examples of such floor plans. In the next chapter, we will model the three-layer channel routing problem as a node coloring problem on interval graphs and present a parallel algorithm for the task; an implementation of the algorithm on Intel iPSC/2 hypercube will also be presented.

Routing Problem	Architecture	Interconnection Network	Algorithm	S/P or G/P?	Ref
Global Routing	MIMD	Hypercube	Integer Prog.	G/P	[OM89]
		Bus	Lee-like	G/P	[Ros88]
		Hypercube	Lee-like	G/P	[OM87]
	SIMD	Orthogonal Array	Cut-and-Paste	S/P	[RS89a]
Channel Routing	MIMD	Bus	And-Or Search	G/P	[Zar88]
		Hypercube	Annealing	G/P	[BB88]
		Flexible	Multiple Channel Routing	G/P	[RS91a]
Lee Routing	SIMD	2-D Mesh	Lee's Algorithm	S/P	[BS80]
	Pipeline			S/P	[W+88]
	MIMD	2-D Mesh		S/P	[S+86]
		2-D Mesh		S/P	[HN82]

Table 5.1: Classifying Routing Accelerators. G/P and S/P are used to indicate General-Purpose and Special-Purpose, respectively.

5.1 A Taxonomy of Parallel Routing Systems

Table 5.1 shows a classification of existing parallel routing systems. Most of the early work in the area has focussed on parallelizing Lee's routing algorithm through the use of hardware accelerators [HN82, BS80, S+86]; the essential idea behind each of these systems is to parallelize the wavefront expansion phase of Lee's algorithm [LB88]. An individual processor is assigned to each cell in the two-dimensional routing grid. The processor which is assigned to the source pin S begins the wavefront expansion phase by simultaneously sending its label to its north, west, east and south neighbors. Each processor which receives a label updates the label with its own stamp and passes it on to its neighbors. When the label reaches the processor assigned to the destination pin D, a *halt* signal is generated. A sequential implementation of wavefront expansion requires $O(p^2)$ time, where p is the length of the shortest path from source S to destination D. The parallel algorithm uses $O(N^2)$ processors for an $N \times N$ grid and takes $O(p)$ time to find a path of length p. More recently, Sahni and associates [W+88] have presented a slightly different

parallelization of Lee's algorithm. They make use of pipelined processing in the following way. Lee's algorithm is a breadth-first search procedure and operates by storing a queue of cells that need to be examined before the wavefront touches the destination D. The core of the wavefront expansion process consists of three subtasks – deleting a cell from the queue, computing its neighbors and storing the neighbors in the queue. The basic idea in [W+88] is to recognize that the three subtasks mentioned above can be executed in a pipelined fashion. At most three new neighbors are generated at one time (with the exception of the source point S which can have four neighbors). Therefore, three pipelines are used to simultaneously expand the wavefront in three different directions.

Parallel algorithms for hierarchical routing have begun to appear more recently. Olukotun and Mudge have described a parallel global routing procedure which uses an integer linear programming formulation of the problem [OM89]. The routing area is geographically partitioned into smaller areas. This divides a large global routing problem into several smaller ones, each of which can be solved as an integer linear program. The authors used an NCUBE/Six hypercube with $P = 64$ processors and obtained a speedup of 6 for most problem instances. The low speedup is explained by our analysis of their procedure. The authors used a quad-tree decomposition to divide the routing region. In other words, each decomposition step divides the region into four smaller regions. There are k such decomposition steps, leading to 4^k regions. A single processor is assigned to each subregion; thus $4^k = P$. Let T be the time taken by one processor to carry out quadrant decomposition. The parallel algorithm requires $T_p = kT = \log_4 PT$ time to complete. A sequential implementation requires $T_s = \sum_{k=0}^{\log_4 P - 1} 4^k \times T$. The speedup $\frac{T_s}{T_p}$ can be shown to be equal to $\frac{1}{3} \cdot \frac{(P-1)}{\log_4 P} = O(\frac{P}{\log P})$.

Zargham [Zar88] has reported a parallel algorithm for two-layer channel routing which runs on a Sequent Balance shared-memory multiprocessor with 6 computing elements. The channel routing algorithm is formulated as an AND/OR search like in several AI programs. The search proceeds in a tree-like fashion. The parallel algorithm achieves speedup by conducting the AND/OR search in parallel. All the

processors in the machine are treated as a global resource pool. Whenever a node is expanded, a processor is drawn from the pool; when a node terminates, the assigned processor is returned to the global pool. Since the search tree can be imbalanced, it is not easy to achieve load balancing. For a channel with 12 columns, Zargham has reported a speedup factor less than 2 when 6 processors are used. A theoretical analysis of speedup is difficult and is an open problem.

Brouwer and Banerjee [BB88] have presented a two-layer channel routing algorithm based on Simulated Annealing. Their algorithm has been implemented on an iPSC/2 with 16 processors. An initial track assignment is obtained using the left-edge algorithm [HS71]. Such a track assignment is optimal in terms of the number of routing tracks, but may not be a valid assignment due to conflicting vertical constraints (explained in Chapter 1). A "move" consists of interchanging the track positions of two nets. A move can either decrease or increase the number of conflicts in vertical constraints. The Metropolis criterion [KGV83] is used to accept or reject a move. Multiple processors are used to simultaneously make several moves. The speedup results presented in [BB88] is based on empirical study. Interestingly, the authors report superlinear speedup for one of their test cases. Theoretical analysis of the speedup behavior of the algorithm is still open. The largest test case involves about 200 subnets. It is not possible to predict the performance of the algorithm for larger test cases by considering the empirical evidence presented by the paper.

In the next two sections, we present our techniques for parallel hierarchical routing.

5.2 Parallel Global Routing

Hierarchical global routing has been successfully used in gate array, standard cell and custom chips [HS85, L+87, BP83]. The process consists of dividing the chip into smaller regions, routing within the regions, and pasting the solutions together. The global routing problem was described in Section 1.6.4, along with an integer programming formulation of the problem. The size of the resulting integer program can

be too large for VLSI circuits. Therefore, Hu and Shing proposed a decomposition algorithm for global routing [HS85], where a large integer programming problem is decomposed into several smaller ones. We have adapted this algorithm for a parallel implementation [RS89a].

Hu and Shing used a relaxed integer programming approach to global routing [HS85]. Since such a formulation results in an unacceptably large linear program, the authors proposed a *Cut-and-Paste* technique which partitions the area of the chip into global cells such that a hierarchical solution to the global routing problem can be obtained. The algorithm was inspired by the Dantzig-Wolfe decomposition method for solving large linear programs [BJ77]. There are three phases in the algorithm, *cutting*, *region routing*, and *pasting*. We have shown that the Cut-and-Paste algorithm (CAP) is well suited for a parallel implementation on the *orthogonal array* architecture [RS89a]. A modified version of cut-and-paste algorithm (MCAP) was used in [RS89a]. The rest of this section describes the MCAP algorithm. For a description of the orthogonal array, the please refer to Appendix A.

Consider a vertical cut-line which partitions the chip into left and right regions. With reference to this cut-line, each net may be classified into three groups : nets with all their terminals to the left of the cut-line, nets with all their terminals to the right of the cut line, and nets which are cut by the vertical line. R is defined as the ratio of number of nets cut by the vertical line to number of available tracks across the vertical line. The nets can be routed only if $R \leq 1$. The vertical cut-line is chosen to maximize the ratio R since it is not desirable that a vertical line should cut the same net more than once. After such a cut-line c has been located, the regions to the left and right of c are further partitioned using horizontal cut-lines. Figure 5.1 shows a chip cut by four cut-lines $c_0, \cdots, c_3$. The atomic regions are marked by alphabets $A, \cdots, E$ in Figure 5.1. The cutting process is modeled by a binary tree, called *partition tree*. The root of the partition tree represents the entire chip and the two children of the root represent the two regions to the left and right of cut-line c. The leaves of the partition tree correspond to *atomic regions* which need not be

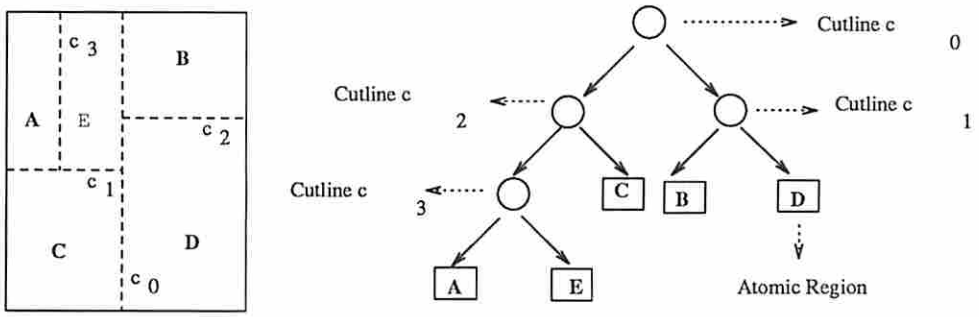

Figure 5.1: The Cutting Phase in Global Routing

further partitioned. Each atomic region has no more than M nets, where M is a constant.

Region routing refers to the process of routing nets within an atomic region. To optimally route a P-pin net, we must construct a *Rectilinear Steiner tree* containing the P points. When P is 3, 4, or 5, Hanan [Han66] describes simple techniques to construct the corresponding Steiner trees. The process of cutting may have split a single net into several pieces, which must be now connected using wires of minimum length. This step is called Pasting. While [HS85] performs pasting after region routing, we have developed a modified cut-and-paste (MCAP) algorithm which carries out the pasting step *before* region routing [RS89a].

5.2.1 Parallel Cut-and-Paste Routing

In Figure 5.2, the net A is separated by the vertical cut-line c; the net A consists of a set of pins A_l to the left of c, and a set of pins A_r to the right of c. Pasting involves connecting some pin $p_l \in A_l$ and a pin $p_r \in A_r$ such that the length of the pasting segment is minimized and the number of bends (vias) in the pasting segment is minimized. The conventional cut-and-paste algorithm proceeds in a "depth first" fashion, introducing cut-lines until atomic regions are obtained. As a result, pins p_l and p_r are separated by one or more cut-lines before pasting commences. The conventional pasting step will introduce zero or more *virtual pins* (images of p_l and p_r) and connect p_l and p_r through small segments of wire passing through these virtual pins.

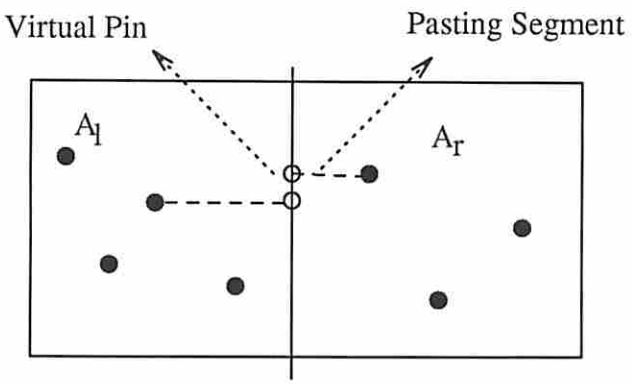

Figure 5.2: Pasting before Region Routing

In the modified scheme, as soon as the bisection step is completed, the pasting step is carried out by identifying pins p_l and p_r for each net separated by the cut-line. The modified cut-and-paste algorithm (MCAP) has a more natural parallelization on the orthogonal array. Since MCAP does not support the concept of virtual pins, it uses lesser number of vias than the conventional CAP algorithm. In a parallel implementation, the MCAP algorithm requires lesser inter-processor communication than the CAP algorithm. The pasting step in CAP requires the routing information to be passed up from the atomic regions to higher levels of the routing hierarchy, resulting in complex and expensive data movement operations among processors. The pasting scheme in the MCAP algorithm eliminates this communication overhead. The modified algorithm is shown in Figure 5.3.

The parameters M and P used in the MCAP algorithm are chosen as follows. M and P are related by $P = A/M$, A being the area of an atomic region. Since the grid area of L^2 is cut into L approximately equal regions, $A = L$. Thus, $P = L/M$. As discussed in the context of region routing, the constant P is selected to be a small positive integer, say 5. Knowing the number of processors in the array, the above relation can be used to fix the parameter M.

5.2.1.1 Parallel Cutting Step

In this section, an $\ell \times \ell$ grid graph is assumed as input. The grid consists of ℓ^2 cells; the cell in row i and column j is denoted G_{ij}. The technique consists of assigning a

```
procedure MCAP (ρ : region);
begin
    if ( region ρ has more than M nets ) or
    ( there exists a net with more than P pins in the region ρ) then do
    begin
        Step 1 :
            Examine the cut-lines in region ρ;
            Find the cut-line c which has maximum ratio R ;
        Step 2 :
            Using c, cut the region ρ into two parts;
            Identify the two parts ρ1 and ρ2 as the children of region ρ
        Step 3 :
            for each net ℵ in region ρ do
                if (net ℵ is separated by the cut-line c ) then
                    Let ℵ1 in region ρ1 and ℵ2 in region ρ2
                        be the subnets of net ℵ;
                    Find a pin π1 ∈ ℵ1 and a pin π2 ∈ ℵ2
                        such that the Manhattan Distance dπ1π2 is minimal;
                    Connect pins π1 and π2 using this minimal path;
                end { if };
        Step 4 :
            call MCAP (ρ1);
            call MCAP (ρ2);
    end else
        Step 5 :
            for each net ℵ in region ρ do
            begin
                Let there be σ pins in net ℵ;
                Find the optimal Rectilinear Steiner Tree of σ points;
            end {for}
end procedure
```

Figure 5.3: The Modified Cut-and-Paste Algorithm

single processor to each column (row) of the grid graph. Thus ℓ processors and ℓ^2 memory elements are required in the orthogonal array. If the input is a $p \times q$ grid, we set $\ell = max(p, q)$. The memory element M_{ij} corresponds to the cell G_{ij} of the grid. If the global cell G_{ij} is a *terminal cell* in the grid, i.e., there exists a pin of some net in the cell G_{ij}, the corresponding *net id* is stored in the memory element M_{ij}. Provision is made for several pins to exist in the same global cell. The horizontal channel capacity h_{ij}, and the vertical channel capacity v_{ij} associated with the cell G_{ij}, are stored in the memory element M_{ij}.

Since processor PE_i has access to the row i and column i of the 2-D array of memory elements, it is possible to concurrently compute the ratio R for each row (column) of the grid structure during a horizontal (vertical) bisection. Each PE_i has a register $\Re_i$ associated with it. The jth bit of $\Re_i$ is set to one if and only if the net j has a pin in row i or column i. It is easy to see how the processor PE_i can update its register $\Re_i$ in $O(\ell)$ time. Suppose that vertical bisection is in progress. Each processor starts by clearing the register $\Re$. PE_i will search all the memory elements in column i and set $\Re_{ij}$ to 1 if some memory element in column i contains a pin belonging to net j. A similar row computation is necessary during horizontal bisection.

Let a *line* denote a row or a column. Let r_i (l_i) represent the set of all nets which have pins to the right (left) of column i. Similarly, let t_i (b_i) represent the set of all nets which have pins to the top (bottom) of row i. Let TC_i be the track capacity of line i. The ratio R_i for column i, $1 \leq i \leq \ell$, is given by $R_i = \frac{|r_i \cap l_i|}{TC_i}$. Moreover, $R_0 = 0$ and $R_\ell = 0$. Similarly, the ratio R_i for row i, $1 \leq i \leq \ell$, is given by $R_i = \frac{|t_i \cap b_i|}{TC_i}$.

During a vertical bisection, the register $\Re_i$ is a bit-vector representation of set $r_{i-1} \cap l_{i+1}$. Using this representation, it is possible to efficiently compute the sets r_i and l_i on the orthogonal array. Suppose that PE_i receives the contents of the $\Re$ register from the left neighbor PE_{i-1}, $1 < i \leq \ell$. A bit-wise OR operation on $\Re_{i-1}$ and $\Re_i$ gives the bit-vector representation of the set of all nets which have pins to the left of $i + 1$. This phase, where the data in $\Re$ registers moves in the right direction,

is known as the *right movement phase.* At the end of the right movement phase, each PE_i has the bit-vector representation of l_i. A similar *left movement phase* can be used to obtain the bit-vector representation of the r_is. PE_i will then compute $r_i \cap l_i$ by using a bit-wise AND. The resulting bit-vector is stored in a register ω_i of PE_i. A count of the number of 1's in the register ω_i evaluates the quantity $\mid r_i \cap l_i \mid$. The track capacity TR_i for line i can be found as

$$TR_i = \begin{cases} \sum h_{i,*} & \text{for vertical bisection} \\ \sum v_{i,*} & \text{for horizontal bisection} \end{cases}$$

where * indicates that the entire *region* must be covered by the summation.

Lemma 11 *Cutting an $\ell \times \ell$ region requires $O(\ell)$ time on an ℓ-processor orthogonal array.*

Proof : To update register $\Re_i$, each processor requires $O(\ell)$ time. Implemented in the manner described in the previous paragraphs, the right and left movement phases require $O(\ell)$ time. Similarly, the summation involved in computing TR_i can take $O(\ell)$ time. However, note that the *union* and *sum* operations are both associative, and an $O(\log \ell)$ implementation of both these operations based on the *tree computation* technique exists and is described by Alnuweiri and Kumar [AK86]. After each PE_i has computed R_i, the processors evaluate the maximum of the R values using a tree computation technique. The $TreeMax$ operation requires $O(\log \ell)$ time on an ℓ-array. Therefore, the total time complexity of the cutting step is $O(\ell + \log \ell) = O(\ell)$ ■.

5.2.1.2 Parallel Pasting

As in the previous section, an $\ell \times \ell$ grid graph is assumed, and ℓ processors. Recall that the cut-line c which maximizes R has been found at the end of the cutting step. The MCAP algorithm requires that a short pasting segment be located to connect the pieces of each separated net. As shown in Figure 5.2, this can be done by projecting $\times$ marks from each pin of the separated net onto the cut-line c. Then

two pins are selected, one on either side of the line c, which are at shortest distance from the $\times$ marks.

The pasting step has a simple realization on the orthogonal array. In what follows, a vertical bisection is assumed. Following the cutting step, the identity of the cut-line c (which we shall denote ω_c) is broadcast to all the PEs on the orthogonal array. The ω register corresponding to line c is also broadcast to all the PEs. Recall that if jth bit of the ω register is set to 1, then the net j is separated by the cut-line c. PE_i performs a bit-wise AND on its register $\Re_i$ and the received ω_c register. If the resulting bit-vector contains a 1 in its jth position, it implies that the memory column M_i contains one or more pins of net j. PE_i computes the distance from cut-line c and associates this distance d_{ij} with net j in its memory column. Finally, two minima are computed for every i for which $\omega_{c_i} = 1$ (i.e., for every separated net i). The first minima corresponds to the shortest distance among the d_{ij}s, $1 \leq i < c$ (nets to the left of c). The second minima corresponds to the nets that lie to the right of c. Suppose that the minima occur at lines c_l and c_r for some net i. The processors PE_{c_l} and PE_{c_r} may have more than one pin of net i in their memory column. However, it is sufficient if each processor chooses one pin to form the pasting segment (see Figure 5.2). In order to minimize the vertical length of the pasting segment, PE_{c_l} chooses the pin $\pi_{c_l j}$ from memory $M_{c_l j}$ for which j is maximum. For a similar reason, PE_{c_r} chooses the pin $\pi_{c_r j}$ from memory $M_{c_r j}$ for which j is minimum.

Lemma 12 *The pasting step on an $\ell \times \ell$ region requires ℓ^3 time on ℓ processors.*

Proof : There can be $O(\ell^2)$ nets in the input region. In the worst case, all these nets may be separated by the cut-line c. The computation of minima requires $O(\log \ell)$ time. Each PE_i may have to scan its entire memory column to locate the pins of the net. This requires $O(\ell)$ time. These computations are repeated for each net, and therefore complexity of the pasting step is $O(\ell^2(\log \ell + \ell)) = O(\ell^3)$ ■.

5.2.1.3 A Process Tree Computational Model

The previous section has described efficient cutting and pasting on an orthogonal array with ℓ PEs. These results are used to parallelize the entire MCAP algorithm. A computational model can be derived to describe the MCAP algorithm by associating a *process* with each node of the partition tree. The resulting tree is known as a "Cut-and-Paste process tree" (CPT). The root of the process tree operates on the entire chip of dimension $L \times L$, performing the following computations : **(a)** Cutting the chip, and **(b)** Pasting the separated nets. The root process then spawns two child processes and terminates. The child processes repeat identical computations on the two regions created by the root process. The leaves of the process tree carry out region routing. The height of the process tree is the same as the depth of recursion β in the procedure MCAP. Thus, it can be shown that the height of the process tree is $O(\log L)$ (see Lemma 13). The number of nodes in the process tree is thus $O(L)$.

In what follows, the process tree is mapped to an orthogonal array with only $O(L)$ processors. Two key observations are necessary in order to appreciate the mapping of the MCAP procedure onto the orthogonal array. First, note that processes at only one level of the CPT are active at any instant of time. Second, the size of the region handled by a process at level k is $L^2/2^k$; thus $L/2^k$ processors are required to parallelize the computations of a process at level k. Since there are 2^k processes at level k, the processor requirement of any level k is $L/2^k.2^k = L$. In other words, an L-processor array is sufficient to parallelize all the levels of CPT. The root process is assigned to the entire array of L elements. The children of the root process are assigned to two *subarrays* of size $L/2$. In general, a subarray of size $L/2^k$ is used in the parallel execution of a process at level k.

The PEs in a subarray must know the region ρ upon which they operate on. This is necessary during the horizontal and vertical movement phases of the cutting step. The *broadcast* feature of the orthogonal array is used to derive this knowledge. When a region ρ has been cut by some cut-line c, all the processors currently assigned to

the region ρ use broadcasting to exchange the information concerning c. It requires $O(\log \ell)$ time to perform a broadcast among ℓ PEs.

5.2.1.4 Performance Analysis of Parallel Routing

The following lemma is useful in estimating the time complexity of routing on the proposed hardware accelerator.

Lemma 13 *If there are n nets in the routing problem then the depth of recursion in the execution of procedure CUT-AND-PASTE is $O(\log n)$.*

Proof : The grid graph has dimensions $L \times L$ to start with. With each recursive call to procedure MCAP, the area of the input region ρ decreases by a factor of 2. Thus, the kth call of MCAP receives as input a region of dimension $L^2/2^k$. There are $O(L^2/2^k)$ nets in the atomic region. When $k = \beta$, this value is equal to the constant M. Hence $\beta = \log L$. We also know that $L = O(\sqrt{n})$; therefore, $\beta = O(\log n)$ ■.

