PARALLEL ALGORITHMS FOR PLACEMENT AND ROUTING IN VLSI DESIGN

BY

RANDALL JAY BROUWER

B.S., Calvin College, 1985
M.S., University of Illinois, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

CHAPTER 1.

INTRODUCTION

1.1. Parallel Processing for CAD

In view of the increasing complexity of very large scale integrated circuits (VLSI), there is a growing need for sophisticated computer-aided design (CAD) tools to automate the synthesis, analysis, and verification steps in the design of VLSI systems.

Although the increased performance of today's processors has helped, there are still many tasks in VLSI CAD which continue to take a long time to finish. A recent approach to handling the problem's complexity and decreasing the running time of such tasks has been to apply parallel processing [1]. The advantages of parallel processing include: the ability to solve *larger problems sizes*, the ability to achieve *high-quality results*, and the availability of *low-cost multiprocessors*. Some of the tasks in the automatic design of integrated circuits which have been solved with parallel processing include the following: floor planning [2], circuit extraction [3, 4], circuit simulation [5, 6], logic simulation [7], and test generation/fault simulation [8]. The above results have demonstrated the wide variety of CAD applications that can be solved with parallel processing. However, it has also become very clear that parallel algorithm design is very difficult.

The research presented in this thesis is a small part of a larger project called the "HIPERCAD Project" (HIgh PERformance CAD environment) at the Center for Reliable and High-performance Computing at the University of Illinois. An outline of the project is shown in Figure 1.1. The goal of this project is to develop parallel algorithms for solving each of the tasks in the design and testing of integrated circuits. The tasks in bold type are the subject of this thesis.