Lemma 14 *The parallel cutting step requires $O(L)$ time on an array of L processing elements.*

Proof : Earlier discussion has shown that the cutting step on an $\ell \times \ell$ grid requires $O(\ell)$ time (see Lemma 11). Since the chip area is approximately halved by every application of the cutting step, it is clear that the kth iteration of cutting step will receive a region of area $L^2/2^k$ as input. The total cutting time spent by the router is, therefore, given by $T_{PAR-CUT} = \sum_{k=0}^{\beta} L/2^k$. From lemma 13 above, $\beta = O(\log L)$. Thus, the above summation yields $T_{PAR-CUT} = O(L)$.

Theorem 3 *A speedup of $O(L)$ is obtained by parallelizing the cutting step on an orthogonal array with L processors.*

Proof : To cut a region of dimension $\ell \times \ell$, a uniprocessor needs $O(\ell^2)$ time. This is because the processor must look at ℓ cut-lines, and it takes $O(\ell)$ time to process each cut-line. The time $T_{SEQ-CUT}$ required by a uniprocessor to execute all the cutting steps is given by the summation $T_{SEQ-CUT} = \sum_{k=0}^{\beta} L^2/2^k = O(L^2)$. The

speedup obtained by parallelizing the cutting step is the ratio $T_{SEQ-CUT}/T_{PAR-CUT}$ $= O(L)$ ■.

Lemma 15 *The total time required to execute pasting step on the orthogonal array is* $O(L^3)$.

Proof : The result follows from Lemma 12. Since it takes $O(\ell^3)$ time to paste in a region $\ell \times \ell$, the total parallel pasting time is $T_{PAR-PASTE} = \sum_{k=0}^{\beta} L^3/2^k = O(L^3)$ ■.

Theorem 4 *The speedup obtained by parallelizing the pasting step is* $O(L)$.

Proof : Consider the computational complexity of sequential pasting in an $\ell \times \ell$ region. For each net, $\times$-marks must be projected from all its pins, which could, in the worst case, be $O(\ell^2)$ in number. Since there are $O(\ell^2)$ nets, $O(\ell^4)$ time is required to paste in the given region. The total time spent by a uniprocessor in the pasting at all levels of the partition tree is

$$T_{SEQ-PASTE} = \sum_{k=0}^{\beta} (L/2^k)^4.2^k = O(L^4). \tag{5.1}$$

Combining this result with Lemma 15, its is seen that the speedup obtainable for the pasting step is $O(L)$ ■.

Theorem 5 *By routing regions in parallel on the orthogonal array, a speedup of* $O(L)$ *is obtained.*

Proof : First, observe that $O(L)$ atomic regions result from the cutting procedure. Assuming uniform net density, there are n/L^2 nets per unit area of the chip; this shows that the area of a region, A_R, is given by $A_R \geq L^2.M/n$, since there are not more than M nets in an atomic region. This shows that the number of regions does not exceed n/M which is $O(L)$. The theorem follows from the fact that L processors concurrently carry out $O(L)$ independent routing operations. It takes $O(\aleph)$ time to construct an optimal Steiner tree for each of the $\aleph$ nets, provided each net has less

Step	Sequential Time	Parallel Time	Speedup
Cutting	$O(L^2)$	$O(L)$	$O(L)$
Pasting	$O(L^4)$	$O(L^3)$	$O(L)$
Region Routing	$O(L^2)$	$O(L)$	$O(L)$

Table 5.2: Parallel Cut-and-Paste : Summary of Complexity Results

than a constant number of pins. During *parallel region routing*, the total number of routed nets is $L.M = n$. Thus, a uniprocessor requires $O(n)$ time to complete region routing. On the other hand, the parallel execution of region routing requires $O(M)$ time. The resulting speedup is, therefore, $O(n)/O(M) = O(L)$ ■.

Theorem 6 *The parallel implementation of the Cut-and-Paste procedure on an L-processor orthogonal array provides a speedup of* $O(L)$.

Since the orthogonal array has L processors, it can be used to route $O(L^2)$ nets. Table 5.2 summarizes the performance of the parallel algorithm when routing $O(L^2)$ nets.

5.3 Parallel Channel Routing

The phrase "parallel channel routing" can be interpreted in two different ways : (a) simultaneously routing several channels (inter-channel parallelism) and (b) simultaneously routing several nets within a channel (intra-channel parallelism). The first interpretation corresponds to large grain parallelism, and the second to small grain parallelism. Both these approaches will be discussed in this section. Note however that the two approaches do not exclude the use of each another. When both techniques are employed, the resulting speedups are multiplicative.

5.3.1 Routing Several Channels In Parallel

At first sight, it may seem that the channel routing problem presents a great opportunity for parallel processing since all channels can be simultaneously routed.

However, this is not the case, as illustrated below with the aid of Figure 5.4. The channels a and b in the diagram form a T intersection. Unless channel a is routed first, it is not possible to determine the pin positions on the left hand side of channel b. Therefore, channel b cannot be routed before routing channel a. A directed graph known as *channel order graph* (COG) can be constructed to reflect the constraints on the order in which channels must be processed. There is one node in the COG for every channel; a directed edge is drawn from node i to node j if (and only if) channel i must be routed before channel j. A COG is not necessarily acyclic. For instance, nodes a, b, c, d in Figure 5.4 form a cycle. One way to overcome the problem of a cycle in a COG is described in [SS84]. The procedure selects a node randomly from the cycle and estimates the width of the corresponding channel x. All channels in the cycle except x are routed. Then the procedure attempts to route x in the estimated number of tracks; if this is not possible, the entire process is repeated using a larger estimate for the width of x.

A different approach is necessary for parallel processing. By the construction of the channel order graph, two channels i and j can be scheduled for parallel routing if there is no directed *path* from node i and j. Based on this observation, it is possible to order the channels such that all the parallelism in the problem is exploited. This *channel ordering problem* is isomorphic to the *task scheduling* problem in multiprocessing systems. A task scheduling algorithm uses a *task dependence graph* to represent execution dependences between tasks. The objective of the scheduler is to minimize the number of time steps required to complete all the tasks. The scheduler deals with cycles in the task dependency graph by clustering the nodes of a cycle into a *super node*. Using the analogy of task scheduling, the following parallel algorithm can be devised for channel routing. Channels that form a cycle are grouped into a **super channel**. This eliminates cycles from the COG. The channels of an acyclic channel order graph (ACOG) can be scheduled for routing on an *as soon as possible* basis. A super channel is handled using Preas' tie-breaking approach mentioned earlier [SS84].

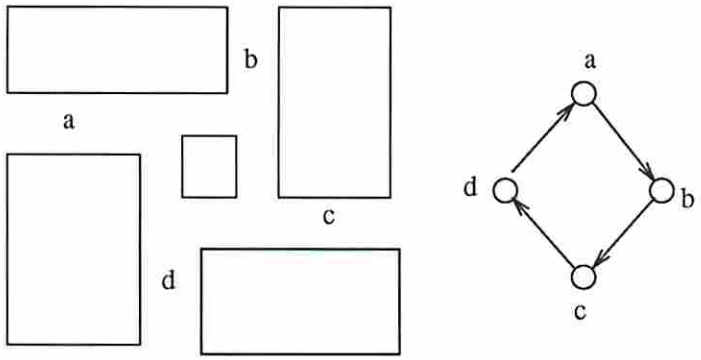

Figure 5.4: A Cyclic Channel Order Graph

5.3.1.1 Acyclic Channel Order Graphs

The cyclic/acyclic property of the channel order graph is related to the floor plan of the circuit. It turns out that an important floor planning technique known as *slicing* always results in acyclic channel order graphs. On the other hand, cyclic channel order graphs are associated with non-sliced floor plans, which are used in custom layout and general cell layout.

Slicing is a hierarchical procedure to generate floor plans for module placement. The procedure starts with a rectangle which represents the entire chip and cuts it into k slices using $k-1$ vertical cutlines, $k \geq 2$. These slices are further decomposed using horizontal cutlines. The direction of the cutlines alternates between vertical and horizontal. An example of a slicing structure with $k = 2$ is shown in Figure 5.5, where cutlines $c_1, \cdots, c_7$ divide the chip into 8 slots $M_1, \cdots, M_8$. We associate a k-ary tree with a slicing structure, where non-leaf nodes represent cutlines and leaf nodes represent slots. The edges of the tree are directed toward the parent node. Starting from the slicing tree S, the (acyclic) channel order graph G is obtained by deleting the leaf nodes of S. Figure 5.6 shows the ACOG for the example of Figure 5.5.

The directed edges of an ACOG represent the routing dependences in the channels. These dependences flow from the leaf nodes of the ACOG toward the root (see Figure 5.7). Using this fact, we have designed a simple algorithm for parallel channel routing [RS91a].

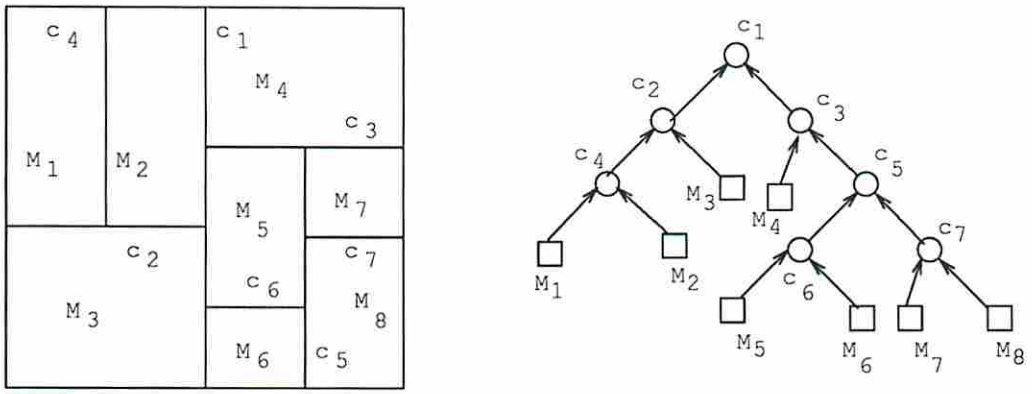

Figure 5.5: A Slicing Structure for $k = 2$

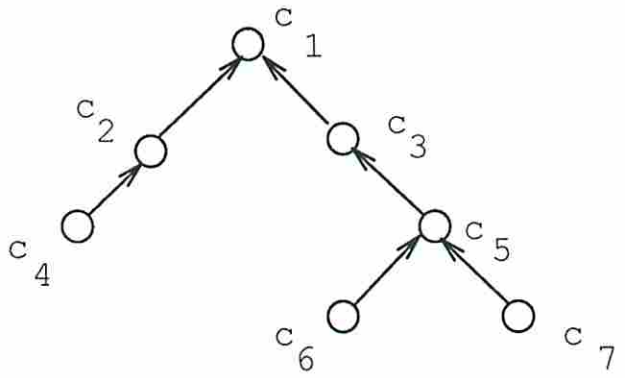

Figure 5.6: The Channel Order Graph for The Running Example

```
procedure ScheduleChannel(G);
{G is an acyclic channel order graph}
begin
        if G has a single node then begin
          Schedule G for routing
          TimeStep = TimeStep + 1
          return ;
        end else begin
            Schedule all the leaf channels of G for concurrent routing.
            TimeStep = TimeStep + 1
            Delete all the leaf nodes of G.
            Call ScheduleChannel(G);
        end
end
```

Figure 5.7: A Channel Ordering Program for an Acyclic COG

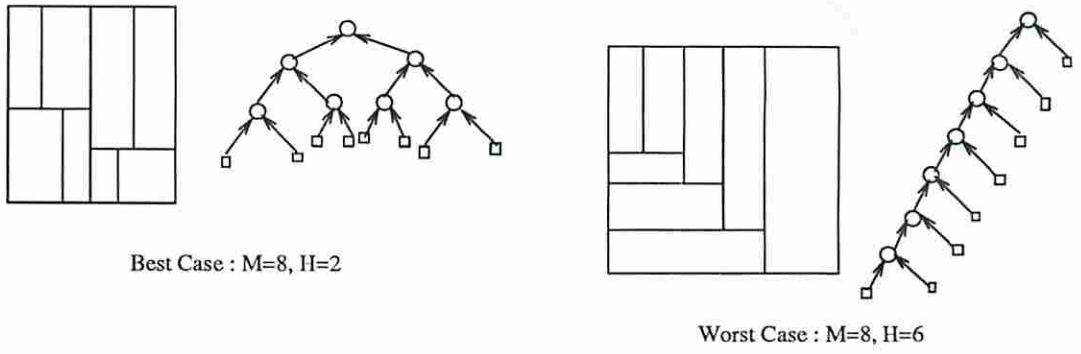

Figure 5.8: Best and Worst Case Heights of Channel Order Tree

5.3.1.2 Analysis of scheduling procedure

In the subsequent analysis, it is assumed that the slicing degree $k = 2$.

Fact 1 *At the termination of procedure ScheduleChannel, the value of TimeStep is equal to the height H of the input channel order graph.*

If n_0 is the number of leaf nodes in the slicing tree S, and n_2 is the number of nodes with degree 2 in S, then $n_0 = n_2 + 1$. Recall that n_0 is the number of circuit blocks, M, and n_2 is the number of channels, r. It follows that there are $M - 1$ channels in the floor plan. The depth of the ACOG depends on the slicing structure. The best case, where $H = \log_2 M - 1$, occurs when S is a full binary tree. The worst case height of $M - 2$ corresponds to a skewed slicing tree. These ideas are illustrated in Figure 5.8. The average value of H is a measure of the parallelism in the channel scheduling problem.

$$E[H] = \sum_{h=\log_2 M-1}^{M-2} hp_h$$

where p_h is the probability of the event $H = h$. The probability measure must be taken over the set of all possible non-isomorphic floor plans. We conjecture that $E[H]$ is $O(\sqrt{M})$, but no mathematical analysis is currently available.

5.4 Summary

Routing can be solved using a hierarchical approach or a direct approach. Lee routing is an example of the direct approach; a grid is superimposed on the routing surface and nets are routed using a shortest path algorithm. The difficulty with such a direct approach is that the order of routing the nets is hard to determine. The ordering determines the routability of the circuit as well as the total wire length. A hierarchical approach reduces this difficulty considerably by routing the circuit in two stages. The global routing stage plans a rough path for each net; a path is stated as a sequence of channels through which the net will be routed. The channel routing stage determines the exact routing tracks for each net that passes through a channel. This chapter discussed parallel algorithms for global routing as well as channel routing.

Several problems related to routing were not considered in this chapter. The next chapter will discuss the via minimization problem and a different version of the channel routing problem under the assumption that pin locations can be reassigned. Different routing models, design styles and technologies lead to different routing problems. River routing, single row routing, switchbox routing, gridless routing, over-the-cell routing, segmented channel routing and 45° routing are some special routing problems. Most of these problems are found described in [LB88]. Finding the routing order for nets, finding the order in which channels must be routed, compacting a given layout, and verifying a given layout for design errors are routing-related problems [LB88]. All these problems involve combinatorial optimization, are NP-complete, and have been solved using heuristic techniques. Investigating parallel algorithms to solve these problems is a fruitful area of research.

Chapter 6

Perfect Graphs and their applications to Layout

This chapter considers parallel algorithms for a class of graphs known as *perfect graphs*. A graph is said to be perfect if its chromatic number is equal to its clique number. Many optimization problems, such as graph coloring, are solved in polynomial time for perfect graphs. These optimization problems are NP-complete for general graphs. Some examples of perfect graphs are interval graphs, triangulated graphs and permutation graphs. Perfect graphs have several applications in the design of electrical circuits, as explained in this chapter. In particular, interval graphs have applications in three-layer channel routing, gate matrix layout, and several problems in circuit synthesis. Permutation graphs have applications in topological via minimization. We describe parallel algorithms for several optimization problems on both interval graphs and permutation graphs.

6.1 Perfect Graphs

The chromatic number $\lambda(G)$ of a graph G is the minimum number of colors required to obtain a proper coloring of G. The clique number $\mu(G)$ of a graph G is the size of a maximum clique of G. It is straightforward to see that $\lambda(G) \geq \mu(G)$ for all graphs. For certain classes of graphs, known as *perfect graphs*, $\lambda(G) = \mu(G)$. Interval graphs and permutation graphs are both examples of perfect graphs.

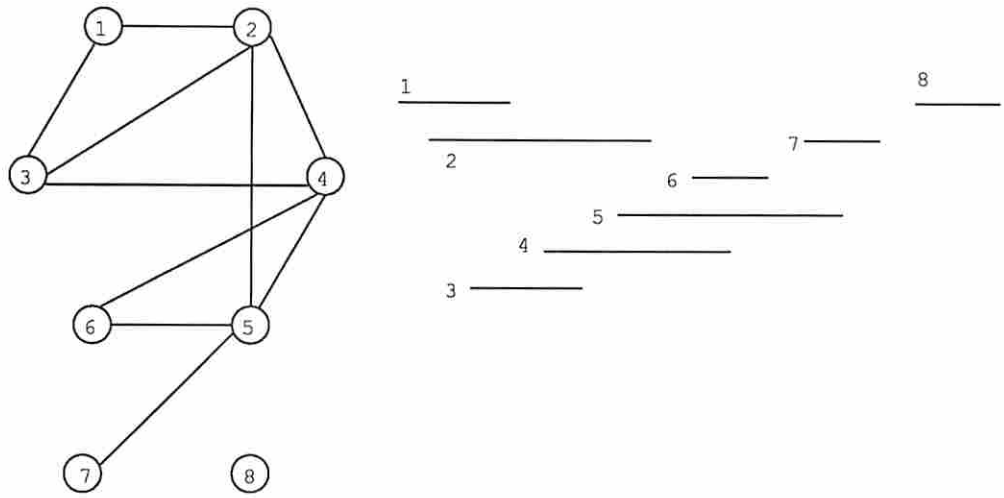

Figure 6.1: An Interval Graph and Corresponding Intervals

6.1.1 Interval Graphs

An undirected graph $G = (V, E)$ is an *interval graph* if, for each vertex $v \in V$, there exists a finite closed interval $I(v)$ such that $(u, w) \in E$ would imply that $I(u) \bigcap I(w) \neq \phi$. An example of an interval graph is given in Figure 6.1. Nodes 1 and 2 in the example share an edge; the intervals 1 and 2 that correspond to nodes 1 and 2 are also shown in the figure and are seen to have a non-zero intersection. The interval 8 in the figure does not intersect with the rest of the intervals. Accordingly, node 8 in the interval graph has no edges originating from it. Given a graph G, it is not straight forward to decide if it is an interval graph or not. There is a polynomial-time algorithm for this purpose and is described in Golumbic's book [Gol80]. However, it is straight forward to construct the interval graph for a given set of line intervals. A node is created for each line interval and an edge is added between nodes i and j if (and only if) intervals i and j overlap. Therefore, a set of line intervals can form an alternate representation of an interval graph. In all the applications that are considered in this chapter, the line intervals will be known before hand. Thus our algorithms for interval graphs can conveniently use the interval representation.

6.1.2 Permutation Graphs

An undirected graph G on N nodes is a *permutation graph* if there exists a permutation π of $\{1, 2, \cdots, N\}$ such that G is isomorphic to the *inversion graph* of π. Given a permutation π, its inversion graph $G[\pi] = (V, E)$ is constructed as follows.

$$\begin{aligned} V &= \{1, 2, \cdots, N\} \\ e_{ij} \in E &\Leftrightarrow i > j \text{ and } j \text{ appears before } i \text{ in } \pi \end{aligned}$$

Figure 6.2 shows a graph on 6 nodes. This graph is a permutation graph because it is also the inversion graph of the permutation 436152. To see how, consider the number 4. In the natural sequence 123456, the number 4 appears after 1,2 and 3. In the permutation 436152, the number 4 appears in the wrong place with respect to 1,2 and 3. Therefore, the following three edges must exist in the inversion graph – 41, 42 and 43. Similarly, the other edges in the graph can also be accounted for. Figure 6.2 also shows the *permutation diagram* for the graph. This is obtained by writing down the permutation 123456 on the top and the permutation 436152 on the bottom and then connecting the points that have the same number. The permutation diagram, permutation graph and the permutation itself are all alternate representations for the same information. Given any one of these representations, the others can be derived. For example, Golumbic describes an algorithm to arrive at the permutation starting from the permutation graph [Gol80]. The permutation representation is undeniably the simplest representation for computer programs since it can be stored as an array. In the applications of permutation graphs which are considered in this chapter, the permutation representation is used.

6.1.3 Coloring Perfect Graphs

Obtaining an optimum node coloring is a hard optimization problem for general graphs. However, interval graphs and permutation graphs can both be optimally colored in deterministic polynomial time. These problems are of interest since they have applications to several problems in VLSI design. The following section will

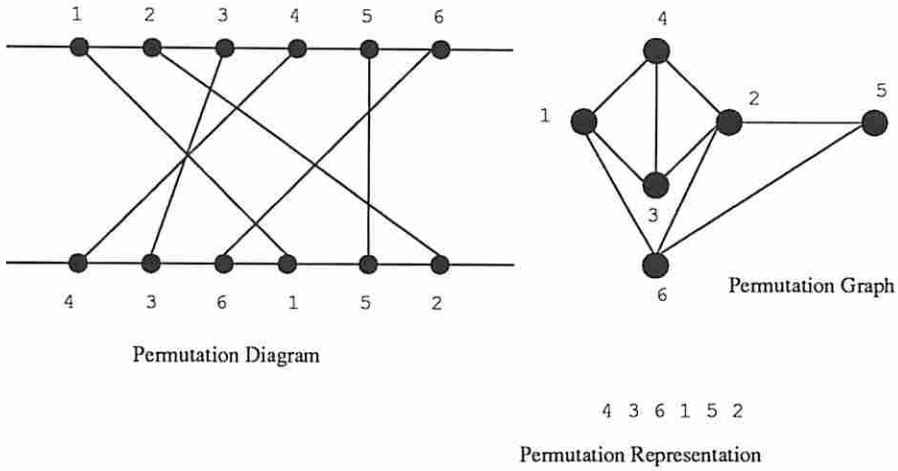

Figure 6.2: A Permutation Graph and the Corresponding Permutation

illustrate applications of interval graphs and provide a parallel algorithm for coloring interval graphs. Section 6.3 will look at applications of permutation graphs and indicate a parallel technique for coloring permutation graphs.

6.2 Interval Graphs

6.2.1 Applications

Several important problems that arise in VLSI design automation are related to interval graphs. Examples are channel routing, gate matrix layout, register allocation and task scheduling. Channel routing has been discussed in Section 1.6.4. Gate matrix is a layout style for building blocks of integrated circuits. Circuit blocks such as arithmetic/logic units, decoders, encoders, multiplexers and demultiplexers are used frequently in digital system design. Therefore, it is useful to design them carefully and store their layout in a cell library. This is indeed the approach used in standard cell design. When designing a large system using these blocks, their layout can be copied from the cell library. But there is one problem with this approach. When design rules change (e.g. owing to improvements in VLSI fabrication process) the cell library becomes obsolete. In a highly dynamic environment, it pays to use the "cell generation" approach, where a *behavioral description* of the cell, rather than

its layout, is stored in the library. A cell generator is a program that compiles the behavioral description of the circuit into its layout. Thus cell generation is a form of "low level synthesis." Gate matrix is a layout style often used by cell generators for CMOS circuits.

In principle, "high level synthesis" of a system is similar to cell generation. A high level synthesis program accepts as input the behavioral description of a system and generates an efficient implementation of the system. A typical form of input to a synthesis program is a data flow graph, which is a directed graph with data operators for nodes. A directed edge connects node i to node j if the operation i must precede operation j. The data flow graph captures the behavior of the system. A synthesis program analyzes the dependences among various data operations in the flow graph and *schedules* these operations (tasks) for execution. A schedule is an itinerary specifying in what order the operations must be carried out. The synthesis program takes advantage of the concurrency in the flow graph to arrive at a good schedule.

A synthesis program also performs hardware allocation. Hardware components such as registers, memory ports, arithmetic and logic circuits and busses must be used as building blocks of the system. Since resources are expensive, it is mandatory to use as few of each resource type as possible. Hardware units can be reused during the course of algorithm execution. For example, if there are two addition operations in the algorithm, a single adder can be used to carry out both the additions. Similarly, registers can be reused. A register is employed to store the value of a variable. When a variable ceases to exist, the register to which it is bound can be used to store another variable.

6.2.2 Interval Representation

Channel routing, gate matrix layout, task scheduling and hardware allocation – these seemingly diverse problems are all reincarnations of the same fundamental graph problem, namely, *interval graph coloring.* In order to substantiate this claim, the concept of *interval* and *color* must be identified in the context of each of these

problems. Consider channel routing first. The relation between channel routing and interval graphs has already been mentioned in Section 1.6.4. The horizontal portion of a two-point net is known as the interval (or span) of the net. The corresponding interval graph has as many nodes as there are nets. Two nodes u and w are connected if (and only if) the corresponding intervals overlap; nodes u and w must be placed on separate tracks to avoid short circuit. Therefore, the concept of color corresponds to a track. In order to minimize chip area, it is usually required to use the minimum number of tracks to wire all the nets. But for vertical constraints, which have been introduced in Section 1.6.4, the problem of channel routing is identical to coloring the horizontal constraint graph. However, there are instances of the channel routing problem in which there are no vertical constraints. The following examples illustrate two such cases.

Example 6.1 :

The channel routing problem was originally discovered by Hashimoto and Stevens when they were designing layouts for printed circuit boards [HS71]. In a PCB environment, vertical constraints can be overcome by the use of horizontal jogs. Consider the example channel shown in Figure 6.3. At column 3, there is a vertical constraint that states net 3 must be placed above net 2. This has been avoided by means of a horizontal jog in the route for net 3. On a PCB, the space between two pins of a component is sufficient to introduce such a jog. Thus vertical constraints can be ignored.

Example 6.2 :

Consider the problem of three-layer channel routing with the following routing model. Only horizontal and vertical wires are allowed. All horizontal segments of wire are placed in layer 2 (the middle layer). A vertical segment of wire required to connect a net to the top of the channel is placed in layer 1. A vertical segment is placed in layer 3 if it connects to the bottom of the channel. Figure 6.4 shows a channel wired using this routing model. No vertical constraints can ever occur in this routing model, since vertical segments that go to opposite sides of the channel are placed in different layers.

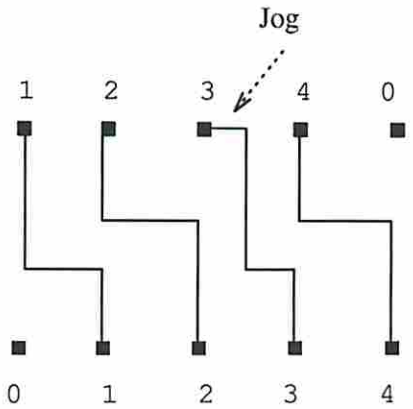

Figure 6.3: Using jogs to avoid vertical constraints in a PCB channel.

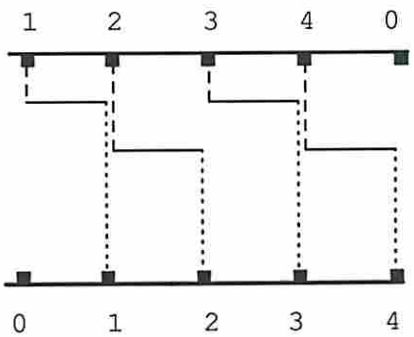

Figure 6.4: A channel wired using three layers. Wires on layer 1 are shown solid. Wires on layers 2 and 3 are shown dotted and dashed respectively. The channel density for this example is 2. Using 2 layers and the Manhattan routing model, 4 tracks are required to route this channel.

In the register allocation problem, an interval corresponds to the life time of a data variable. Registers correspond to colors. Two variables which are simultaneously alive may not be assigned to the same register. Since not all variables are active at any period of time, a small number of registers can be dynamically assigned and deassigned to hold the values of the variables. For a given set of variables, it is required to minimize the number of registers.

In the task scheduling problem, an interval corresponds to the duration of a task. A processing element, on which a task is scheduled for execution, is analogous to a color. Two tasks that are simultaneously active cannot be assigned to the same processor. Under this constraint, it is necessary to minimize the number of processors required to complete all the tasks.

Finally, consider the Gate Matrix layout problem. The term *gate* in Gate Matrix stands for a transistor gate and should not be confused with a logic gate. As the name signifies, a gate matrix consists of intersecting rows and columns. The columns are composed of polysilicon material, whereas the rows are composed of diffusion. At the intersection of a row and a column is a potential transistor. The columns serve as gates of the transistor. A CMOS NAND gate, implemented in the gate matrix style, is shown in Figure 6.5. The input to the gate matrix layout problem consists of a list of nets, each net being a set of gates that must be connected. The first step in the layout process is to determine a permutation of columns (gates). The second step is to implement the nets using the minimum number of tracks. It can be shown that the second step is an instance of interval graph coloring. The interval here corresponds to the horizontal segment of wire used to implement a net. Nets that do not intersect can be placed on the same track. Thus a track corresponds to a color. One wishes to minimize the number of tracks, so that the area occupied by the layout is minimized.

6.2.3 Serial Algorithm

A sequential algorithm for coloring interval graphs is first described. This algorithm uses the interval representation for interval graphs. Each interval is associated with

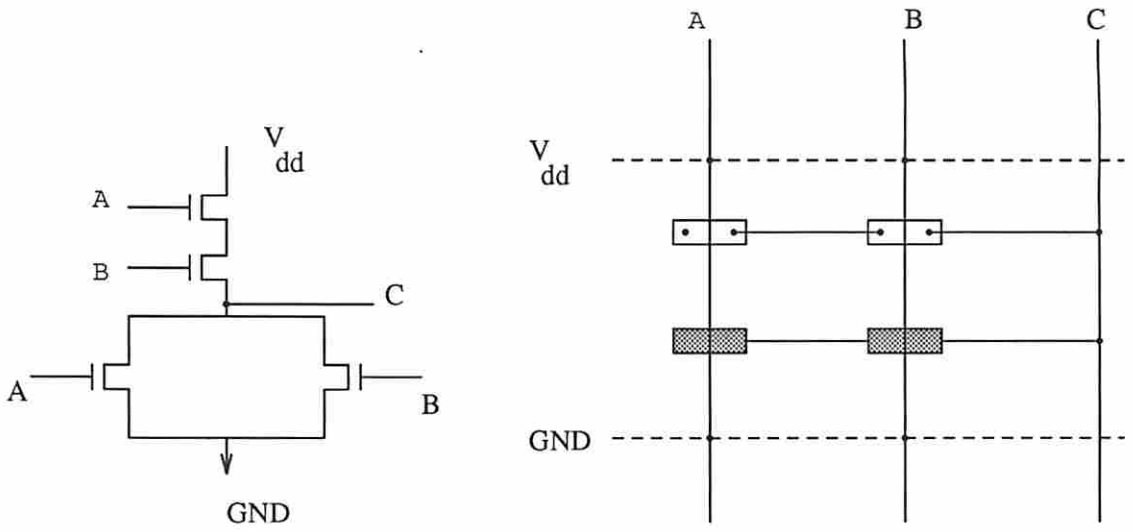

Figure 6.5: A CMOS circuit implemented in Gate Matrix style. Diffusion is shown in rectangular boxes. The boxes filled using gray filling represent N-type diffusion, whereas empty boxes represent P-type diffusion.

an interval number and is represented by a pair of end points. The end points are stored as an array of records, called POINTS. The two end points of interval i are stored in POINTS$(2i-1)$ and POINTS$(2i)$ respectively. Each entry in the POINTS array consists of the following fields – the interval number, the type of the end point (left or right), the position of the end point, and the color assigned to the end point. The last field is initially empty and is the output of the algorithm. The algorithm consists of the following two steps [GLL79].

1. Sort the array POINTS on the 'position' field.
2. **for** $i = 1$ **to** $2N$ **do**

    if POINT(i) is marked "left" **then begin**

        /* The point marks the beginning of an interval */

        get a free color γ

        assign γ to the interval that corresponds to POINT(i)

    end

    else begin

        /* The point marks the ending of an interval */

        release the color assigned to the interval of POINT(i)

L_1 : Assign color 1 to interval 1.
L_2 : Assign color 2 to interval 2.
L_3 : Assign color 3 to interval 3.
R_1 : End of interval 1; release color 1.
L_4 : Assign color 1 to interval 4
R_3 : End of interval 3; release color 3.
L_5 : Assign color 3 to interval 5.
R_2 : End of interval 2; release color 2.
L_6 : Assign color 2 to interval 6.
R_4 : End of interval 4; release color 1.
R_6 : End of interval 6; release color 2.
L_7 : Assign color 2 to interval 7.
R_5 : End of interval 5; release color 3.
R_7 : End of interval 8; release color 2.
L_8 : Assign color 2 to interval 8.
R_8 : End of interval 8; release color 2.

Figure 6.6: Execution of Left-edge algorithm

end

The above algorithm is popularly known as "left-edge" algorithm and was invented by Hashimoto and Stevens in the context of PCB routing [HS71]. To illustrate the algorithm for the example of Figure 6.1, the following notation is used. The left end point of interval i is denoted by L_i and the right end point of interval i is denoted by R_i. There are 8 intervals in the example. After the sorting step, the end points are stored in the following order:

$$L_1 L_2 L_3 R_1 L_4 R_3 L_5 R_2 L_6 R_4 R_6 L_7 R_5 R_7 L_8 R_8$$

The second step of the algorithm examines the end points in the above order; Figure 6.6 illustrates the second step for the example. When there is a choice of several colors to paint an interval, the algorithm uses the color that has been *most recently released.* For example, when interval 8 is colored, there is a choice of two colors – 3 and 2. Color 2 is used since it is most recently released. Figure 6.11 shows the result of executing the left-edge algorithm for the above example.

6.2.4 Parallel Algorithm

The left-edge algorithm can be parallelized on a shared memory computer with $2N$ processors. The POINTS array is stored in the shared memory. The processors operate in the EREW PRAM mode. Thus each processor is assumed to have access to every memory location in $O(1)$ time. Further, no two processors attempt to read from or write into the same memory location. Several parallel sorting schemes are available to implement Step 1 of the left-edge algorithm on an EREW PRAM. The parallel merge sort algorithm due to Richard Cole [Col88] uses a linear number of processors to sort N records in $O(\log N)$ time. Thus Cole's algorithm is optimal in the number of processors used.

Now consider how Step 2 of the left-edge algorithm may be parallelized. First, a procedure is described to compute the chromatic number λ of the interval graph. The left-edge algorithm assigns a color if POINTS(i) is a left end point, and releases a color if POINTS(i) is a right end point. This effect can be modeled by a variable X[i], which is set to $+1$ if POINTS(i) is a left end point, or to -1 if POINTS(i) is a right end point. Now consider the partial sum $Y_i = \sum_{j=1}^{i} X_j$. This sum represents the number of colors that have been used until the time of encountering ith end point. $Z_i = max_{j=1}^{i} Y_j$ indicates the maximum number of colors that have been used by the algorithm at any point of time. Thus, Z[$2N$] represents the chromatic number of the interval graph. The algorithm of Figure 6.7 uses the above ideas. The only difference is that Y and Z are not stored in separate arrays – the array X itself is reused. The algorithm requires $O(\log N)$ time. An example run of the algorithm is shown in Figure 6.8.

6.2.5 A coloring algorithm

The procedure *AssignColors* in Figure 6.9 may be used to color an interval graph. Only the processors assigned to the left end points are active during the procedure. After sorting the POINTS array, let the left end points occur in the sequence $1, 2, \cdots, N$. Let X_i denote the color which is assigned to interval i. Without loss of

Step 1.

Assign an individual processor to each end point.

Step 2.

for each processor $i = 1$ to $2N$ **do in parallel**
if my point is a left end point, set $X[i] = 1$;
else , set $X[i] = -1$;

Step 3.

for $j = 1$ **to** $\lceil \log_2 2N \rceil$ **do**
for each processor $i = 1$ to $2N$ do in parallel
if $(i + 1 - 2^j) \geq 1$ **then** $X[i] = X[i] + X[i + 1 - 2^j]$;

Step 4.

for $j = 1$ **to** $\lceil \log_2 2N \rceil$ **do**
for each processor $i = 1$ to $2N$ do in parallel
if $(i + 1 - 2^j) \geq 1$ **then** $X[i] = max(X[i], X[i + 1 - 2^j])$;

Figure 6.7: Parallel computation of Chromatic Number λ of an interval graph

PROCESSOR :	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
INTERVALS :	1	2	3	1	4	3	5	2	6	4	6	7	5	7	8	8
END POINTS:	L	L	L	R	L	R	L	R	L	R	R	L	R	R	L	R
X (Step 2):	1	1	1	-1	1	-1	1	-1	1	-1	-1	1	-1	-1	1	-1
X (Step 3):	1	2	3	2	3	2	3	2	3	2	1	2	1	0	1	0
X (Step 4):	1	2	3	3	3	3	3	3	3	3	1	3	1	1	1	3

Figure 6.8: Computing Chromatic Index, N=8.

generality, interval 1 is assigned color 1 i.e. $X_1 = 1$. For an interval $i > 1$, X_i is determined using the following recursive definition.

1. If the end point of interval $i-1$ is labeled **left**, it implies that i and $i-1$ cannot be assigned the same color. The algorithm "opens up" a new color for interval i. Therefore, $X_i = X_{i-1} + 1$

2. If the end point of interval $i-1$ is labeled **right**, intervals $i-1$ and i have an empty intersection and can be assigned the same color. In other words, $X_i = X_{i-1}$.

The recursive formulation above results in N linear equations of the form

$$X_i = X_{Y_i} + Z_i$$

for $1 \leq i \leq N$. To start with $Y_i = i - 1$ for $2 \leq i \leq N$. Z_i is 1 or 0 depending on cases (1) and (2) above. Step 1 of the procedure *AssignColors* initializes the variables X_i, Y_i, and Z_i as explained above. It uses a boolean variable V_i which is set to 1 if and only if the final value of X_i has been computed.

Step 2 of the procedure solves for $X_2, X_3, \cdots, X_N$. Advantage is taken of the fact that each of these equations has no more than two variables. The coefficients of the X variables in all these equations are unity. A doubling technique is therefore used to solve for $X_2, \cdots, X_N$ in $\log_2 N$ steps. Figure 6.10 shows the execution results of *AssignColors* for the example of Figure 6.8. Figure 6.11 shows the interval graph after the nodes have been painted.

6.2.6 Interval Graph Coloring on iPSC/2

The parallel coloring algorithm was implemented on a 32-node Intel iPSC/2. The preliminary results of the implementation are presented in this section. The machine architecture of iPSC/2 is discussion in Appendix A. The input data to the parallel algorithm is stored in a file which resides on the concurrent file system. Nodes read input data from this file concurrently. Input data consists of $2N$ records, one record

```
procedure AssignColors;
begin
Step 1.
        for each processor i = 1 to 2N do in parallel
            if my end point is marked L then begin
              if i=1 then set X[i] = 1, V[i] = 1.
              else begin
                              V[i] = 0
                              if POINTS(i − 1) is marked L then
                                set Y[i] = i − 1, Z[i] = 1.
                              else set Y[i] = i − 1, Z[i] = 0.
              end
            end
Step 2.
        for j = 1 to ⌈log 2N⌉ do
            for each processor i = 1 to 2N do in parallel
                if V[i] = 0 and my end point is L then begin
                  if V[Y[i]] = 1 then
                  begin
                    Y[i] = Y[Y[i]]
                    X[i] = Y[i] + Z[i]
                    V[i] = 1
                  end else
                  begin
                        Y[i] = Y[Y[i]]
                        Z[i] = Z[i] + Z[Y[i]]
                        V[i] = 0
                  end
                end
end procedure
```

Figure 6.9: Assigning colors to intervals

Figure 6.10: Example of Interval Assignment; $N = 8$

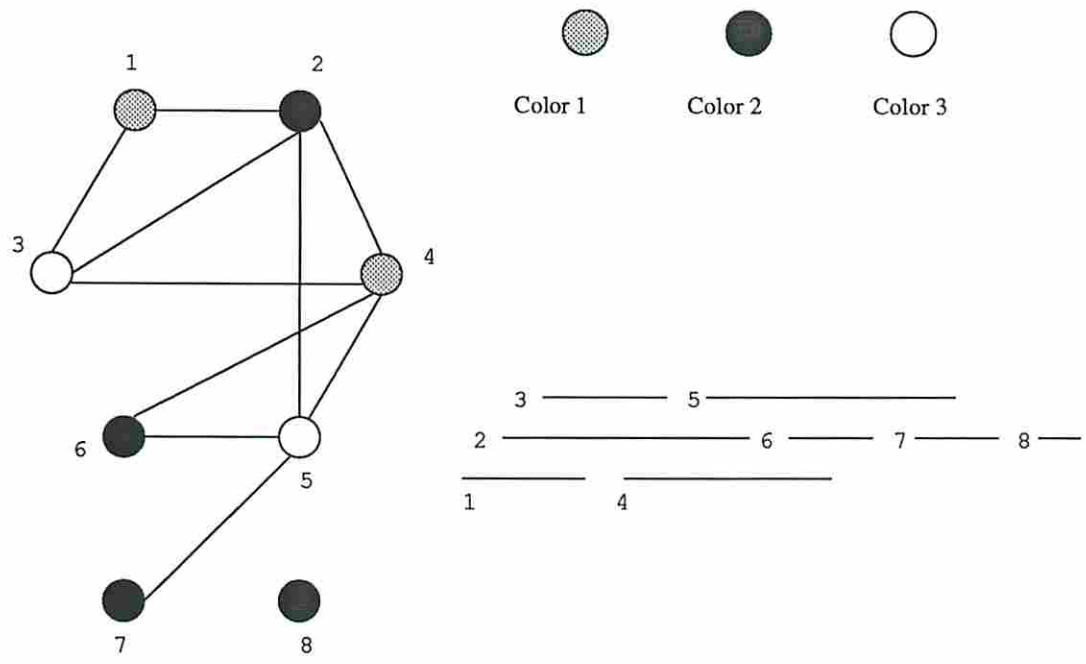

Figure 6.11: After coloring the intervals, the interval graph may be colored

for each of the N left and right end points. An input record consists of three fields – the name of the interval, abscissa of the end point and its *type* (left/right). Each node reads $2N/P$ end points; for convenience, it is assumed that $2N/P$ is a round number.

A simple sorting algorithm based on Odd-Even Merge [L+84] was implemented to sort the end points based on their abscissa values. A P processor linear array is simulated on the hypercube for this purpose. The following technique is used to embed a linear array of size P into a hypercube of P nodes. Let $g(i)$ indicate the successor to number i in the sequence of binary gray codes. For example, $g(0) = 1, g(1) = 3, g(3) = 2$, etc. It is easy to see that the processors $0, g(0), g^2(0), \cdots, g^{P-1}(0)$ form a linear array. The sorting algorithm consists of two steps. In the first step, each processor sequentially sorts the $2N/P$ endpoints available locally. At the end of the first step, we have P sorted lists distributed across P processors. These are merged into one sorted list during the second step, which consists of P iterations of the merge procedure. Each iteration is classified as an *odd*

step or *even* step – during the *odd* step, the odd-numbered processors in the linear array send their sorted lists to the even-numbered processors. An even-numbered processor merges its local list with the one received, hence creating a compound sorted list of length $4N/P$. It retains the top $2N/P$ elements as its local list and sends back the bottom $2N/P$ elements to its right neighbor. During the *even* step, the roles of the odd-numbered and the even-numbered processors are reversed.

The time required to execute the parallel algorithm consists of three components. If quick sort is employed for the local sorting step, it requires $O(\frac{2N}{P}\log(\frac{2N}{P}))$ time. The communication time for each iteration of odd-even merge is $O(\frac{N}{P})$; therefore the total communication time for the algorithm is $O(N)$. The computation during each iteration of odd-even merge takes $O(\frac{4N}{P})$ time, since it involves merging two lists with $\frac{2N}{P}$ elements each. Thus, the total amount of time spent in merge is also $O(N)$. The time requirement of the parallel algorithm is therefore $O(\frac{2N}{P}\log(\frac{2N}{P})) + O(N)$. A sequential algorithm would require $O(2N\log(2N))$ time. The speedup for large values of N is P. For smaller values of N, the inter-processor communication time causes speedup degradation.

After the sorting step is completed, the nodes compute the chromatic number and carry out color assignment. Each node scans the $\frac{2N}{P}$ end points and computes $X(i)$ for each end point i, $1 \leq i \leq \frac{2N}{P}$. $X(i)$ is set to $+1$ if i is a left end point and to -1 otherwise. The chromatic number λ is given by $max_i \sum_{j=1}^{i} X(j)$; the processors compute λ in the following manner.

1. Each node processor sequentially computes $\sum_{j=1}^{i} X(j)$ for $1 \leq i \leq \frac{2N}{P}$. Let $X_j(\frac{2N}{P})$ indicate the value $X(\frac{2N}{P})$ in processor j.