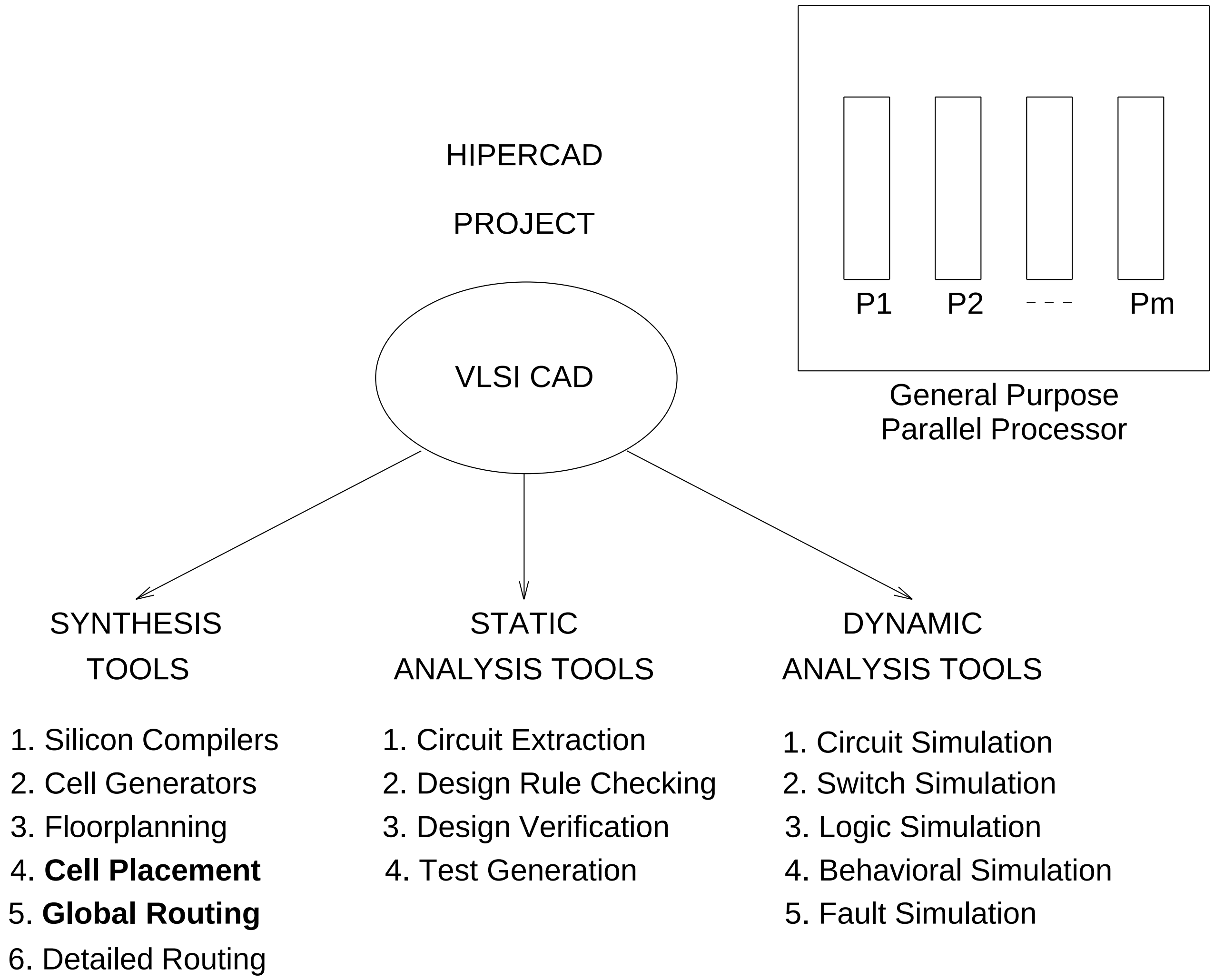


Figure 1.1. The Hipercad Project overview

1.2. Parallel Processing Architectures

There are many considerations in the development of a parallel algorithm for a given application. One of the most important factors is the type of parallel architecture that is to be used. MIMD (Multiple Instruction-Multiple Data) architectures allow each processor to be executing different instruction streams (IS), independent of what the other processors are executing. The MIMD architecture is shown in Figure 1.2. Two subclasses of MIMD multiprocessors are the distributed-memory and the shared-memory types. Distributed-memory multiprocessors can be scaled relatively easily to large numbers of processors; however, they suffer a substantial loss in performance when processors must communicate or share data often. Figure 1.3 shows a typical structure for a distributed-memory multiprocessor. Shared-memory multiprocessors can handle the sharing of data and communication among processors very efficiently since the processors can share real memory; however, since the processors normally share a common bus or interconnection network, only a limited number of processors can be attached and used. Figure 1.4 shows a typical structure for a shared-memory multiprocessor.