2. The processors compute the partial sums $\sum_{j=1}^{P} X_j(\frac{2N}{P})$. The binary tree computation [Akl89] is used for this purpose.

3. Processor j adds the offset value of $X_{j-1}(\frac{2N}{P})$ to each element $X_j(i)$, $1 \leq i \leq \frac{2N}{P}$.

The coloring algorithm is a straight-forward implementation of the procedure shown in Figure 6.9. The computation of chromatic number and the assignment of colors

both require $\log P$ iterations; each iteration requires $O(\frac{2N}{P})$ time. When compared to a sequential algorithm, the speedup obtained during either step is $\frac{2N}{(\log P \frac{2N}{P})}$ or $\frac{P}{\log P}$.

Figures 6.12 and 6.13 show the empirical results obtained by running the parallel algorithm on the Intel iPSC/2. The program was implemented in C and required about 500 lines of code excluding comments. The **mclock** function was used to obtain an estimate of the execution time of each processor. The test data was generated by a program. The execution time includes both computation time as well as I/O time. Figure 6.12(a) plots the execution time of the parallel coloring algorithm as a function of $\log_2 N$, using P as a parameter. Figure 6.12(b) plots the execution time as a function of P, using N as a parameter. For a fixed value of N, we observe that as P is increased, the execution time decreases, reaches a bottom value, and then increases again. The decreasing portion of the curve is where parallel processing is paying off. The increase in execution time for larger P is due to the increase in interprocessor communication overheads.

Figures 6.13(a) plots the speedup of the parallel coloring algorithm as a function of $\log_2 N$, using P as a parameter. We observed that the speedup increases up to a threshold value of N and then decreases again as N is increased. After hitting a minimum, the speedup increases for larger values of N. This again is due to the interplay between computation time and communication time. For very large values of N, the computation inside a node exceeds the communication time, hence resulting in an increased speedup. When computation time is negligible compared to communication time, the speedup can be lower than unity. A similar effect is again observed in Figure 6.13(b), which plots the speedup of the algorithm as a function of P using N as a parameter. For a fixed problem size, increases linearly as P is increased up to a threshold value of P. The speedup drops off for larger P due to communication overhead. Odd-even merge sort is communication-intensive for large N and P; we are presently improving the performance of the sorting step by using a more efficient sorting algorithm called bitonic sort.

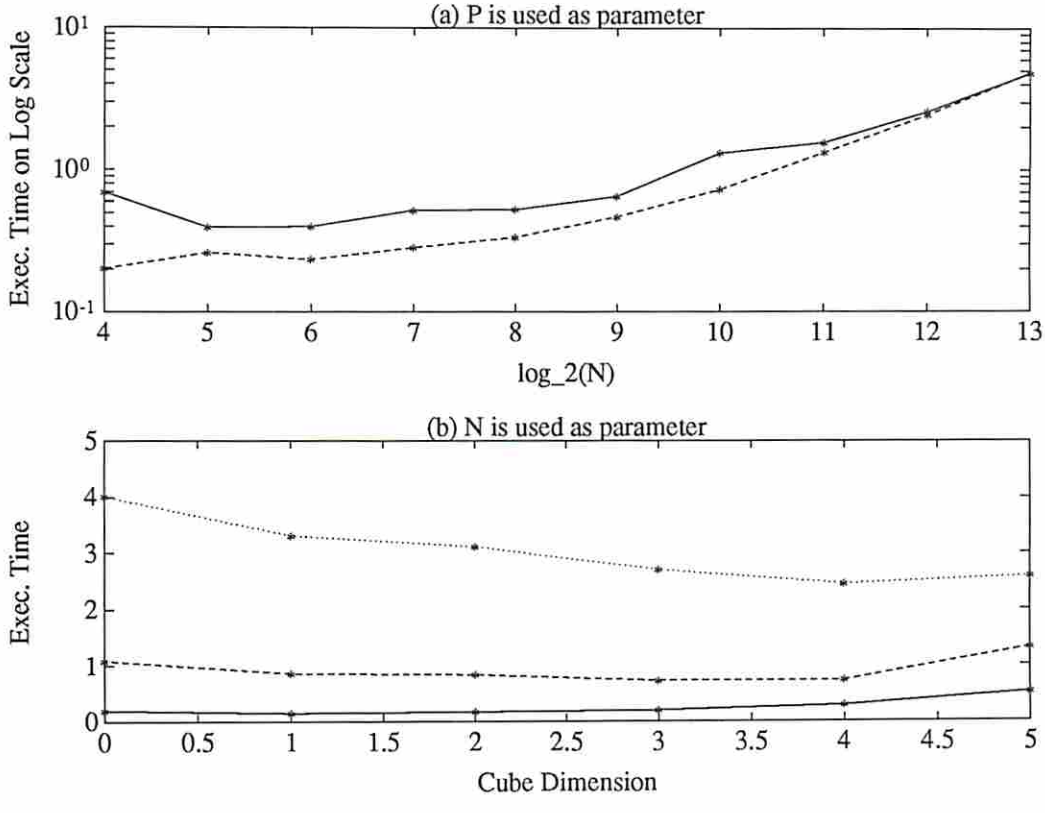

Figure 6.12: (a) Execution Time of Coloring Algorithm on iPSC/2, as a function of $\log_2 N$. The solid and dashed curves correspond to $P = 16$ and $P = 32$. (b) Execution time as a function of cube dimension d. The solid, dashed and dotted curves correspond to $N = 128, 1024, 4096$.

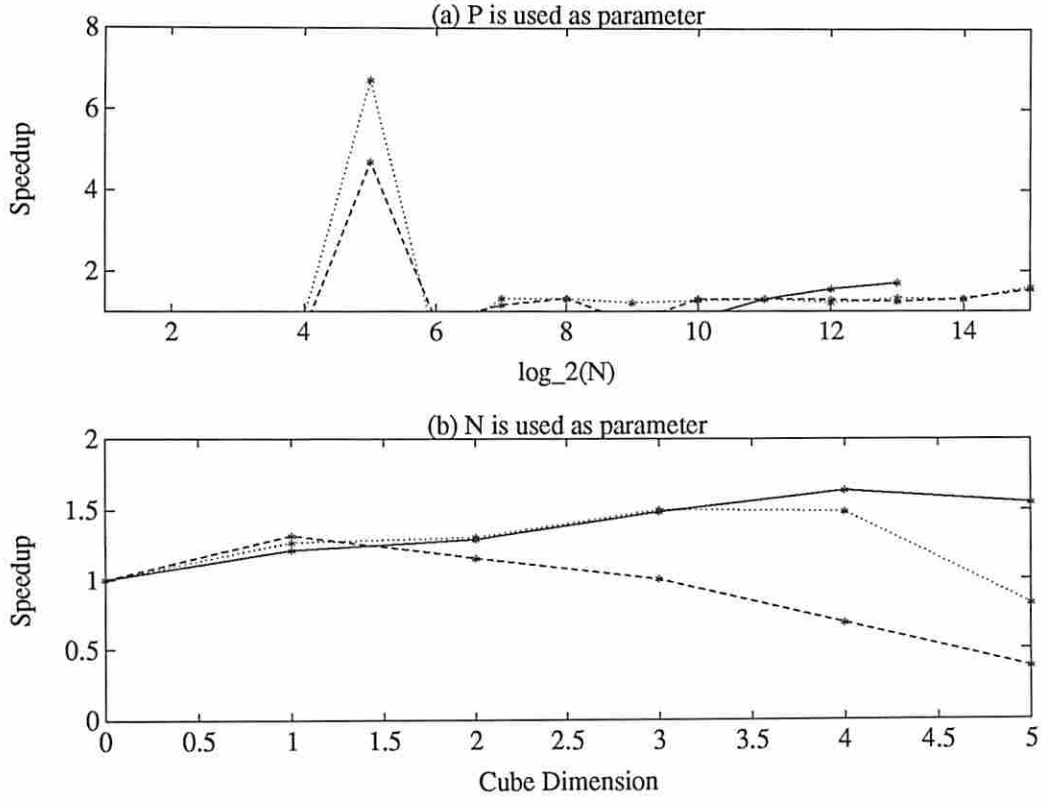

Figure 6.13: (a) Speedup of the coloring algorithm on iPSC/2, as a function of $\log_2 N$. The solid, dashed and dotted curves correspond to $P = 32, 4, 2$ respectively. (b) Speedup as a function of P. The solid, dotted and dashed curves correspond respectively to $N = 4096, 1024, 128$.

6.3 Coloring Permutation Graphs

6.3.1 Applications

Permutation graphs are useful in problems related to via minimization. The permutation diagram of Figure 6.2 may be considered as a rectangular channel in a circuit layout. The cross lines are two-point nets, each of which connects a pin at the top of the channel to a pin at the bottom. If two nets cross each other, they must be placed on two different layers in order to avoid a *via*. An interesting problem is to find the minimum number of layers λ required to route all the nets without *vias*. Given λ layers, how should nets be assigned to layers so that minimum-via routing is achieved? This problem is modeled by creating the following graph. A node is created for every net in the channel. Nodes i and j are connected if (and only if) nets i and j cross each other in the permutation diagram. It is clear that the graph is a permutation graph. Consider an optimum coloring of the graph. The colors assigned to nodes i and j can be interpreted as layers. Two nodes i and j are assigned different colors, say γ_i and γ_j, if they share an edge. Thus if nets i and j are assigned to layers γ_i and γ_j, there will be no overlap. The number of layers required to for zero-via routing is the chromatic number of the permutation graph. In the example of Figure 6.2, the graph can be optimally colored using 3 colors. Nodes 4 and 6 are assigned color 1; nodes 3 and 5 are assigned color 2. Finally, nodes 1 and 2 are assigned color 3. The corresponding layer assignment is as follows. Nets 4 and 6 are assigned to layer 1; note that these nets do not intersect and thus can be assigned to the same layer. Nets 3 and 5 are assigned to layer 2. Nets 1 and 2 are assigned to layer 3.

6.3.2 Movable Terminals

Consider the following version of the two-layer channel routing problem. Suppose that the objective of the router is to minimize the number of *vias*. Further suppose that the terminals on the channel are *movable* i.e. as long as the relative positions of the pins are maintained, it is permissible to move the pins to different positions.

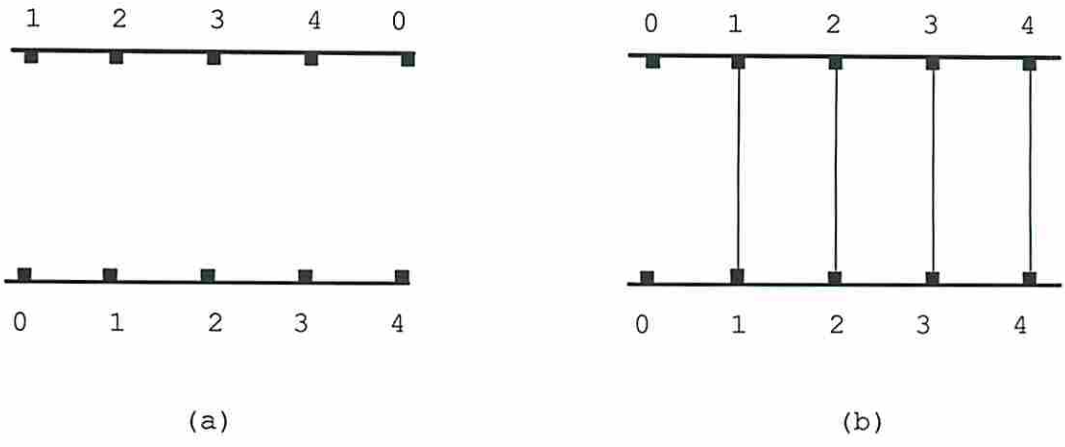

Figure 6.14: Channel Routing with Movable Terminals

The concept of movable terminals can be understood with the following example. Figure 6.14(a) shows a channel with 5 columns. There are four nets in the example. Pins marked 0 are not connected to any net. The order of pins on the top of the channel is 1,2,3,4. The order of pins on the bottom of the channel is also 1,2,3,4. With this pin assignment, the channel has a density of 2. Under the Manhattan routing model, at least 2 tracks are necessary to route the channel. However, due to vertical constraints, four tracks are necessary to route the channel. This was illustrated in Figure 1.7 (page 25). If the pins are movable, the channel can be simplified substantially, as shown in Figure 6.14(b). Note that the order of the pins has not been altered in reassigning the pins. The density of the modified channel is 0 – all the nets can be implemented as "abutments" (straight connections). No vias are necessary to route the modified channel.

Under the assumption that pins are movable, the via minimization problem is an easy optimization problem. Deogun and Bhattacharya have shown that the above problem reduces to one of finding the maximum independent set in a permutation graph [DB89]. For simplicity, let all the nets be two-point nets. As seen earlier, it pays to move the pins of the channel so as to create abutments. To be more precise, the number of vias reduces by 2 for every net realized as an abutment. In the example of Figure 6.14, it was possible to realize all the nets as abutments by a reassignment of pins. This is not possible in general; the best one can do is to maximize the number

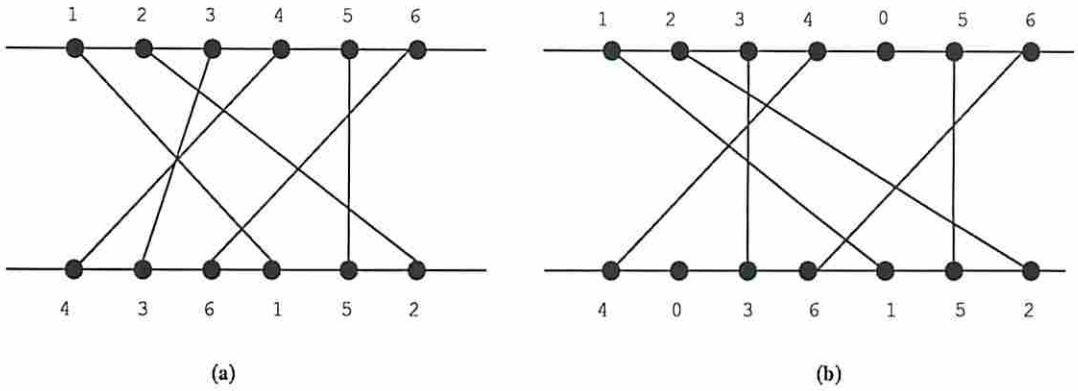

Figure 6.15: Maximizing the number of abutments by reassigning pins.

of abutments through the process of pin reassignment. Which nets should be selected for abutment? A moment's reflection shows that the nets that are parallel to each other in the permutation diagram are good candidates for abutment. This is because all of them can be simultaneously "straightened out." Therefore, the objective of pin reassignment is to pick the maximum sized set of mutually parallel nets in the permutation diagram. By the construction of the permutation graph, if two nets are parallel in the permutation diagram, the corresponding nodes in the graph will not share an edge. Hence, the problem of pin reassignment boils down to one of selecting the maximum independent set in the permutation graph. To illustrate, consider the channel shown in Figure 6.15(a). This is the same channel as in Figure 6.2 which was used earlier to explain the concept of permutation graphs. Since nets 3 and 5 are parallel, they can be simultaneously implemented as abutments by introducing two dummy terminals, as shown in Figure 6.15(b). Alternately, nets 4 and 6 can be selected for this purpose, or nets 1 and 2. The size of the maximum independent set in the corresponding permutation graph (shown in Figure 6.2) is 2. Therefore, at most 2 abutments can be realized in this example.

6.3.3 Algorithm for Coloring Permutation Graphs

A sequential algorithm for coloring permutation graphs is described by Golumbic [Gol80]. It requires $O(N \log N)$ time to obtain an optimum coloring of an N-node

permutation graph. The input to the algorithm is the permutation representation π for the graph G. The numbers in the permutation are processed from left to right, inserting them in different queues. All nodes which can be given the same color are inserted into the same queue. λ queues will be created at the end of the algorithm. When inserting the ith entry, the algorithm tries to be greedy and checks if an existing queue can be used. There can be at most $i-1$ queues before π_i is inserted. π_i can go into any one of these queues, provided the last element in the queue is smaller than π_i. The procedure inserts π_i into the first queue which satisfies this property. If there is no such queue, a new queue is created and π_i is made the first entry in the new queue. Binary search can be used to locate the queue in which π_i needs to be inserted. Thus, it takes $O(\log i)$ time to insert element i, $1 \leq i \leq N$. The complexity of the procedure is therefore $O(N \log N)$.

Figure 6.2 illustrates the technique. The permutation representation for the example is 436152. The node 4 is examined first and is inserted into queue 1. Next comes node 3. Since $3 < 4$, node 3 shares an edge with node 4 in the graph; therefore, a second queue is created and 3 is inserted into queue 2. Node 6 can be inserted into queue 1, since $6 > 4$. Node 1, which is the next in the sequence, cannot go into queue 1 or queue 2. Thus node 1 is inserted into a newly created queue 3. Node 5 can go into queue 2. Finally, node 2 can be inserted into queue 3. The chromatic number of the graph is 3 and the following is an optimum coloring of the graph. Color 1 to nodes 4,6; color 2 to nodes 3,5 and color 3 to nodes 1,2.

A parallel algorithm can be designed for coloring permutation graphs. The algorithm uses a linear array of N processors. Each processor has a local memory consisting of $N+1$ registers, each $\log_2 N$ bits long, which form a queue. It is sufficient if the processor has capability to compare to $\log_2 N$ bit numbers and has built-in logic for queue insertion algorithm. The processors operate in a pipelined fashion. The permutation representation is fed as input to processor 1 in the sequence $\pi_1 \cdots \pi_N$. Before the algorithm begins execution, register R_0 in each processor is initialized to 0. Each processor executes the same code, shown in Figure 6.16. At the end of the

```
for each processor 1, 2, ···, N do in parallel
    do N times
        begin
            receive element l from left processor;
            if l > R_0 then begin
              insert l into local queue;
              send 0 to right processor;
            end else
                send l to right processor;
        end
```

Figure 6.16: Pipelined Algorithm for Permutation Coloring

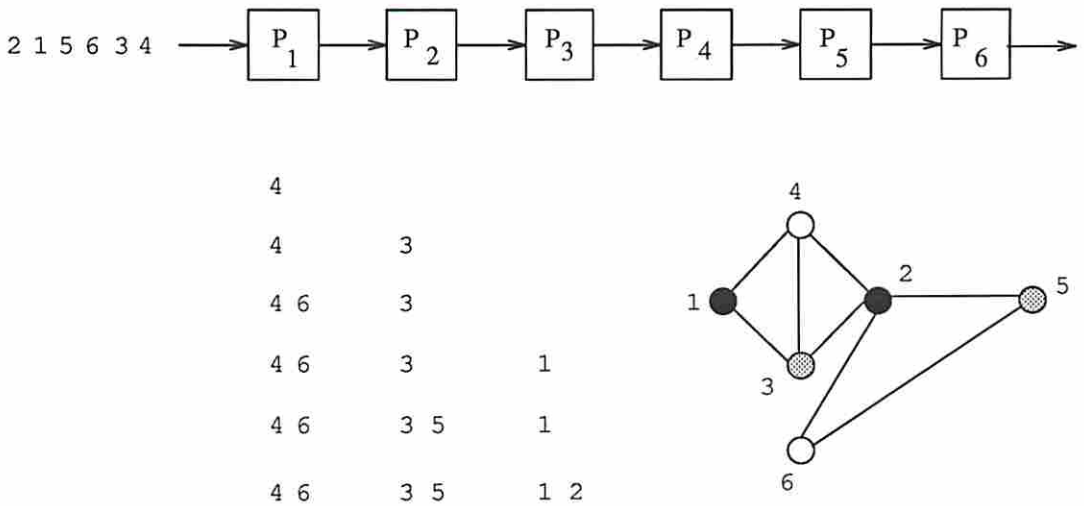

Figure 6.17: Example of Permutation Graph Coloring

algorithm, all the elements in the local queue of processor i have the color i. An example is shown in Figure 6.17.

The parallel algorithm for coloring permutation graphs requires $O(N)$ time since the pipeline takes $2N - 1$ steps to flush out all the inputs. Compared to the $O(N \log N)$ serial algorithm, the speedup obtained is $\log N$. The processor-time product for the parallel algorithm is $O(N^2)$. Thus the parallel algorithm is not optimal. This is because the serial algorithm uses an intelligent binary search to find the queue for inserting element π_i. The parallel algorithm, however, searches through all the possible $i-1$ queues before inserting element π_i. This requires $O(N^2)$

work and $O(N)$ parallel time. Can an optimal parallel algorithm be designed for coloring permutation graphs? Such an algorithm uses N processors and $O(\log N)$ execution time. Designing such an algorithm (or proving that no such algorithm can be designed) is an open question.

6.3.4 Maximum Independent Set

The algorithm for coloring a permutation graph G can also be used to identify a maximum independent set in G. In fact, it can also be used to determine a maximum clique in G. To see how, consider the permutation representation π of the graph G. Recall that there is an edge between nodes i and j in the graph G if i and j do not appear in their natural sequence in π. For example, consider the permutation 436152 and its permutation graph (Figure 6.2). Nodes 4 and 3 share an edge since they appear in the decreasing order in the permutation. If there is a decreasing sequence $i_1 > i_2 > \cdots > i_k$ in the permutation π, the nodes $i_1, i_2, \cdots i_k$ form a clique in the permutation graph G. In our example, nodes 4,3, and 1 form a clique since the decreasing sequence 431 appears in the permutation 436152. Similarly, nodes 6, 5 and 2 form a clique. Therefore, it can be concluded that a largest decreasing sequence in π corresponds to a largest clique in G. Nodes 4,3 and 1 indeed form a maximum clique in the example.

To find a maximum independent set of G, the following fact is used. A maximum independent set in a graph G corresponds to a maximum clique in the complement of G. The permutation representation for the complement of G is simply the reversal of π. In the example above, 251634 is the permutation representation of the complement permutation graph. A longest decreasing sequence in 251634 is 53. Thus nodes 5 and 3 form a maximum independent set in the original permutation graph.

The coloring algorithm of the previous section can be used to identify a longest decreasing sequence in π. Let λ be the chromatic number of G. As seen earlier, the coloring algorithm opens up λ queues. Recall that the algorithm starts a new queue if (and only if) it cannot insert a number π_j into one of the existing queues. Suppose that π_j gets inserted into the ith queue i.e. the color assigned to node π_j is i. Then

the last entry in the $i-1$st queue must be larger that π_j - otherwise π_j would have been inserted into the $i-1$st queue. Thus if we remember the last entry of $i-1$st queue while inserting π_j into the ith queue at the end of iteration j, we can detect the longest subsequence in π. In the running example, $\pi = 436152$. The number 3 cannot be inserted into the first queue and the algorithm is forced to open a second queue ($i = 2$). Thus we remember 4 which caused this phenomenon. The next time this occurs is when trying to insert 1. The algorithm begins the third queue ($i = 3$) since it cannot insert it into the first or second queues. We remember number 3, the last entry on the second queue. There are no more occurrences of opening a new queue. Thus 4,3 and 1 form a decreasing subsequence in the permutation 436152. It is straight forward to extend the parallel algorithm of Figure 6.16 to compute the maximum independent set in the permutation graph.

6.4 Summary

Although layout optimization problems are NP-complete in general, special cases which result from making assumptions on the routing model may be solvable in polynomial-time. These special cases can be modeled as optimization problems on perfect graphs. This chapter considered several such specialized problems related to routing and high level synthesis, and formulated them as coloring problems on interval graphs and permutation graphs. Parallel algorithms were presented for both these coloring problems. Several other applications of these problems, such as resource allocation and task scheduling, were pointed out.

Future research in this area can proceed by considering specialized layout problems such as river routing, over-the-cell routing, and segmented channel routing. It is extremely useful to be able to model these problems as optimization problems on perfect graphs. Parallel algorithms for these problems are also of great interest.

Chapter 7

Conclusions and Further Research

Historically, interest in applying parallel processing to electronic design automation dates back to early 1980's, when special-purpose architectures were designed for PCB routing based on the Lee-Moore algorithm. Since then, both special-purpose and general-purpose parallel computing have been actively pursued in the development of faster layout algorithms. However, due to considerations of price/performance ratio, the emphasis appears to be shifting towards mapping algorithms onto general-purpose architectures. Much work remains to be done in this area, and this chapter will summarize some of the important research topics to be pursued. A *parallel framework* needs to be identified and developed for computer-aided VLSI design; Figure 7.1 illustrates the idea of such a framework. This chapter will discuss the issues pointed out in Figure 7.1. Open research problems related to these issues are pointed out.

7.1 A Parallel Framework

7.1.1 Core Computations

The first important task is to identify "stable algorithms" for computer-aided design and "core" computations of these stable algorithms. The field of computer-aided-design is highly dynamic; new design styles, innovative data structures, new

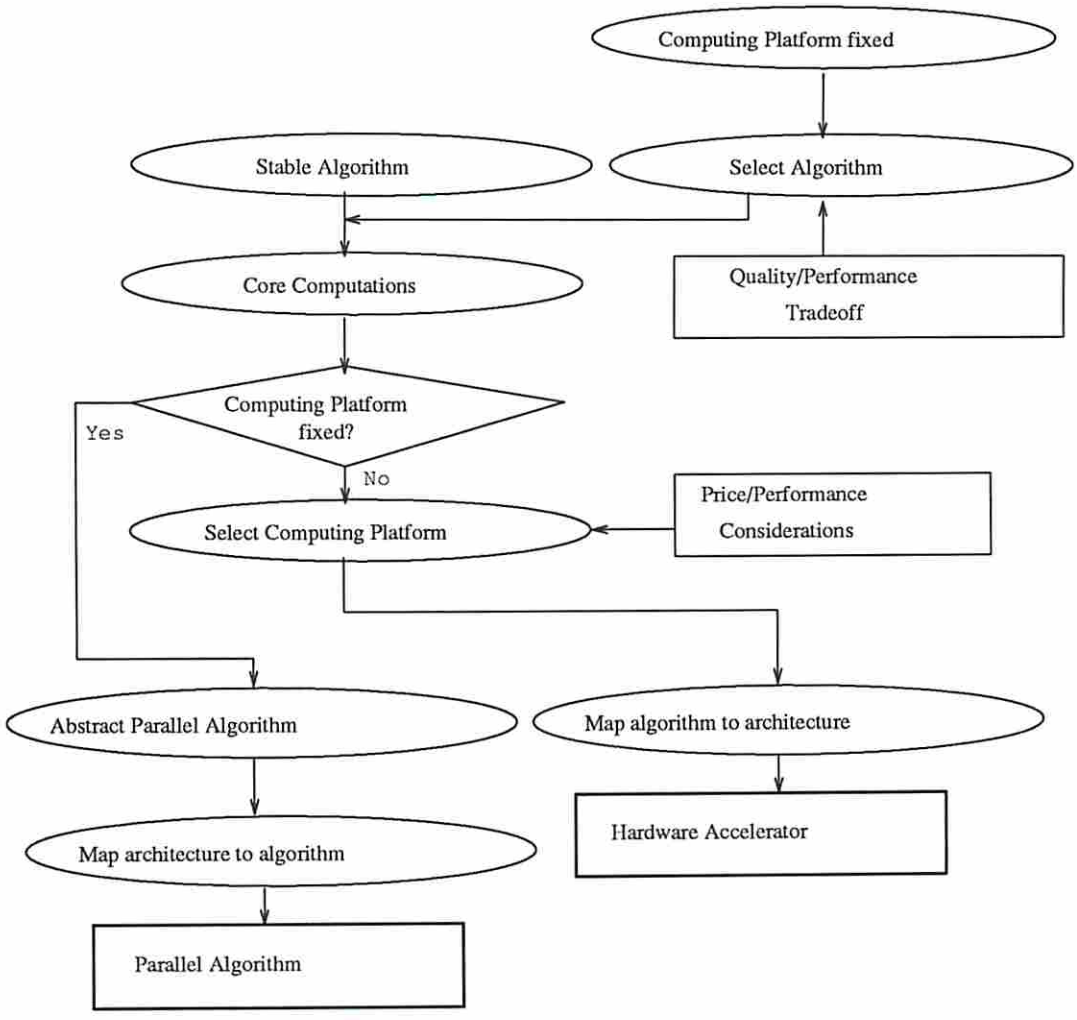

Figure 7.1: A Parallel Framework for Design Automation.

objective functions, new computing paradigms are continually introduced. As a result, CAD programs need to be updated quite often. As an example, consider the circuit placement problem. The problem has been approached by a variety of methods – constructive placement, iterative improvement, min-cut, simulated annealing, divide-and-conquer, genetic algorithms and many others. The objective of the placement problem can be different in different placement algorithms. Frequently used objectives are (a) to minimize total chip area, (b) to minimize total wire length, (c) to maximize routability, (d) to minimize the length of the longest wire, and so on. There are several design styles such as gate arrays, standard cells, general cells, sea-of-gates and field programmable gate arrays; the placement problem varies with design styles. Similarly, circuit routing has undergone several changes. Early automatic routers were developed for PCB routing and predominantly used Lee routing. Lee routing is inadequate for routing large integrated circuits; hence hierarchical routing has evolved. There are several approaches to solve global routing – integer programming, cut-and-paste, simulated annealing and expert systems to name a few. There are several objective functions to be considered during global routing – routability, total wire length, and length of the longest wire. Channel routing has also undergone several changes. Objective functions have ranged over channel width, number of *vias*, number of layers, and so on. A *stable* algorithm is one which can be stand through changing design styles and objective functions. In the long run, it only pays to build hardware accelerators for stable algorithms. Lee's algorithm for routing can be stated as an example of a stable algorithm. It is a flexible algorithm which can be used for multiple routing models involving several design styles (see Ravikumar and Sastry [RS90]). There are not many stable algorithms for computer-aided-design. It is important to consider this factor when comparing hardware accelerators and general-purpose parallel computers.

The *core* of an algorithm is the most time consuming part of the algorithm. For example, the inner most loop in a sequence of nested loops is a core computation. By speeding up a core computation, the performance of an algorithm can be dramatically improved. Let us consider some examples of core computations in physical

design. An iterative improvement algorithm spends the majority of its execution time making local changes to the placement and evaluating the resulting improvement in cost function. Therefore, considerable gain in performance is achieved by executing this core step in parallel. In Chapter 4, several techniques have been examined towards this end. A particular technique can be selected for implementation based on price/performance considerations.

7.1.2 Analysis of Parallelism

Having identified the core computations, it is necessary to analyze the parallelism in these computations. A distinction must be made between parallelism in a CAD problem and parallelism in an algorithm to solve the CAD problem. Parallelism in a problem is more difficult to characterize than parallelism in an algorithm. This difficulty is increased if the problem is known to be NP-complete. In Chapter 2, we indicated the analysis of parallelism in a placement problem. Similar studies must be conducted for other physical design problems.

Analyzing a particular algorithm for parallelism is of pragmatic interest. When a point accelerator is to be built around the algorithm, understanding the parallelism in its core computation helps select the best architecture for the algorithm. On the other hand, if the target machine is a general-purpose supercomputer, the analysis helps in matching the algorithmic structure to the architecture of the supercomputer in the best possible way.

7.1.3 Abstract Parallel Algorithms

After analyzing a CAD algorithm for parallelism, it is instructional to develop a parallel version of the algorithm for an abstract machine model such as the PRAM. PRAM algorithms are of both theoretical and practical interest. On the theoretical side, they represent a systematic way of designing a parallel version of a sequential algorithm. A PRAM algorithm does not make assumptions about the machine architecture – whether the machine has 6 processors or 1024, what microprocessor

is used, what memory technology is employed or what the clock rate is. It therefore becomes possible to reason about the parallelism in the given sequential algorithm. It also becomes possible to express the parallel algorithm in a language that is universally understandable. Since there are systematic methods to map a PRAM algorithm to a real machine architecture, PRAM algorithms are also of practical interest as well. In Chapter 3, a PRAM algorithm was described for circuit partition. We considered PRAM algorithms for several graph optimization problems in Chapter 6. It is a fruitful area of research to develop PRAM algorithms for other layout applications discussed in this dissertation.

7.1.4 Computing Platform

The next step is to decide the computing platform for the parallel implementation of the CAD algorithm. Price/performance considerations and practical considerations guide the selection of the target machine. For example, the user may already have access to a specific supercomputer. Or it may be necessary to build a special-purpose machine for the algorithm. In either situation, there are some tradeoffs to be considered. In the former situation, the user has the responsibility of selecting the best algorithm for the target computer. The user must also consider alternate ways to map the algorithm onto the supercomputer architecture and settle for the best. In the latter situation, price/performance tradeoffs dictate the selection of the target architecture. An architecture that exploits all of the parallelism in the algorithm is also likely to be the most expensive choice. For example, a mesh-of-processors architecture with one processor per grid element is ideally suited for Lee's routing algorithm. This amounts to N^2 processors for $N \times N$ grid. On the other end of the spectrum, one can consider a pipelined machine for Lee routing as a relatively inexpensive alternative. The pipelined architecture cannot offer the same performance as the array architecture, but is more affordable. If there is a choice between a supercomputer and a special-purpose machine, there are tradeoff factors such as price and programmability to be examined.

7.1.5 Implementation

7.1.5.1 Parallel Algorithms

Unlike abstract machines such as PRAM, real machines have a finite number of processors, a finite amount of memory and impose memory access restrictions. As a result, it may not be possible to take advantage of all the parallelism in the algorithm. At best, a speedup of P is obtainable through a deterministic parallel algorithm, P being the number of processors.

Coding a parallel algorithm is no small task, since debugging facilities and program development tools are limited on parallel machines. The programmer must be well aware of the architecture of the supercomputer to be able to deliver an efficient implementation. Three major decisions that affect the efficiency of the algorithm are data storage schemes, task partition, and synchronization. In a shared memory computer [HB85], the global memory is not *uniformly accessible* to all the processors. In other words, memory access time is different for different memory units. Therefore data storage schemes will affect the average latency of memory access. In a distributed memory machine, it is important to decide which variables are stored in the local memory of each processor. Some times, some amount of duplication can result in a better implementation. For example, storing the entire connectivity matrix in each node can avoid interprocessor communication that is necessary to fetch connectivity values. But this scheme may be infeasible for large circuits, in which case the programmer may have to design alternate data structures that trade space with time. Task partition also affects the interprocessor communication. For example, consider an application which involves a tree-like computation such as Min-cut placement. When the target architecture has a different interconnection topology, say two-dimensional mesh, the tree computation must be *embedded* into the mesh. It is desirable that two processes that communicate be mapped to two neighbor processors so that the overhead of communication is reduced. The embedding problem is equivalent to the circuit placement problem and is therefore hard to solve. The programmer is therefore forced to use heuristic techniques for task embedding. The overhead of *synchronization* can be detrimental to a parallel algorithm if a large

number of processors must be synchronized in short bursts of time. So much so, that this problem has prompted researchers to design radically different algorithms that do not involve synchronization; such algorithms are called asynchronous parallel algorithms. An example of an asynchronous parallel algorithm is the MPMD parallel algorithm for iterative improvement (Chapter 4, page 100). The difficulty with asynchronous algorithms is that they may fail to converge on some problem instances.

In addition to making the above decisions, a programmer may face other difficulties in a parallel computing environment. Debugging and evaluating the performance of the parallel program can be challenging due to lack of software tools. The programmer can also obtain inconsistent results i.e. executing the same algorithm with the same input may result in slightly different outputs. This is to be expected when using a multiprocessor operating system; different runs of the same program can have different task scheduling histories. Further, even in a homogeneous multiprocessor, two different processors cannot be expected to work at the same speed. As another source of inconsistency, consider the summation of n real numbers $a_1, a_2, \cdots, a_n$. A sequential algorithm always adds the numbers in the same order. The same cannot be said about a parallel algorithm. In other words, the parallel algorithm may compute $a_1 + a_2 + \cdots + a_n$ in several different ways. If we remember that computer addition is not associative, it is no more surprising that inconsistent results are generated on two different runs of the same algorithm.

Chapters 3, 4 and 5 considered several layout algorithms that have been implemented on actual machines. A good source of parallel programming techniques for several supercomputers is available in [II87] and [Ost89].

7.1.5.2 Hardware Accelerators

Designing and implementing a hardware accelerator is the domain of an expert engineer. Selecting the components to build the accelerator, deciding on the interface between the accelerator and the host machine, incorporating modularity and scalability in the design, all these issues call for expertise and experience. Managerial

expertise is also involved in deciding the final form of the product. Whether the product should interface to a specific engineering workstation or a class of workstations and other computing platforms, whether the product should be designed as an add-on board or as a box attachment, what data formats must be used by the input and output of the accelerator, these decisions are based on present market trends and future projections.

7.1.6 Performance Evaluation

Performance evaluation of parallel computers is an entire discipline by itself. The main performance metrics used in Chapters 3, 4 and 5, are the speedup ratio and processor utilization. The definition of speedup requires that the parallel computing time be compared with that of the "best sequential implementation." The latter is not a well defined concept if we are talking numbers. For example, the same serial algorithm will run faster if ported to a faster uniprocessor of tomorrow. To overcome this problem, many workers compute the serial time by running the program sequentially on one of the nodes in the parallel machine. This method is applicable only to MIMD systems. The scheme has limitations even with multiprocessors, since it may not be possible to run the program on a single node due to limited amount of local memory per node.

Performance evaluation a critical issue because it can help us determine if the estimated performance is indeed delivered by the final product. If not, it is necessary to explore the cause for the discrepancy and apply the necessary correction. Performance evaluation also provides useful feedback about fundamental design decisions made in the implementation. This feedback can be used to improve future designs.

7.2 Matching Architectures and Algorithms

Numerous parallel machines have now become available in the commercial market and more are announced every year. These machines are based on radically different processor architectures, memory architectures, interconnection networks. In the

early days of parallel computing, Flynn classified parallel computers based on the number of instruction streams and number of data streams [HB85]. SIMD computers use synchronous operation and exploit data parallelism. MIMD computers operate asynchronously and exploit both control parallelism and data parallelism. MISD computers (pipelines) exploit temporal parallelism and use synchronous operation. Flynn's classification is not adequate to classify today's parallel computers, which vary widely in the degree of parallelism, granularity of processors and processor-memory interface. The *degree of parallelism* refers to the number of processing elements in a parallel computer. Massively parallel computers have thousands of processors. The Connection Machine, which has 65536 processors, and NCUBE/ten, which has 1024 processors, can be stated as examples. There are also machines with less than ten processors; the Alliant FX/80 allows up to 8 computing elements. The *granularity of a processor* refers to its hardware capabilities. In the Connection Machine, each processor is only capable of bit-serial arithmetic. Such a processor is an example of a *low-granularity* processing element. The computing element of an Alliant FX/80 is an example of a *high-granularity* processing element; it can handle 64-bit arithmetic, the entire instruction set of Motorola 68020 processor and vector instructions. *Processor-memory interface* refers to the way in which processors and memory are interconnected; shared memory computers allow all processors to access a common bank of memory. In a distributed memory computer, each processor has its own private memory bank. Based on the characteristics mentioned above, one can imagine a multi-dimensional space of parallel computers. This section takes the viewpoint of a user who has a design application at hand and must consider various options before selecting a computing platform for the application. Price/performance considerations guide this selection.

7.2.1 Classifying Physical Design Algorithms

We now classify physical design algorithms based on their structure. This classification is helpful in matching a design algorithm to an appropriate architecture.

7.2.1.1 Tree structured algorithms

A large number of layout algorithms are tree structured. The Min-cut placement algorithm, branch-and-bound algorithms, and the cut-and-paste algorithm for global routing are examples of tree-structured algorithms. The min-cut algorithm and the cut-and-paste algorithm are both based on the divide-and-conquer paradigm. A divide-and-conquer procedure exhibits a natural tree structure; it works by dividing the area of a large VLSI chip into two regions using a cut-line. The subregions are recursively bisected, until *atomic regions* are formed. An atomic region is one whose size is small enough to be handled by a simple, non-hierarchical procedure. The division process corresponds to a *binary tree* with the atomic regions as leaves. The above procedure can be generalized to generate a k-ary tree rather than a binary tree; at each step, a region is subdivided into k subregions using $k-1$ cut-lines. When $k=4$, the resulting tree is called a quad-tree. After each atomic region has been solved separately, it is necessary to pass information upward along the tree since some merging operations may be necessary across the boundaries of cut-lines.

Mapping a binary tree process to a tree-of-processors architecture is straight forward. Similarly, a quad-tree process can be mapped to a pyramidal architecture in a natural way. Trees and pyramids are easily *embedded* into a network such as a hypercube [R+88]; therefore mapping tree-structured algorithms to a hypercube is also straight-forward. The drawback of this method is that it results in a poor processor utilization, since only one level of the tree machine is active at any instant of time. In Chapters 4 and 5, we have seen how to overcome this drawback. At each node of the tree is a process which can be further decomposed for parallel execution. The problem size handled by a process at level k of the tree is 2^k. Furthermore, there are 2^{h-k} nodes at level k, h being the height of the tree. The number of processors required to decompose a node process is proportional to the problem size handled by the process. Thus the number of processors needed at level k is proportional to $2^{h-k} \times 2^k = 2^h$. Since 2^h is the number of atomic regions, it is related linearly to original problem size n. Therefore, $\Theta(n)$ processors are necessary and sufficient to

parallelize each level of the tree. A linear array of n processors was used in Chapters 4 and 5.

The divide-and-conquer process will involve the following three types of communication. Information must be passed from a node to its children during the division step. When pasting the partial solutions, communication is necessary among the nodes at the same level. Information must also be passed from nodes to their parents. A tree or a pyramidal structure allows for all the three forms of interprocess communication. On the other hand, when a processor array is used, as in Chapters 4 and 5, an appropriate interconnection network must be used along with the array. We employed a communication-efficient architecture called the Orthogonal Array which uses n^2 memory modules and allows the n processors to access the memory modules in a non-conflicting manner.

7.2.1.2 Iterative Algorithms

Iterative improvement and Simulated Annealing are general-purpose techniques for combinatorial optimization; the two algorithms are identical in structure. Both algorithms can be viewed as combinatorial search algorithms; these algorithms start with an initial state and search in the local neighborhood of the state for better solutions. If none of the neighborhood states are better than the current state, the iterative improvement algorithm terminates; under the same condition, the annealing algorithm moves on to the next lower temperature. The core of either of these algorithm consists of three steps : S_1: make a small perturbation to the current state and generate a neighbor state S_2: evaluate the cost improvement from current state to new state, and S_3: accept or reject the new state. As mentioned in section 1.4, temporal parallelism permits the pipelined execution of steps S_1, S_2, and S_3. The step S_2 is a long computation in the loop when compared to steps S_1 and S_3. In order to eliminate this speed mismatch, step S_2 must be further parallelized. In Chapter 4, pipelined cost evaluation is used to speed up step S_2. To further enhance the performance of an iterative algorithm, the algorithm can be modified to evaluate multiple state transitions. In Chapter 4, we classified such algorithms

into three classes – SPSD, MPMP and MPSD – based on the number of transitions evaluated and number of accept/reject decisions made in one iteration.

7.2.1.3 Maze Algorithms

Routing algorithms impose a rectangular grid structure on the routing area. The problem of connecting two points on the grid can then be formulated as finding a shortest path between the two points e.g. Lee's algorithm [Lee61]. A two-dimensional mesh of processors is a natural choice for implementing Lee's algorithm. Several hardware accelerators for Lee routing are based on the mesh architecture [BS80, HN82, S+86]; a less obvious solution is the pipelined execution of Lee's algorithm [W+88]. Recently, we have extended Lee's algorithm for routing in several routing models other than the Manhattan model [RS90]; the router Vyuha presented by the authors can handle 45° wiring, routing in multiple layers, mixed 45° and 90° routing, channel routing, switchbox routing and irregular routing areas. The existing hardware accelerators for Lee routing cannot be directly used for executing Vyuha. The wavefront in a conventional Lee router is diamond-shaped since the algorithm tries to expand the wavefront in north, east, west and south directions; as a result, the near-neighbor connections of a two-dimensional mesh architecture suffice for wavefront expansion. Since Vyuha employs mixed 45° and 90° routing, a octagonal array would be more appropriate. Such an array has eight communication links per processor. Each non-peripheral processor is connected to its north, east, west, south, north-east, north-west, south-east and south-west neighbors (Figure 7.2).

7.3 Further Research

It is not uncommon to find the same layout problem being solved by a variety of different methods. The principal reason for this diversity is the fact that many layout-related problems are NP-complete. Since exact solutions cannot be found in polynomial time, fast heuristics are used by VLSI practitioners. Typical solution

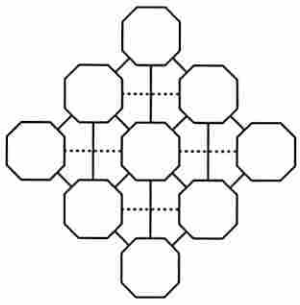

Figure 7.2: Octagonal Array for Modified Lee's Algorithm

methods involve two steps **(1)** *Reduction :* Formulate the problem using a formal model such as graph theory, linear algebra, or integer programming; the problem then reduces to an abstract problem such as graph coloring, maximum clique, or solution of linear equations. **(2)** *Approximation :* Apply a heuristic procedure to solve the abstract problem. Depending on the method of reduction, different parallel techniques may be applicable to solve the layout problem. In this section, we examine several popular paradigms for solving layout problems, namely, graph theory, integer programming, artificial intelligence and numerical optimization. The purpose of this section is to provide pointers for further research in these areas. Artificial neural networks, a relatively new technique for optimization, is examined in a separate section.