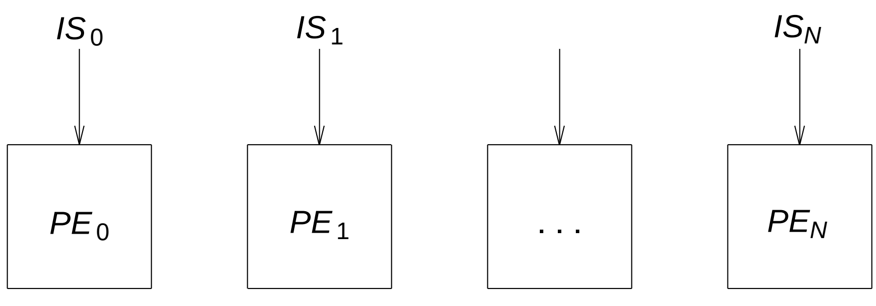


Figure 1.2. Multiple Instruction-Multiple Data multiprocessors

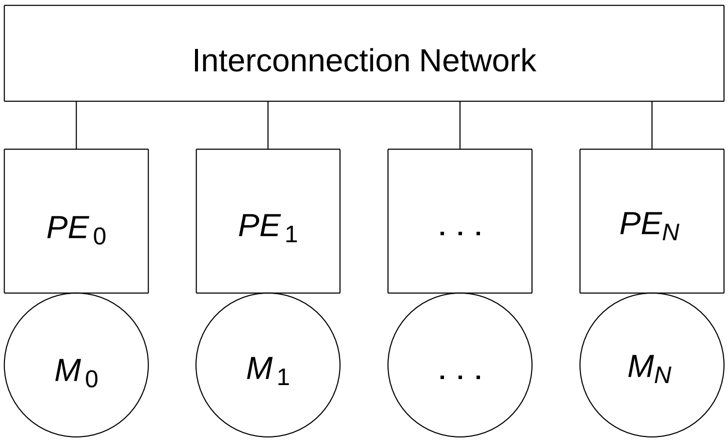


Figure 1.3. Distributed-memory multiprocessor

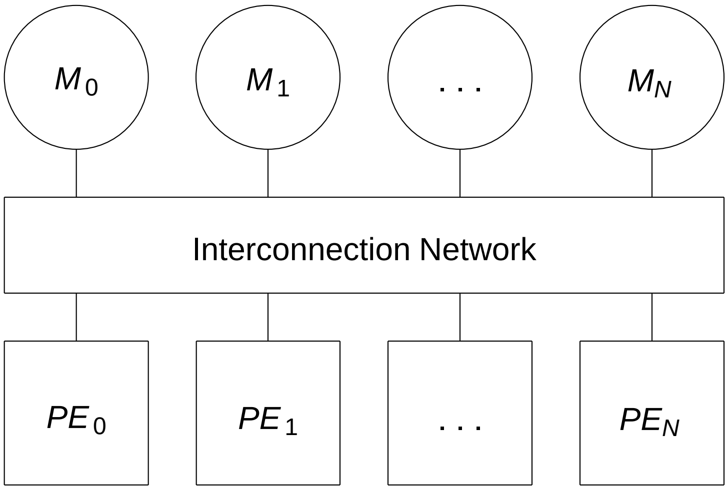


Figure 1.4. Shared-memory multiprocessor

SIMD (Single Instruction-Multiple Data stream) architectures operate on the premise that the same instructions can be executed by all of the processors on different data; however, not all applications can be partitioned this way. The SIMD architecture is shown in Figure 1.5. High-performance vector processors are best suited for the small subset of design automation problems that can be modeled as vector-matrix operations.

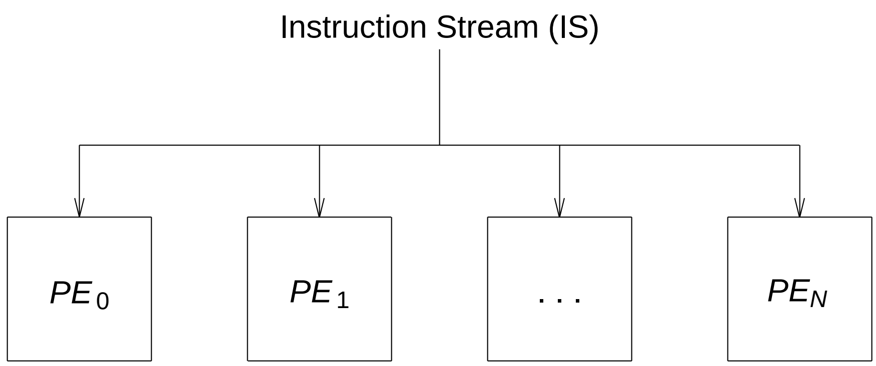


Figure 1.5. Single Instruction-Multiple Data multiprocessors

Special purpose hardware accelerators have been developed as well [9]; however, these can be very expensive and if designed for a particular algorithm, can be rendered obsolete when better algorithms are developed. Thus, the methods for partitioning the tasks and the data of the application are dependent not only on the problem, but also on the type of parallel architecture intended and available for use.

Throughout the next few chapters, we will attempt to describe our approach to solving two VLSI CAD problems on a shared-memory multiprocessor. The first problem is global routing, and the second problem is simultaneous placement and routing. Throughout the rest of the thesis, our solutions to these problems will be discussed.

1.3. Thesis Outline

Chapter 2 of the thesis discusses the problems of the placement of cells and the global routing of nets in a row-based design methodology. The problem definitions and the models upon which cell placement and global routing are based are presented. Next, a brief review of some of the previous work in the areas of both uniprocessor algorithms and parallel algorithms is discussed.

Chapter 3 is devoted to a thorough discussion of our new parallel algorithm for global routing. Specific aspects of the global routing model and its relationship to the generation of the solution are presented. Alternative methods of decomposing the routing problem are described and evaluated. A description of the parallel decomposition of each method is provided, along with mathematical models of the parallel algorithm complexity. Finally, empirical results comparing the alternative methods for varying levels of parallelism are presented.

Chapter 4 contains a detailed discussion of our new parallel algorithm for placement and routing. Many specific methods employed throughout the algorithm for the achievement of simultaneous placement and routing are described, as are the types of parallelisms provided in the algorithm. Empirical results are presented in order to contrast the various methods, as well as to evaluate the algorithm and its inherent parallelism.

Finally, in Chapter 5 we summarize our contributions and discuss areas of future research.

CHAPTER 2.

CELL PLACEMENT AND GLOBAL ROUTING PROBLEMS

2.1. Introduction

The *cell placement* problem involves placing a set of cells or gates on a VLSI layout, given a netlist which provides the connectivity between each cell and a library containing layout information for each type of cell. This layout information includes the width and height of the cell, the location of each pin, the presence of equivalent (internally connected) pins, and the possible presence of feedthrough paths within the cell. The primary goal of cell placement is to determine the best location of each cell so as to minimize the total area of the layout and the length of the nets connecting the cells together.

The task of *global routing* is to take a netlist, a list of pin positions, and a description of the available routing resources and determine the connections and macro paths for each net. The net list is taken from the circuit/network description and the pin positions; routing resource information is derived from a placement of the cells in the circuit as generated by any high-quality placement algorithm.