7.3.1 Graph Theory

There is a close relation between graph theory and circuit design. Undirected graphs, directed graphs and hypergraphs have been used to model design problems. In this dissertation, we have encountered circuit graphs to model connectivity information, interval graphs to model horizontal constraints in channel routing, and permutation graphs to model layer constraints in via minimization. Some more examples are included below.

Example 7.1 :

Given a set of two-point electrical nets, the problem of *topological via minimization* is to assign layers to each net such that no two nets cross in the same

layer; the objective is to minimize the vias. A graph model for this problem can be constructed as follows. Build a graph G by associating a node with each net. An undirected edge is drawn between nodes i and j if nets i and j intersect. If two nets do not intersect, they can be placed on the same layer without causing interference. In graph theoretic terms, two nets i and j can be placed on the same layer if a proper coloring of the graph assigns the same color to nodes i and j. In this example, there are only two layers to deal with; in other words, there are two colors to paint the intersection graph G. An instance of the problem is shown in Figure 7.3(a), where there are 3 nets to be assigned to two layers. Figure 7.3(b) shows the graph model for the problem.

Not all graphs are two-colorable. In fact, a graph is two-colorable only if it has no cycles of odd length. When G is not two-colorable, it has to be so rendered by adding some virtual nodes. If an odd-length cycle γ is detected in G, a dummy node can be added on one of the edges of γ to make the length of the cycle even. The graph of Figure 7.3(b) has a cycle of length 3; therefore, we add dummy node 4 to break this odd-length cycle as shown in Figure 7.3(c). Each such dummy node corresponds to a via hole. To illustrate this point, refer to Figure 7.3(d) which shows a topological routing of the channel shown in Figure 7.3(a); net 1 changes layer using the via hole.

From the above discussion, it may be concluded that the via minimization problem reduces to an abstract problem of adding as few dummy nodes as possible so that the intersection graph is two-colorable. This graph problem is NP-complete for general undirected graphs [GJ79].

Example 7.2 :

Consider a variation of the problem in Example 7.1 in which two or more layers are allowed. It is required to assign nets to a minimum number of layers such that there are no via holes. This problem is related to graph coloring. Consider a proper coloring of the intersection graph G using as few colors as possible. When two nodes are colored using the same color, they do not intersect and

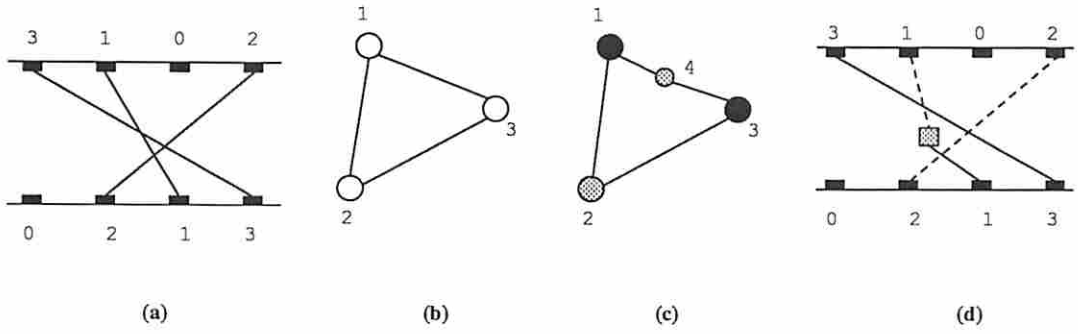

Figure 7.3: Topological Via Minimization in two layers.

can therefore be placed on the same layer. Minimizing the number of colors will maximize the number of nets that are assigned to the same layer. If the intersection graph is a general undirected graph, then the graph coloring problem in NP-complete. If the graph has some additional structure, it may be optimally colorable in polynomial time. For example, if the nets correspond to two-point nets in a channel, then G is a permutation graph, for which there is a fast algorithm for optimum coloring (see Chapter 6).

Example 7.3 :

If the aim of a channel router is to minimize the number of tracks and there are no vertical constraints, then the problem reduces to that of coloring interval graphs. Interval graphs are also optimally colorable in polynomial-time (see Chapter 6).

The graph formulation of design problems is useful in many ways. It can help analyze the complexity of the original problem. If the reduction graph has some structure to it, then the problem may be solvable in polynomial-time (see Golumbic's book [Gol80]). Even if the problem turns out to be NP-complete, good approximation algorithms are known for such problems as graph coloring, maximum independent set, maximum clique, minimal clique cover and many others (see [Deo81, PS82]). Parallel techniques have also been studied extensively for graph applications (see [Akl89, GR87]). Therefore, the importance of parallel graph techniques is evident for those who pursue research in parallel CAD algorithms.

7.3.2 Integer linear programming

Several interesting problems in VLSI design have been successfully modeled as instances of integer programming. In Section 1.6.4, we have examined a formulation of global routing as a 0-1 linear programming problem. Burstein and Pelavin formulated global routing as an integer programming problem [BP83]; Chapter 5 gives an overview of this formulation. A typical instance of integer linear programming takes the following form.

$$\begin{aligned} \text{Minimize} \quad & c_1\, x_1 + c_2\, x_2 + \cdots + c_n\, x_n \\ \text{Subject to} \quad & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m \\ & \\ & x_i \geq 0, \text{ integers.} \end{aligned}$$

c_i, b_i and a_{ij} are real constants. Integer programming is known to be a "hard" problem. One approach to find an approximate solution is to relax the requirement that x_i are integers. Under the assumption that x_i are real variables, the problem is transformed to *linear programming*, which admits deterministic polynomial-time solutions. A standard technique to solve linear programming is the Simplex method [BJ77]. It is possible to obtain an approximate solution to the integer programming problem by suitably rounding off the values of x_i that were obtained by solving the linear program [Lau86].

The size of an integer program (or a linear program) is defined by the number of variables n and the number of constraints m. Both n and m can be quite large for VLSI design problems. As a result, linear programming techniques are quite expensive in terms of CPU-time. Although parallel techniques have been investigated for linear programming [LM87], the state-of-the-art is still in its infancy. The field is open for further research. We have recently implemented an integer linear program solver on an Alliant FX/80 in FX/Fortran. The program uses the relaxing technique mentioned above. The implementation has been tested for small examples where the

solution can be verified by hand. We are currently in the process of generating data for large size examples.

7.3.3 Artificial Intelligence

Rule-based expert systems have been developed for routing, floor planning, and logic synthesis [Ack88]. It is argued that rule-based systems can approach the layout problem with the expertise of a human designer. Unlike conventional algorithms, they are capable of learning from experience. They accommodate backtracking to compensate for incomplete design knowledge. Rule-based systems have been conventionally implemented as *production systems*, consisting of an *inference engine*, a *rule memory*, and a *working memory*. Design rules are stored as a set of production rules (**if** < *condition* > **then** < *action* >) in the rule memory. The working memory is a global database of objects that describe the current state of the problem. An inference engine provides the control by determining which rules are ready to fire and scheduling the rules for execution. The advantages of expert systems are somewhat dimmed by their poor speed performance. Therefore, they are good candidates for parallel processing. However, to this date, there is no work on parallelizing rule-based layout systems.

The Branch-and-bound (BAB) algorithm is used as a powerful search mechanism in AI applications. The BAB algorithm holds high promise for parallel algorithmists, since it is inherently parallel. Several authors have investigated the parallelization of the BAB algorithm (see [J+88, LS84]). Being an inherently enumerative technique, the BAB technique is known to yield high quality results when applied to layout problems. For example, Hanan and Kurtzberg report the results of applying BAB techniques for module placement [HK72]. Unfortunately, the method is too expensive when used for problem sizes larger than 25 or 30. Needless to say, BAB is a good candidate for parallel processing. There is a great deal of potential for further research in this direction.

7.3.4 Numerical Optimization

Several layout problems have been formulated as numerical optimization problems. In Chapter 4, we have seen the formulation of the linear placement problem as an eigensystem problem. In this section, we shall illustrate several other problems which can be solved using purely numerical techniques.

Example 7.4 :

Cheng and Kuh [CK84] have related the placement problem to the minimization of power dissipation in a resistive network. A simplified version of their model is presented below. Consider a linear passive network composed of resistors. There are n inputs and n outputs to the network. Let $\mathbf{V_1}$ indicate the vector of input voltages and $\mathbf{V_2}$ the vector of output voltages. Let $\mathbf{Y}$ be the admittance matrix of the network. $\mathbf{Y}$ can be decomposed into $\mathbf{Y_{11}}$, $\mathbf{Y_{11}}$,$\mathbf{Y_{21}}$, $\mathbf{Y_{22}}$ i.e.

$$\mathbf{Y} = \begin{pmatrix} \mathbf{Y_{11}} & \mathbf{Y_{12}} \\ \mathbf{Y_{21}} & \mathbf{Y_{22}} \end{pmatrix}$$

$\mathbf{Y_{11}}$ represents the self admittance of the network as seen from the input; similarly, $\mathbf{Y_{22}}$ represents the self admittance as seen from the output ports. $\mathbf{Y_{12}}$ and $\mathbf{Y_{21}}$ are trans-admittance terms. Let $\mathbf{I_1}$ and $\mathbf{I_2}$ respectively be the current vectors at the input and output ports. The network equations are

$$\mathbf{I_1} = \mathbf{Y_{11}V_1} + \mathbf{Y_{12}V_2}$$
$$\mathbf{I_2} = \mathbf{Y_{21}V_1} + \mathbf{Y_{22}V_2}$$

The vector $\mathbf{I_2}$ is the zero vector, since all the output nodes are floating. The power dissipation P in the network is given

$$P = (\mathbf{V'_1}, \mathbf{V'_2})\,\mathbf{Y} \begin{pmatrix} \mathbf{V_1} \\ \mathbf{V_2} \end{pmatrix}$$

which can be written as

$$P = \mathbf{V}_1'\mathbf{Y}_{11}\mathbf{V}_1 + 2\mathbf{V}_1'\mathbf{Y}_{12}\mathbf{V}_2 + \mathbf{V}_2'\mathbf{Y}_{22}\mathbf{V}_2 \tag{7.1}$$

If we momentarily set the output voltages to zero, the power dissipation becomes

$$P = \mathbf{V}_1'\mathbf{Y}_{11}\mathbf{V}_1 \tag{7.2}$$

Comparing Equation 7.2 to the Equation 4.14 above, the following analogies can be drawn. The *sum of wire lengths* z corresponds to the power dissipation P in the network. The admittance matrix $\mathbf{Y_{11}}$ is analogous to the matrix B as defined by Equation 4.15. The voltage V_{1i} at the input i corresponds to the position x_i of module i. Therefore, minimizing the *sum of wire lengths* z in a linear placement problem amounts to minimizing the power dissipation in the resistive network by adjusting the voltages V_{1i}. Of course, the trivial solution to the minimization problem is obtained by setting $V_{1i} = 0$. To avoid this trivial solution, we need some constraints. In the placement problem, each module must be assigned a unique slot position. Therefore, the values x_i must be unique. Let $\sum_{i=1}^{n} x_i = d$. Since there is a one-to-one correspondence between voltage V_{1i} and the module position x_i, we have $\sum_{i=1}^{n} V_{1i} = d$. Alternately, we write

$$\mathbf{1}'\mathbf{V}_1 = d \tag{7.3}$$

Therefore, the placement problem translates to minimizing the quadratic function P given by Equation 7.1, subject to the linear constraints posed by Equation 7.3. Kuh, Tsay and Hsu employed the above technique in their PROUD placement algorithm for sea-of-gates integrated circuits [TKH88]. The matrix $\mathbf{Y}$ for a large placement problem is sparse. Therefore, the Successive Over Relaxation method (SOR) was used for optimization. The SOR technique is quite expensive in CPU time for large circuits; several hours were required for a chip with a few thousand transistors. Since the algorithm can be expressed

conveniently in terms of vector operations, it is a good candidate for vector supercomputing. ■

Example 7.5 :

Quinn and Breuer formulated PCB placement as a numerical problem of solving a set of non-linear algebraic equations [JB89]. Components are viewed as particles subject to a system of "forces." Two types of components are distinguished, fixed components and movable components. I/O pads and heat sinks are examples of fixed components, since their positions on the PCB are preassigned. Let M is the number of movable components and N be the number of fixed components.

1. If two components i and j are connected by C_{ij} wires, there exists a force of attraction F_{ij}^a between the components, given by

$$F_{ij}^a = C_{ij}\Delta_{ij} \tag{7.4}$$

   where Δ_{ij} is a *vector* whose magnitude is equal to the Manhattan distance between the components. The vector Δ_{ij} has a direction along the line joining module i to module j. Since two-dimensional placement is being considered, the force F_{ij}^a can be resolved into two parts along the x and y directions. Note that higher the connectivity between two modules, the larger the force of attraction between them trying to place the two modules close together.

2. There is also a force of repulsion F_{ij}^r between two modules i and j that are not connected. The direction of the force is along the line joining modules i and j, and the magnitude of the force is given by

$$|F_{ij}^r| = \frac{1}{T}\sum_{i}^{M}\sum_{j=1}^{M} C_{ij} \tag{7.5}$$

   where T is the number of connectivity terms that are nonzero. The force of repulsion can also be resolved into its x and y components.

3. Fixed components exert a collective force F^c on movable components. This force is called the *force on center of mass*. Let F_x^c and F_y^c indicate the x and y components of the force F^c. Then

$$F_x^c = \sum_{i=1}^{M} \sum_{j=M+1}^{N} -C_{ij}\Delta x_{ij} \tag{7.6}$$

$$F_y^c = \sum_{i=1}^{M} \sum_{j=M+1}^{N} -C_{ij}\Delta y_{ij} \tag{7.7}$$

Now consider the total force F_i on a component i and let F_{x_i} and F_{y_i} indicate its x and y components. F_{x_i} is the sum of three terms corresponding to **(a)** the collective force of attraction, **(b)** the collective force of repulsion, and **(c)** the x component of the force on the center of mass of i.

$$F_{x_i} = \sum_{j=1}^{N} -F_{x_{ij}}^a + \sum_{j=1}^{N} F_{x_{ij}}^r - \frac{F_x^c}{M} \tag{7.8}$$

A similar equation can be written down for F_{y_i}. The system of components is in a 'rest' state, the state of minimum energy, if the total force on each component is zero. Therefore, a solution to the relative placement problem is found by setting $F_{x_i} = 0$ and $F_{y_i} = 0$ and solving for $x_1, x_2, \cdots, x_M$ and $y_1, y_2, \cdots, y_M$. Then (x_i, y_i) gives the position of module i on the board. The equations $F_{x_i} = 0$ and $F_{y_i} = 0$ form a non-linear system of equations [JB89]. Classical techniques, such as Newton-Raphson method, can be used to solve these equations. It is known that Newton-Raphson converges quadratically i.e. no significant changes occur to the values of variables x_i, y_i after M^2 iterations. Further, the amount of arithmetic operations per iteration is $O(M^2)$. For large values of M, the force-directed placement method requires considerably large CPU-time. Therefore the method is seldom used for chip design. However, since the algorithm is expressible in terms of vector operations, vector machines are good computing platforms for these algorithms. ■

7.3.5 Artificial Neural Networks

The main goal of the fifth generation computers has been to solve difficult problems such as vision, pattern recognition, and natural language understanding. The first four generations of computers have not been able to tackle these problems, despite the fact that the fourth generation computers included powerful mainframes which achieved speeds of several million instructions per second. On the other hand, the human nervous system routinely solves these difficult problems. Artificial intelligence techniques, which were predominantly the theme of fifth generation computers, have been only partially successful in solving problems such as computer vision. Recently, there has been a revival of interest in neural computing and natural intelligence techniques. It is believed by a large number of computer professionals that neural computing is the key to solving difficult problems like pattern recognition and computer vision [Was89]. Artificial neural networks have been named as the theme of sixth generation computers.

The main component of an artificial neural network (ANN) is an artificial neuron. An artificial neuron receives several analog inputs $X_1, X_2, \cdots, X_n$ and generates a single analog output OUT. The output is computed as follows. Each input is weighted down by the neuron; let W_i be the weight associated with input X_i. The net input, denoted NET, is given by

$$NET = \sum_{i=1}^{n} W_i \cdot X_i \tag{7.9}$$

The output is a function F of NET (Figure 7.4). The function F is also known as the activation function of the neuron. A popularly used activation function is the sigmoid function $F(x) = 1/(1 + e^{-x})$. If x is a sufficiently large positive number, the sigmoid function approximates to unity. For sufficiently large negative values of x, the sigmoid function is close to 0. Another popular activation function is $F(x) = tanh(x)$.

Several artificial neurons can be connected to form an artificial neural network. For example, a single layer *feed forward* network consists of m neurons, each with n

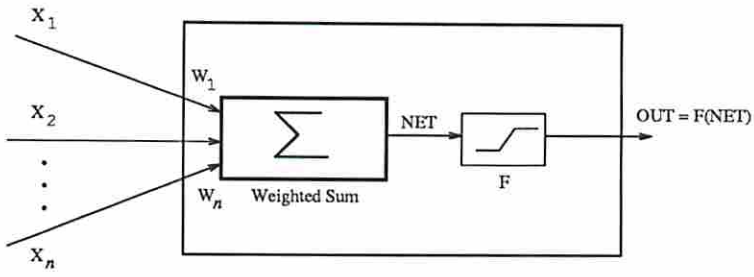

Figure 7.4: An artificial neuron.

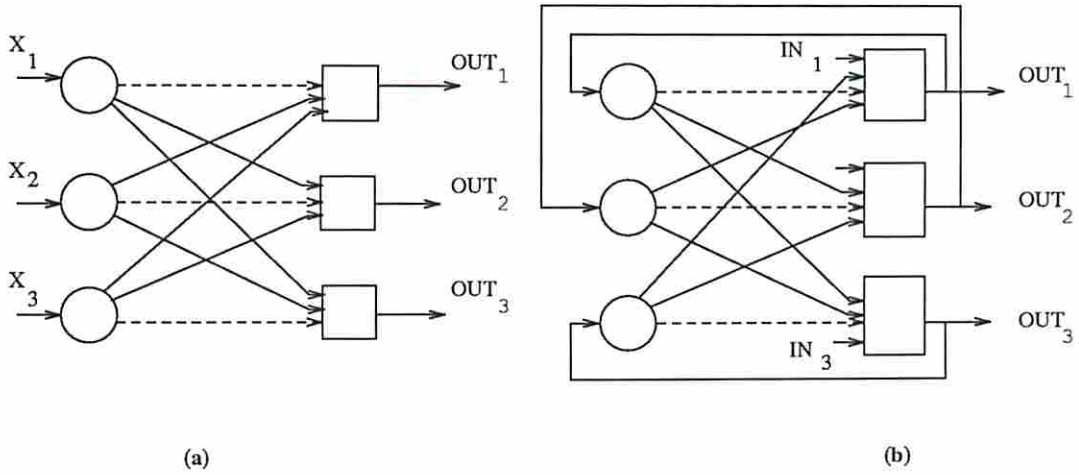

Figure 7.5: (a) A single-layer, feed-forward artificial neural network with 3 neurons. (b) A single-layer, recurrent artificial neural network with 3 neurons.

inputs. The principal inputs to the network are denoted $X_1, X_2, \cdots, X_n$. The weights associated with neuron i are denoted $W_{i1}, W_{i2}, \cdots, W_{in}$. The $m \cdot n$ weights of the network can be compactly represented by the $m \times n$ weight matrix $W = [W_{ij}]$. Figure 7.5(a) shows a feed forward network with three neurons, each with three inputs. The output of neuron i is denoted by OUT_i. A single layer *recurrent* network is similar to a feed forward network, except that the outputs are fed back as inputs to the network. Figure 7.5(b) shows a recurrent network with three neurons, each with three forward inputs and one feed back input. Hopfield and Tank [HT85] used recurrent neural networks to solve optimization problems.

7.3.5.1 Energy function and Stability

Just as temperature plays an important role with Simulated Annealing [KGV83], energy plays an important role with Hopfield's neural networks. The set of all outputs OUT_i is known as the state of the network. Suppose that the activation function of each neuron in the network is a *threshold* function i.e.

$$OUT_i = \begin{cases} 1 & \text{if } NET_i > T_i \\ 0 & \text{if } NET_i < T_i \\ unchanged & \text{if } NET_i = T_i \end{cases} \tag{7.10}$$

where T_i is the threshold level of neuron i. Since we are dealing with a recurrent network, NET_i is given by

$$NET_i = (\sum_{j \neq i} W_{ij} \cdot OUT_i) + IN_i \tag{7.11}$$

It is clear that the network can be in 2^n different states, since each of the n neurons can output either 0 or 1. Each state is associated with an energy level. When the network changes state, there is a change in its energy level. It is known that the network will settle down to a state with minimal energy level if the weight matrix W is a symmetric matrix and all the diagonal entries of the matrix are 0. The network is said to *converge* to the state of minimal energy. By constructing a neural network whose energy function is the objective function of a minimization problem, one can hope to solve the minimization problem.

Example 7.6 :

Consider how an artificial neural network can be set up to solve the two-way circuit partition problem. Given n circuit modules and a connectivity matrix $C = [C_{ij}]$, where C_{ij} denotes the connectivity between module i and module

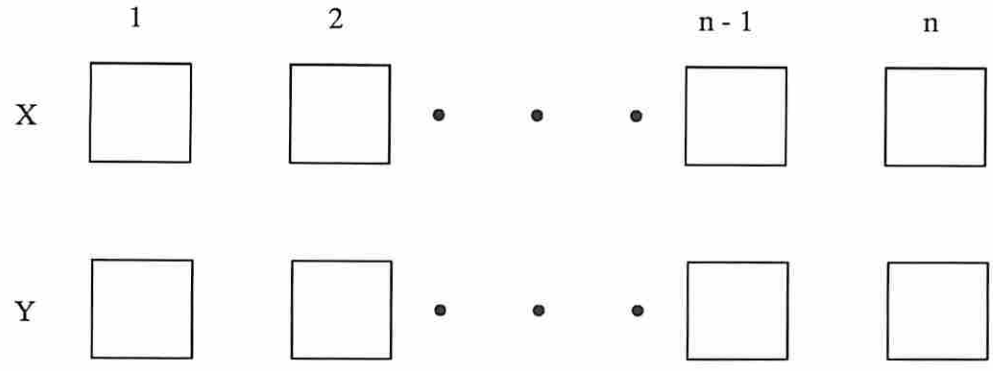

Figure 7.6: An Artificial Neural Network for Two-way Circuit Partition.

j. The objective is to divide the modules equally among sets X and Y such that the following cost function is minimized.

$$EW = \sum_{i \in X} \sum_{j \in Y} C_{ij} \tag{7.12}$$

To solve the problem, a network with $2n$ neurons can be set up; this network consists of a $2 \times n$ matrix of neurons (Figure 7.6). Each column of this matrix corresponds to a circuit module. The two rows correspond to the two sets X and Y. Consider two neurons in column i of this matrix. Their outputs are denoted by OUT_{Xi} and OUT_{Yi} respectively. If $OUT_{Xi} = 1$ then module i is assigned to set X. Similarly, if $OUT_{Yi} = 1$, the module i is assigned to set Y. Now consider the row X of the network. In order to achieve a balanced partition, the sum $\sum_{i=1}^{n} OUT_{Xi}$ should be $n/2$. (Without significant loss in generality, it is assumed that n is an even number.) Similarly, $\sum_{i=1}^{n} OUT_{Yi}$ should be $\frac{n}{2}$ for achieving a balanced partition. The external weight EW is given by $\sum_i \sum_j OUT_{Xi} OUT_{Yj} C_{ij}$. The following *energy function* can be associated with the network. By minimizing the energy function E, the network will arrive at a minimum-cost balanced partition.

$$E = \frac{\alpha}{2}\left(\sum_{i=1}^{n} OUT_{Xi} - \frac{n}{2}\right)^2 + \frac{\beta}{2}\left(\sum_{i=1}^{n} OUT_{Yi} - \frac{n}{2}\right)^2 + \frac{\gamma}{2} \sum_i \sum_j OUT_{Xi}\ OUT_{Yj}\ C_{ij} \tag{7.13}$$

In Equation 7.13, α, β, γ are constants. The first two terms of the energy function account for a balanced partition, while the third term accounts for the minimization of the external wiring.

The energy function E has several minima, some of which are local minima; the network can converge to any one of them. As a result, there is no guarantee that the solution obtained will correspond to a global minimum. Moreover, how does one determine the parameters of the network (the weight matrix, thresholds, the constants involved in the energy function and the activation function)? How sensitive is the final solution to small variations in these parameters? How good is the final solution when compared to other known techniques for solving the same optimization problem? And finally, how fast does the network converge to the final solution? Since neural computing is still an active research area, the answers to these questions are still being investigated.

Analog integrated circuit technology is ideally suited for artificial neural networks. Mead and his associates have implemented analog neural chips to solve problems in computer vision and computer hearing [Mea89]. An alternative to analog implementation is simulation using digital computers. Since the number of neurons in the network can be quite large ($2n$ neurons were required to solve the partition problem in Example 7.6), a simulation of the network can be quite expensive in CPU-time. The advantage of simulation is, of course, that that a single general-purpose computer can be used to simulate more than one network. Parallel processing is expected to play a major role in the simulation of neural networks on digital computers.

The impact of neural computing on VLSI CAD is still unclear. In a recent study, Yu [Yu89] reported the results of applying Hopfield neural networks to the placement problem. His results were not promising. Some of the difficulties pointed out by him are long simulation times, poor solution quality and high sensitivity of the solution to network parameters. At this stage, it can only be concluded that more research is required in order to understand the applicability of neural networks to VLSI CAD problems.

Appendix A

Parallel Computers – Examples

This appendix describes three parallel computer architectures. The first section describes an SIMD array architecture known as the Orthogonal Array due to its orthogonal memory access feature. The second section introduces the Alliant FX/80 supercomputer architecture. The Alliant FX/80 is a shared memory multiprocessor. In addition, each processing element in Alliant FX/80 is a vector processor; in this sense, the machine has a a hybrid architecture. The last section describes Intel iPSC/2, a distributed memory multiprocessor.

A.1 Orthogonal Array SIMD Architecture

An *Orthogonal Array* has an SIMD organization. It consists of a one dimensional array of P processing elements and a two-dimensional array of P^2 memory units. A control processor [HB85] supervises the operation of the processing elements. The processing elements (PEs) are denoted $PE_0, PE_2, \cdots, PE_{P-1}$. The memory units are organized in the form of an $P \times P$ mesh; M_{ij} indicates the memory unit in row i and column j. The processors are not connected through an interconnection network. Interprocessor communication is achieved through shared memory. When data must be transferred from PE_i to PE_j, the processor PE_i must write its data into a common memory location. In the next step, PE_j reads the data from the common location. A processor does not have access to all the memory units; the processing element PE_i can only access the memory units that are in the row i or column i. In order to avoid memory conflicts, the following access rule is imposed. At any time, only the rows or only the columns of memory can be accessed. The architecture derives its name from this *orthogonal access* feature. Figure A.1(a) shows an orthogonal array with $P = 4$.

There are three types of busses in an orthogonal array. A *row bus* is used to connect a processor i to all the memory elements of the type M_{i*}. Similarly, a *column bus* connects processor i to all memories of the form M_{*i}. A *broadcast bus* is used to transfer instructions and data from the control processor to the processing elements. Each processing element has a arithmetic/logic unit capable of basic

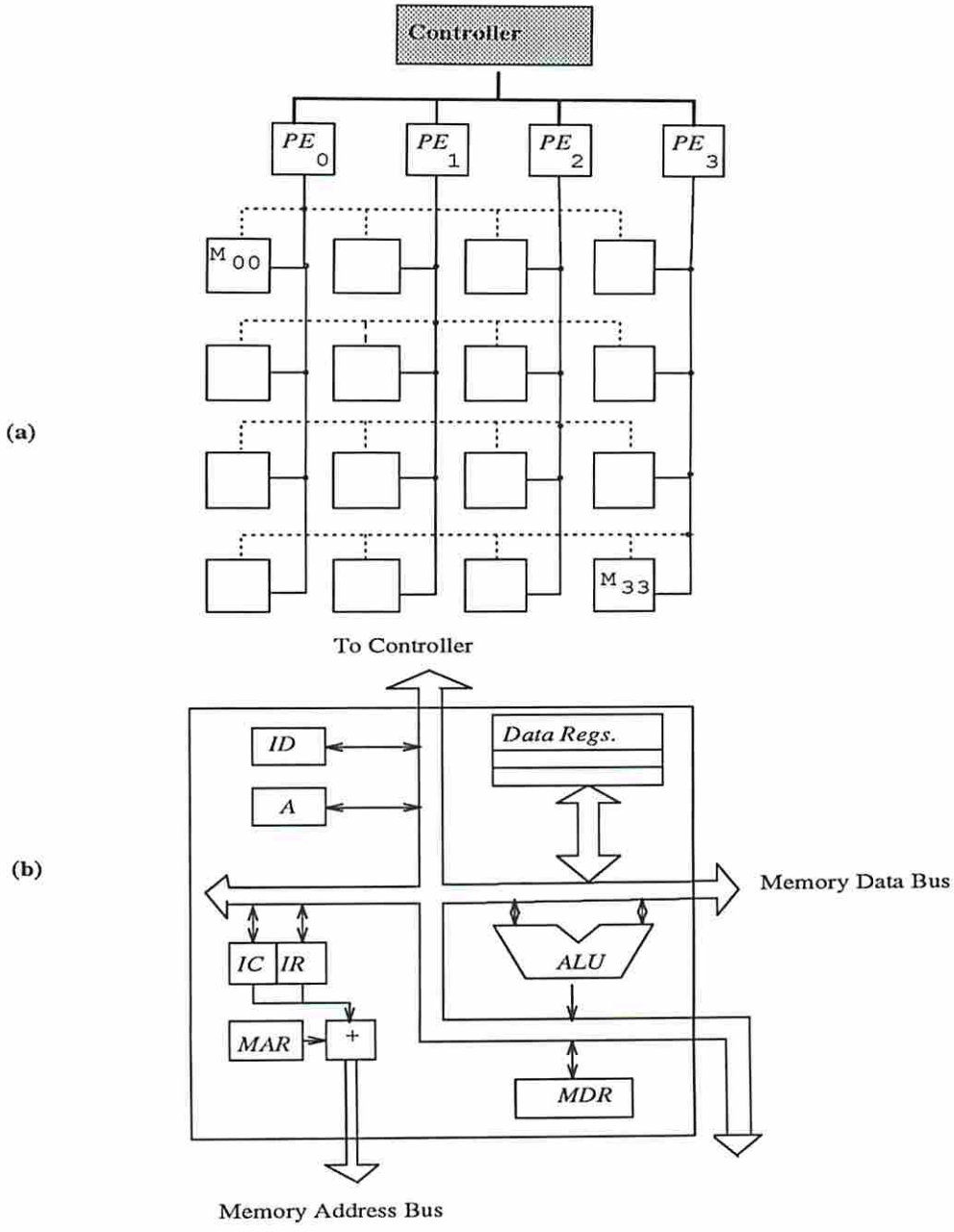

Figure A.1: (a) Orthogonal Array, $P = 4$. Row busses are shown dotted, and Column busses are shown solid. (b) Internal Organization of a Processing Element PE_i. The memory address bus and memory data bus are shown separately for clarity. In reality, these are multiplexed to form a single memory bus which connects to row bus i and column bus i.

operations such as integer addition, multiplication, arithmetic shift, etc. In addition, there are several registers in each processor (see Figure A.1(b)). *Data registers* are used to hold the intermediate results of computation. A single-bit *flag register A* is provided to indicate the 'active/inactive' status of the processor. A processor participates in the execution of a vector instruction only if its status register contains a logic 1. A *memory address register* indicates the row (or column) number of the memory element to be accessed by the processor. An *index register* indicates the particular location in the memory unit being accessed. A *tag register ID* holds the identification address of the processing element. A *memory data register* is provided for data transfer from a memory unit to the processor.

The innovation in the design of Orthogonal Array architecture is the fact that it uses massive memory rather than massive processing power. Other SIMD array architectures such as two-dimensional mesh and pyramid use a large number of processors with a near-neighbor interprocessor interconnection network. In such massively parallel architectures, the granularity of the processing element is small due to cost limitations. In current VLSI technology, a 32-bit multiplier can occupy an entire chip. Therefore, massively parallel machines tend to use bit-serial arithmetic inside each processing element. For example, the Connection Machine (from Thinking Machines Corporation) has 65,536 bit-serial processors; the Massively Parallel Processor (built at NASA) has 16,384 bit-oriented processors. The orthogonal array is not a massively parallel architecture; it is a massive memory architecture. Unlike processors, memory can be implemented using much less silicon area using current VLSI technology. Dynamic RAM chips with up to 256 Kbits are currently available and higher memory densities are announced each year. The entire two-dimensional memory of the orthogonal array can be designed as a single chip, or as a set of chips with the current VLSI technology. In the same way, the processors can be built as into a separate set of integrated circuits. This separation between processors and memory offers several advantages such as modular design, simpler testing procedures, and easier fault recovery. Due to the above considerations, the orthogonal array architecture is a cost-effective alternative for high speed computation. It can be built either as a special-purpose processor or as a general-purpose processor. When built as a special-purpose processor, the algorithm is hard-wired into the control unit and the processor architecture is optimized for the particular application. If the machine is intended for general-purpose computing, the control unit must be designed more elaborately to fetch instructions from a control memory, execute scalar instructions, and broadcast vector instructions to the processor array. The orthogonal array is also known as the *Reduced Mesh of Trees*, or the *Reduced Array*. Several applications such as signal processing, image processing and computational geometry have been studied for the orthogonal array [Aln88]. In this dissertation, parallel algorithms for circuit partition, circuit Placement, and global routing have been discussed.

A.2 The Alliant FX/80

Figure A.2 shows the architecture of the Alliant FX/80 multiprocessor [All87]. The exact configuration will vary from one model to another. The configuration shown in Figure A.2 corresponds to the machine Hydra at University of Southern California. The main building block of the Alliant is the advanced computational element (ACE). Up to 8 ACEs are available for parallel operation. Each ACE is a microprogrammed, pipelined processor and supports the instruction set of the Motorola 68020. In addition, the extended instruction set of each ACE includes floating point as well as vector instructions. These can handle double precision (64-bit) arithmetic. The peak MFLOPS rating of an ACE is 23.6. The ACEs are connected by means of a 40-bit wide bus known as *concurrency control bus.*

The memory unit of the Alliant consists of eight modules of 8 MB, totaling to 64 MB. In a fully configured system, the memory unit has 256 MB. Each memory module is 4-way interleaved to increase the memory bandwidth. The memory bus is a high speed, synchronous bus, which consists of two 72-bit wide bidirectional data paths. The bandwidth of the memory bus is 188 MBytes/sec. A cache memory is interfaced between the ACEs and the memory unit. This cache unit is called CP cache (for Computational Processor cache) and consists of two sections of 256 KB each. Communication between the ACEs and the CP cache takes place through a crossbar interconnection network.

Apart from the computational elements, the Alliant also supports up to 12 *interactive processors* (IPs). In the configuration of Figure A.2, only 3 IPs are shown. While the ACEs carry out CPU-intensive tasks, IPs handle interrupts, terminals, disks, tapes and network traffic. The IPs interface to the memory bus through an IP cache. In a fully configured system, IP cache consists of 4 segments of 32 KB each. Only one segment is shown in the diagram.

A.2.1 Operating System

The Alliant FX/80 runs the Concentrix Operating System, which is based on the 4.2 BSD version of UNIX. Under Concentrix, three separate classes of computational resources may be identified. These are the *computational complex*, detached ACEs and interactive processors; the user has the flexibility of specifying which of the three platforms should be used to run his/her job. A computational complex consists of two or more ACEs working in parallel. Scheduling and load balancing is done automatically by the operating system. The Concentrix operating system has the capability to dynamically configure the available computational resources. For example, ACEs 1,2,3 and 4 can form a computational complex to run a single job in a concurrent mode. At the same time, ACEs 5,6,7 and 8 can execute in the detached mode, running independent jobs.

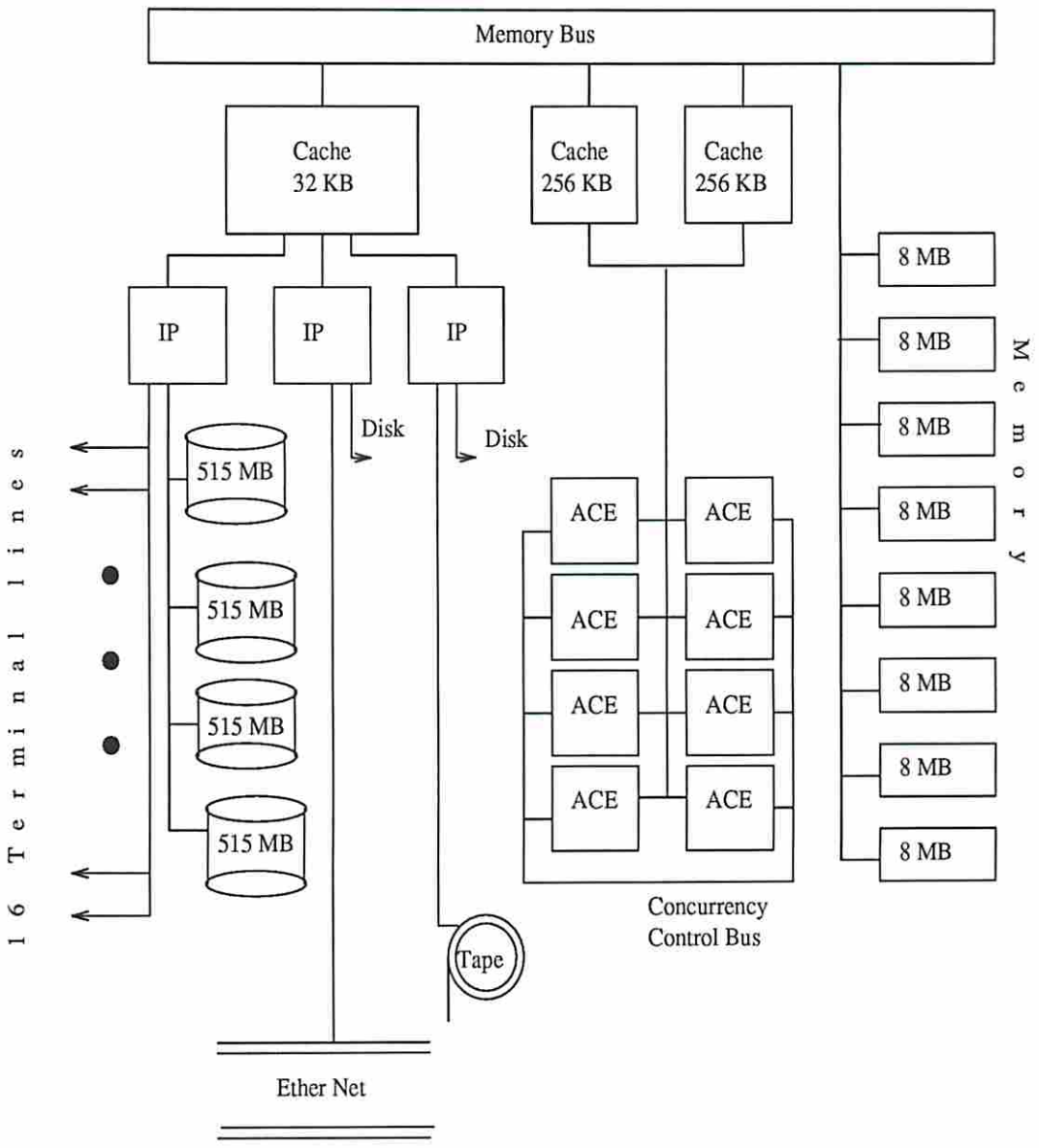

Figure A.2: Alliant FX/80 architecture. IP stands for Interactive Processor. ACE stands for Advanced Computational Element.

The user can create *processes* to run on a computational platform of his/her choice. These processes access the shared memory in the system. If two processes must share a variable, a lock and key mechanism must be used for correct operation. Thus a process P can only access a shared variable x if P can successfully lock x. After x has been suitably updated by P, the process unlocks the variable so that another process is able to obtain access to x. The commands to create processes and to access shared variables are available as library procedures that can be called from high level languages such as C. The Concentrix operating system supports compilers for several languages – FX/C, FX/Fortran, FX/Ada and FX/Pascal.

FX/Fortran on the Alliant is equipped with a parallelizing compiler. Different types of optimization can be specified as command-level options to the Fortran compiler. These optimizations are scalar optimization, vectorization and concurrentization. One or more of these options can be specified at a time. The compiler analyzes the Fortran source code for data dependences and generates instructions that use the concurrency and vectorization features of the hardware. As a first step, the compiler looks for concurrency i.e. if loops and array operations can be scheduled on more than one processor for concurrent execution. Next, the compiler optimizes the program for vectorization i.e. the compiler aggregates several scalar operations into a vector operation. Finally, scalar optimization is carried out; this consists of eliminating redundant expressions and dead storage, moving invariant code out of DO loops, computing and propagating constants at compile-time, etc. The FX/Fortran compiler also provides an option to optimize programs for *associative* transformations.

The FX/Fortran compiler can generate five different types of code, namely, scalar, vector, scalar concurrent, vector concurrent and 'concurrent outer vector inner' (COVI). A scalar instruction is one that operates on a singleton element. Scalar instructions are executed on a single ACE. A vector is an array of elements. A vector instruction operates on one or more vectors.

Example A.1 :

Suppose B is the name of an array of 200 real numbers. Each element of B is a scalar. The entire array B is a vector. The odd-indexed elements of the array B, namely $B(1), B(3), \cdots, B(199)$, also form a vector when considered together. Suppose that each odd-index element of B must be incremented by a scalar constant S. The scalar code for this function will be a series of instructions of the form

$$\begin{aligned} B(1) &= B(1) + S \\ B(3) &= B(3) + S \\ &\vdots \\ B(199) &= B(199) + S \end{aligned}$$

100 scalar instructions will be necessary to complete the operation. All these instructions are executed on the same ACE.

Example A.2 :

In the previous example, suppose that each element of the vector $B(1:200:2)$ must be incremented by a scalar constant S. At a time, an ACE can handle vectors of length 32. Therefore, on a detached ACE, the vector code for the above example requires 4 steps.

$$\begin{aligned} B(1:64:2) &= B(1:64:2) + S \\ B(65:128:2) &= B(65:128:2) + S \\ B(129:192:2) &= B(130:192:2) + S \\ B(193:199:2) &= B(193:199:2) + S \end{aligned}$$

In the scalar-concurrent mode, each of the available ACEs executes the next available iteration of the loop until the entire loop has been executed.

Example A.3 :

Suppose the code in the above example is optimized to operate in the scalar-concurrent mode. Assume that there are 4 ACEs in the system; the execution pattern is shown below. Thus, independent operations are scheduled for execution on the ACEs on an "as soon as possible" basis. When $B(1)$ is being updated by ACE 1, $B(2)$ can be simultaneously be updated by ACE 2, and so on.

ACE 1	ACE 2	ACE 3	ACE 4
$B(1) = B(1) + S$	$B(3) = B(3) + S$	$B(5) = B(5) + S$	$B(7) = B(7) + S$
$B(9) = B(9) + S$	$B(11) = B(11) + S$	$B(13) = B(13) + S$	$B(15) = B(15) + S$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$B(193) = B(193) + S$	$B(195) = B(195) + S$	$B(197) = B(197) + S$	$B(199) = B(199) + S$

The vector-concurrent mode is similar to the scalar-concurrent mode, except that the ACEs execute *vector instructions* concurrently.

Example A.4 :

The running example will be executed in the vector-concurrent mode as shown below; each ACE executes 25 operations in vector mode.

ACE 1	ACE 2
$B(1:193:8) = B(1:193:8) + S$	$B(3:195:8) = B(3:195:8) + S$
ACE 3	ACE 4
$B(5:197:8) = B(5:197:8) + S$	$B(7:199:8) = B(7:199:8) + S$

Finally consider the COVI mode of operation. When there are nested loops to be performed, the FX/Fortran compiler attempts to distribute the outer loop iterations across ACEs ("concurrent outer") and the inner loop iterations as vector operations within ACEs ("vector inner").

Example A.5 :

Consider the nested loop shown below.

```
      DO 100 I = 1, 8, 1
        DO 100 J = 1, 100, 1
           A(I,J) = A(I,J) * 2
100     CONTINUE
```

Given 8 ACEs, the 8 outer loops (indexed by I) can be carried out concurrently. Within each ACE, the inner loop (indexed by J) can be carried out in vector mode.

Given the FX/Fortran parallelizing compiler, it is easy to convert "dusty deck" Fortran programs to parallel programs. The latest version of FX/Fortran (Version 3) incorporates more sophisticated parallelization techniques such as optimization of recurrences, optimization of "if loops," alternate code and many others. For a discussion of these optimization techniques, see [All87]. The Alliant FX/80 also provides a source-level debugging facility (dbx) compatible with UNIX, and a host of other program development aids. Currently, parallelizing compilers are not available for other languages such as C and Pascal. However, library functions are provided for multitasking operations such as creation of a process, creation of a critical region, etc. Using these library functions, parallel programs can be developed in languages other than Fortran.

A.3 The Intel iPSC/2

The Intel iPSC/2 is an example of a distributed memory multiprocessor. The basic building block of the iPSC/2 is a *node*. The nodes of an iPSC/2 are connected by a hypercube interconnection network. A node consists of an Intel 80386 CPU attached to an Intel 80387 numeric coprocessor and up to 8 MB of on-board memory. Figure A.3 shows the organization of a single node. The operating speed of a cube node is 16 MHz, giving it a 4 MIPS performance. The CPU used in the node, the Intel 80386, is a 32-bit microprocessor with separate 32-bit address and data paths. It uses instruction pipelining for high speed operation. The 80386 chip also includes a memory management unit for translating logical addresses into physical addresses. The instruction set of 80386 supports 32-bit integer operations. For floating point operations, the numeric coprocessor 80387 is used. A scalar extension module, called Intel SX, is optionally available for higher performance. The SX board consists of Weitek 1167 numeric chips and is roughly 3 times faster than the coprocessor. The memory in each node is built from modules of 1,4 or 8 MB. In addition, 64 Kbytes cache memory is provided on each node.

The number of nodes depends on the configuration at the installation. The figures mentioned in this section pertain to the **ipsc2** machine at the University of Southern California. There are 32 nodes in the 'ipsc2,' connected by the hypercube interconnection network. A d-dimensional hypercube has 2^d nodes. The nodes are numbered $0, 1, \cdots, 2^d - 1$ using their binary representation. A node i is connected to

further create children processes to run on the same node. Processes running on a node i only have access to the memory of node i. If data must be exchanged between two processes running on two different nodes, then *message passing* is used. Suppose that process p on node i intends to send data to a process q running on node j, This communication will be successfully completed if p executes a send primitive (**csend**) and q executes a receive primitive (**crecv**). The process p must specify the destination of its message in the form of node identification number of j and the process identification number of q. Cube processes can exchange messages which are up to 3 MB long. The 'Direct-Connect' communications network is used for message transfer. The routing of the message is automatically handled by the operating system software.

In order to develop applications, the iPSC/2 supports several programming languages such as C, Fortran, Ada and Lisp. Parallelizing constructs are provided as calls to library functions. The library calls consist of cube primitives (e.g. getcube, relcube, load), interprocess communication primitives (e.g. csend, crecv), global operations (e.g. gsync, gisum) and miscellaneous functions (e.g. myhost, mypid). The global operations are useful for synchronization as well as associative computation. For example, the **gsync** primitive can be used to synchronize all the node processes. Since the iPSC/2 is an MIMD processor, the different node processes execute asynchronously. Thus one process may finish performing a certain task faster than another. There may arise situations when a decision must be taken by pooling the results of all the node programs. Under such a circumstance, global operations are useful. For example, the **gisum** operation is useful for the addition of integers – distributed one on each node process. Let P be the number of nodes in the cube. If $x_0, x_1, \cdots, x_{P-1}$ are integers with x_i residing in the memory of node i, the **gisum** function can be used to evaluate $x_0 + x_1 + \cdots + x_{P-1}$ and make the sum available to each node process. The **gisum** function also acts as a synchronizing primitive, since each node must place a call to the function in order to complete **gisum**. The iPSC/2 library provides several global operations such as giprod, giand, gior, etc. Miscellaneous functions such as **myhost** and **mypid** are also provided. The **myhost** function allows a node process to obtain the identification of the host, which is useful when a node process must send or receive data from its host. Similarly, **mypid** allows a node process to compute its process id.

A common programming model for the iPSC/2 is to write two separate programs – a node program and a host program. A copy of the node program is loaded into each cube node using the **load** command on the host. The nodes independently execute their copies of node programs on their private data sets. In this sense, the iPSC/2 is a true MIMD processor. The host program runs on the host machine and handles user interface and communication functions such as broadcasting data to all the nodes, receiving results from the nodes, etc. The node programs are also permitted to do file I/O. Therefore, a user may decide to eliminate the host program and write only the node program. This simplifies programming, but makes

debugging a little more difficult. For example, if there is an unconditional *print* statement in the node program, all the nodes will simultaneously send print messages to the screen. The order in which these messages will be actually printed cannot be controlled without introducing additional logic. But once the user becomes familiar with the system, the second programming model is much more simple.

The *concurrent file system* (CFS) supported by iPSC/2 allows the nodes to perform file I/O simultaneously. The CFS resides on a set of disks which is separate from the disk attached to the host. The user can create separate data files for nodes and transfer them to the CFS. The CFS is accessible as a directory `/cfs` if a special version of the UNIX shell, called the node shell (`nsh`) is run on the host. The node programs can open their private data files and read data concurrently. Alternately, all the data can reside on a single file in the CFS and the nodes can use the *file seek* operation to position their file pointers appropriately; after the file pointers have been so placed, data can be read simultaneously.

A source-level debugger called DECON is available on the iPSC/2. Like any other debugging utility, the DECON allows the user to set break points, list parts of source, step through statements, display variables, etc. The concept of a debugger *context* is used to identify which node and which process on the node is being referenced. A context is a tuple consisting of node id and process id. It can be dynamically changed during the course of debugging. The arguments to a DECON command refer to variables that are specific to the current context. In the experience of this author, programming the iPSC/2 is straightforward and most errors can be spotted if conditional print statements are intelligently used.

Reference List

[ABF90] M. Abromovici, M.A. Breuer, and A.D. Friedman. *Digital Systems Testing and Testable Design.* Computer Science Press, NY, 1990.

[Ack88] B.D. Ackland. Knowledge-based physical design automation. In Bryan Preas and Michael Lorenzetti, editors, *Physical Design Automation of VLSI Systems*, chapter 9, pages 408–460. The Benjamin/Cummings Publishing Company, Menlo Park, CA, 1988.

[AGN90] S.Devadas A. Ghosh and A.R. Newton. Verification of Interacting Sequential Machines. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 213–219, June 1990.

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison Wesley, Reading, MA, 1983.

[AK86] H.M. Alnuweiri and V.K. Prasanna Kumar. An efficient VLSI Architecture with Applications to Geometric Problems. In *24th Annual Allerton Conference of Communications, Control, and Computing*, 1986.

[AK87] H.M. Alnuweiri and V.K. Prasanna Kumar. Efficient Image computations on VLSI Architectures with Reduced Hardware. In *Workshop for Pattern Analysis and Machine Intelligence, Seattle*, 1987.

[Akl89] S. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[All87] Alliant Computer Systems Corp. *FX/Fortran Programmer's Manual*, 1987. 302-00001-C.

[Aln88] H.M. Alnuweiri. *Communication-Efficient Parallel Architectures and Algorithms for Image Processing.* PhD thesis, Department of EE Systems, University of Southern California, 1988.

[BB88] R.J. Brouwer and P. Banerjee. A Parallel Simulated Annealing Algorithm for Channel Routing on a Hypercube Multiprocessor. In *Proceedings of the International Conference on Computer Design*, pages 4–7, 1988.

[BHMSV84] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[BJ77] M.S. Bazaraa and J.J.Jarvis. *Linear Programming And Network Flows.* John Wiley & Sons, NY, 1977.

[BP83] M. Burstein and R. Pelavin. Hierarchical Wire Routing. *IEEE Transactions on Computer-Aided Design*, CAD-2:223–234, October 1983.

[Bre77] M.A. Breuer. Min-cut Placement. *Journal of Design Automation and Fault Tolerant Computing*, 1(4):343 – 362, October 1977.

[BS80] M.A. Breuer and K. Shamsa. A Hardware Router. *Journal of Digital Systems*, 4(4):393 – 408, 1980.

[C+87] A. Casotto et al. A Parallel Simulated Annealing Algorithm for the Placement of Macro-cells. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):838–847, September 1987.

[CB83] D.J. Chyan and M.A. Breuer. A Placement algorithm for Array Processors. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 182– 188, 1983.

[CK84] C.-K Cheng and E.S. Kuh. Module Placement based on Resistive Network Optimization. *IEEE Transactions on Computer-Aided Design*, pages 218–225, July 1984.

[Col88] R. Cole. Parallel Merge Sort. *Journal of the ACM*, 17(4):771–785, August 1988.

[DB89] J.S. Deogun and B.B. Bhattacharya. Via Minimization in VLSI routing with Movable Terminals. *IEEE Transactions on Computer-Aided Design*, 8(8):917–920, August 1989.

[Deo81] N. Deo. *Graph Theory and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W.H.Freeman, San Fransisco, 1979.

[GLL79] U.I. Gupta, D. T. Lee, and J. Y. T. Leung. An Optimal Solution for the Channel-Assignment Problem. *IEEE Transactions on Computers*, C-28:807–810, November 1979.

[Gol80] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, NY, 1980.

[Gop86] P.S. Gopalakrishnan. *Parallel Approximate Algorithms for Combinatorially Hard Problems.* PhD thesis, University of Maryland, 1986.

[Got81] S. Goto. An Efficient Algorithm for the Two-dimensional Placement problem in Electrical Circuit Layout. *IEEE Transactions on Circuits and Systems*, CAS-28:12–18, January 1981.

[GR87] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1987.

[GS84] J.W. Greene and K.J. Supowit. Simulated Annealing without Rejected Moves. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 658–663, 1984.

[Hal70] K.M. Hall. r-dimensional Quadratic Placement Problem. *Management Science*, 17:219–229, 1970.

[Han66] M. Hanan. On Steiner's Problem with Rectilinear Distance. *SIAM Journal of Applied Mathematics*, 14:255–265, 1966.

[HB85] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, 1985.

[HK72] M. Hanan and J.M. Kurtzberg. Placement Techniques. In M.A. Breuer, editor, *Design Automation of Digital Systems*, volume 1, pages 213 – 282. Prentice Hall, Englewood Cliffs, NJ, 1972.

[HL89] R.L. Hadas and C.L. Liu. Solutions to the Module Orientation and Rotation Problems by Neural Computation Networks. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 400–405, 1989.

[HN82] S.J. Hong and R. Nair. Wire Routing Machines – New tools for VLSI Physical Design. Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 1982.

[HS71] A. Hashimoto and J. Stevens. Wire Routing by Optimizing Channel Assignment Within Apertures. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 155–169, 1971.

[HS85] T.C. Hu and M.T. Shing. A Decomposition Algorithm for Circuit Routing. In T.C. Hu and E.S. Kuh, editors, *VLSI Circuit Layout : Theory and Design*, pages 144 – 152. IEEE Press, NY, 1985.

[HT85] J.J. Hopfield and D.W. Tank. Neural Computation of Decisions in Optimization Problems. *Biological Cybernetics*, 52:141–152, 1985.

[Hwa79] F.K. Hwang. An $O(n \log n)$ Algorithm for Suboptimal Rectilinear Steiner Trees. *IEEE Transactions on Circuits and Systems*, CAS-26:75–77, 1979.

[Hwa87] K. Hwang. Advanced Parallel Processing and Supercomputer Architectures. Technical report, Computer Research Institute, University of Southern California, March 1987. CRI-87-22.

[I+83] A. Iosupovici et al. A Module Interchange Placement Machine. In *Proceedings of the 20th Design Automation Conference*, pages 171 – 174, June 1983.

[II87] R.G. Babb II, editor. *Programming Parallel Processors.* Addison Wesley, Reading, MA, 1987.

[Int89] Intel Corporation. *iPSC/2 Programmer's Reference Manual*, March 1989. 311708-001.

[J+88] V.K. Janakiram et al. A Randomized Parallel Branch-and-Bound Algorithm. *International Journal of Parallel Programming*, 17(3):277–301, 1988.

[Jay87] R. Jayaraman. Floorplanning by Annealing on a Hypercube Architecture. Master's thesis, Carnegie-Mellon University, PA, Department of Electrical and Computer Engineering, March 1987. CMUCAD 87(9).

[JB89] N. Quinn Jr. and M. A. Breuer. A Force Directed Component Placement Procedure for Printed Circuit Boards. *IEEE Transactions on Computer-Aided Design*, CAD-26(6):377–387, June 1989.

[JK89] M.A.B. Jackson and E.S. Kuh. Performance-driven Placement of Cell Based IC's. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 370–375, June 1989.

[JKMP89] R. Jain, K. Kucukcakar, M.J. Mlinar, and A. C. Parker. Experience with the ADAM Synthesis System. In *Proceedings of the ACM/IEEE Design Automation Conference*, June 1989.

[Joh74] D. Johnson. Approximation Algorithms for Combinatorial Problems. *Journal Computer System Sciences*, 9(3):256–278, 1974.

[Kar77] R.M. Karp. Probabilistic Analysis of Partitioning Algorithms for the Traveling Salesman Problem in the Plane. *Mathematics of Operations Research*, 2(3):209–224, August 1977.

[KGV83] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 13 1983.

[Kra86] S.A. Kravitz. Multiprocessor based placement by Simulated Annealing. Master's thesis, Carnegie-Mellon University, PA, Department of Electrical and Computer Engineering, February 1986. CMUCAD 86(6).

[Kra87] S. A. Kravitz. Placement by Simulated Annealing on a Multiprocessor. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):534–550, July 1987.

[KS70] B.W. Kernighan and S.Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Systems Technical Journal*, 49(2):291 – 307, February 1970.

[KS84] R. Kane and S. Sahni. A Systolic Design Rule Checker. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 243–250, 1984.

[Kun79] H.T. Kung. Let's design algorithms for VLSI systems. In *Proceedings of the Caltech Conference on VLSI*, pages 65–89, 1979.

[L+84] S. Laxmivarahan et al. Parallel Sorting Algorithms. In M.C. Yovits, editor, *Advances in Computers*, pages 295–354. Academic Press, NY, 1984.

[L+85] E.L. Lawler et al., editors. *The Traveling Salesman Problem : A Guided Tour of Combinatorial Optimization.* John Wiley & Sons, NY, 1985.

[L+87] W.K. Luk et al. A Hierarchical Global Wiring for Custom Chip Design. *IEEE Transactions on Computer-Aided Design*, pages 518–533, July 1987.

[Lau86] H.T. Lau. *Combinatorial Heuristic Algorithms with Fortran.* Springer-Verlag, Berlin, Germany, 1986.

[LB88] M.J. Lorenzetti and D.S. Baeder. Routing. In Bryan Preas and Michael Lorenzetti, editors, *Physical Design Automation of VLSI Systems*, chapter 3, pages 157–210. The Benjamin/Cummings Publishing Company, Menlo Park, CA, 1988.

[Lee61] C.Y. Lee. An Algorithm for Path Connections and its Applications. *IRE Transactions on Electronic Computers*, VEC-10:346–365, Sep 1961.

[LM87] R.De Leone and O. Mangasarian. Serial and Parallel solution of Large Scale Linear Programs by augmented Lagrangean Successive Overrelaxation. Technical report, University of Wisconsin, Computer Sciences Department, November 1987.

[LS84] T.H. Lai and S. Sahni. Anomalies in Parallel Branch-And-Bound algorithms. *Communications of the ACM*, 27(6):594–602, 1984.

[M+90] A. Malik et al. Reduced Offsets for Two-level Multi-valued Logic Minimization. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 290–296, June 1990.

[MC80] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison Wesley, Reading, MA, 1980.

[Mea89] C. Mead. *Analog VLSI and Neural Systems.* Addison Wesley, Reading, MA, 1989.

[MG85] R. Mehrotra and E.F. Gehringer. Superlinear Speedup through Randomized Algorithms. In *Proceedings of the International Conference on Parallel Processing*, pages 291–300, 1985.

[MH85] G. Mott and R.B. Hall. The Utility of Hardware Accelerators in the Design Environment. *VLSI Systems Design*, pages 62 – 70, October 1985.

[O+84] J.K. Osterhout et al. Magic : a VLSI layout system. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 152–159, June 1984.

[OM87] O.A. Olukotun and T.N. Mudge. A preliminary investigation into Parallel Routing on a Hypercube Computer. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 814–820, 1987.

[OM89] O.A. Olukotun and T.N. Mudge. Hierarchical Gate Array Routing on a Hypercube Multiprocessor. Manuscript, 1989.

[Ost89] A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[P+77] G. Persky et al. LTX – A minicomputer-based system for Automatic Layout. *Journal of Design Automation and Fault Tolerant Computing*, 1(3):217–255, May 1977.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization : Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[R+88] S. Ranka et al. Programming a Hypercube Multicomputer. *IEEE Computer*, pages 69–77, September 1988.

[Rav87] C.P. Ravikumar. Hardware Accelerators for VLSI layout. Master's thesis, Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560 012, India, January 1987.

[Rav90a] C. P. Ravikumar. CAD accelerators. *SIGDA Newsletter*, 1990.

[Rav90b] C. P. Ravikumar. Parallel Combinatorial Search. Manuscript, 1990.

[Ros88] J. Rose. LocusRoute : A Parallel Global Router for Standard Cells. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 189–195, 1988.

[RP87] C.P. Ravikumar and L.M. Patnaik. Parallel Placement by Simulated Annealing. In *Proceedings of the International Conference on Computer Design*, 1987.

[RP90] C.P. Ravikumar and L.M. Patnaik. Performance Improvement of Simulated Annealing Algorithms. *Computer Systems - Science and Engineering*, 5(2), April 1990.

[RS88] C. P. Ravikumar and S. Sastry. Parallel Placement on Reduced array architecture. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 121–127, June 1988.

[RS89a] C. P. Ravikumar and S. Sastry. A Hardware Accelerator for Hierarchical VLSI routing. *Integration, the VLSI Journal*, 7:283–302, 1989.

[RS89b] C.P. Ravikumar and S. Sastry. Parallel Placement on Hypercube Architecture. In *Proceedings of the International Conference on Parallel Processing*, volume 3, pages 97–101, August 1989.

[RS90] C.P. Ravikumar and S. Sastry. VYUHA : A Detailed Router for Multiple Routing Models. Technical Report CENG-90-05, Computer Engineering, Department of EE-Systems, University of Southern California, May 1990.

[RS91a] C. P. Ravikumar and S. Sastry. A Parallel Approach to Three-Layer Channel Routing. In P. Ambler, W. Moore, and P. Agrawal, editors, *CAD Accelerators*, pages 259 – 272. Elsevier Science Publishers B.V., 1991.

[RS91b] C. P. Ravikumar and S. Sastry. Parallel Combinatorial Search Algorithms. Submitted to ACM/IEEE Design Automation Conference, 1991.

[RSL89] C. P. Ravikumar, S. Sastry, and L.M.Patnaik. Parallel Circuit-Partitioning on a Reduced Array Architecture. *Computer-Aided Design*, 21(7):447–455, September 1989.

[RSP88] C.P. Ravikumar, S. Sastry, and L.M. Patnaik. A Data Parallel approach to Local Search Placement Algorithms. In *Proceedings of the VLSI Design Workshop, India.* Computer Society of India, December 1988.

[RSP91] C. P. Ravikumar, S. Sastry, and L.M. Patnaik. Logic Cell Placement on Reduced array SIMD architecture. *Computer Systems – Science and Engineering*, 1991. Will appear.

[S+76] B.T. Smith et al. *Matrix Eigensystem Routines - EISPACK Guide, Second Edition*, volume 6. Springer-Verlag, Lecture Notes in Computer Science, 1976.

[S+86] K. Suzuki et al. A Hardware Maze Router with Applications to Interactive Rip-up and Reroute. *IEEE Transactions on Computer-Aided Design*, CAD-5(4):466 – 476, October 1986.

[SDK83] M.M. Syslo, N. Deo, and J. Kowalik. *Discrete Optimization Algorithms with Pascal Programs.* Prentice Hall, Englewood Cliffs, NJ, 1983.

[SG76] S. Sahni and T. Gonzalez. P-complete Approximation Problems. *Journal of the ACM*, 23(3):555–565, 1976.

[SK87] S. Sastry and V. K. Prasanna Kumar. Efficient Signal Processing on a Reduced Hardware VLSI array. In *Proceedings of the International Conference on Parallel Processing*, 1987.

[SR90] Y.G. Saab and V.B. Rao. Fast effective heuristics for the Graph Bisectioning problem. *IEEE Transactions on Computer-Aided Design*, 9(1):91–98, January 1990.

[SS84] K.J. Supowit and E.A. Slutz. Placement Algorithms for custom VLSI. *Computer-Aided Design*, pages 45–50, January 1984.

[SW84] Y. Sugiyama and T. Watanabe. Parallel Processing of Logic Module Placement. *Electronics Letters*, 20(5):219 – 220, March 1984.

[TKH88] R.-S. Tsay, E.S. Kuh, and C.-P Hsu. PROUD: A fast sea-of-gates placement algorithm. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 318–323, 1988.

[U+83] K. Ueda et al. A Parallel Processing Approach for Logic Module Placement. *IEEE Transactions on Computer-Aided Design*, CAD-2(1):39 – 47, January 1983.

[VZN+81] A. Vladimirescu, K. Zhang, A.R. Newton, D.O. Pederson, and A. Sangiovanni Vincentelli. SPICE Version 2G user's guide. Technical report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, August 1981.

[W+88] Y. Won et al. Hardware Accelerator for Maze Routing. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 800–806, 1988.

[Was89] P.D. Wasserman. *Neural Computing – Theory and Practice.* Van Nostrand Reinhold, NY, 1989.

[Wil88] R.B. Wilhelmson, editor. *High-Speed Computing : Scientific Applications and Algorithm Design.* University of Illinois Press, 1988.

[YK82] T. Yoshimura and E.S. Kuh. Efficient algorithms for channel routing. *IEEE Transactions on Computer-Aided Design*, CAD-1:25–35, January 1982.

[Yu89] M.L. Yu. A study of the applicability of Hopfield Decision Neural nets to VLSI CAD. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 412–417, 1989.

[Zar88] M.R. Zargham. Parallel Channel Routing. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 128–133, 1988.