2.2. Row-Based Layouts

In this thesis, we are primarily focusing our attention on row-based layouts. Some examples of row-based layouts include *gate array*, *standard cell*, and *sea-of-gates* design styles. Figure 2.1 shows what a typical *gate array* layout would look like. The layout is comprised of a two-dimensional array of *basic cells*, laid out in rows which are separated by routing areas called channels. The entire configuration is surrounded by a ring of pads for connections off-chip. Basic cells contain isolated transistors and must be "programmed" with connections in different layers of metal. By programming and connecting one or more basic cells together, all of the basic logic gates (e.g. AND, NOR, NOT) and flip-flops can be created. To reduce the fabrication time and cost per new design, wafers of gate array chips are fabricated in large amounts until the point of programming and connecting the basic cells is reached. This means that the locations of the basic cells and the height of the channels are fixed. Each new design will then

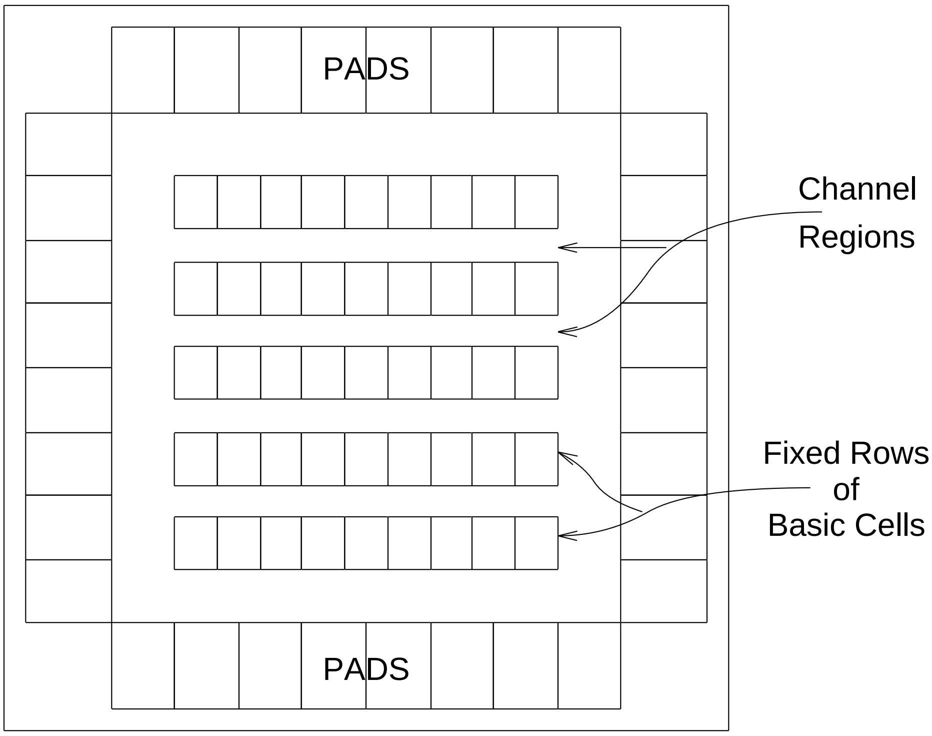


Figure 2.1. Example of a gate array design

require only a few fabrication steps on the prefabricated wafers, and can be completed in much less time.

Although the fabrication time is much less, there are some drawbacks to gate array layout. There is an absolute upper bound on the number of basic cells available and thus the number of possible gates is limited. In addition, the fixed size of the channels can either restrict the routing of nets or cause much wasted chip area. Often, basic cell utilization is much less than 100%. For large quantity productions, the standard cell layout may be better.

A typical *standard cell* layout is shown in Figure 2.2. Since there is no prefabrication of the wafers, standard cell layouts can have rows of variable height, variable length rows, and cells of variable width, depending on the requirements of the design. The overall utilization is much higher; however, the fabrication time is much greater than that of gate array designs. Since the layout area can be better utilized, the benefit of lower cost per chip for large quantities may offset the disadvantage of extra fabrication time.

Sea-of-gates designs are very similar to gate arrays. The primary difference is that there are no predefined areas for routing. Instead, it is assumed that an extra layer of metal can be used for over-the-cell connections. The number of basic cells is much higher than that of gate array, but the fabrication is more costly since more metal layers are necessary and the layout is more difficult.

2.3. Uniprocessor Cell Placement Algorithms

Most cell placement methods can be divided into two classes: *constructive* and *iterative* [10, 11]. Constructive methods determine the next cell's position based on the

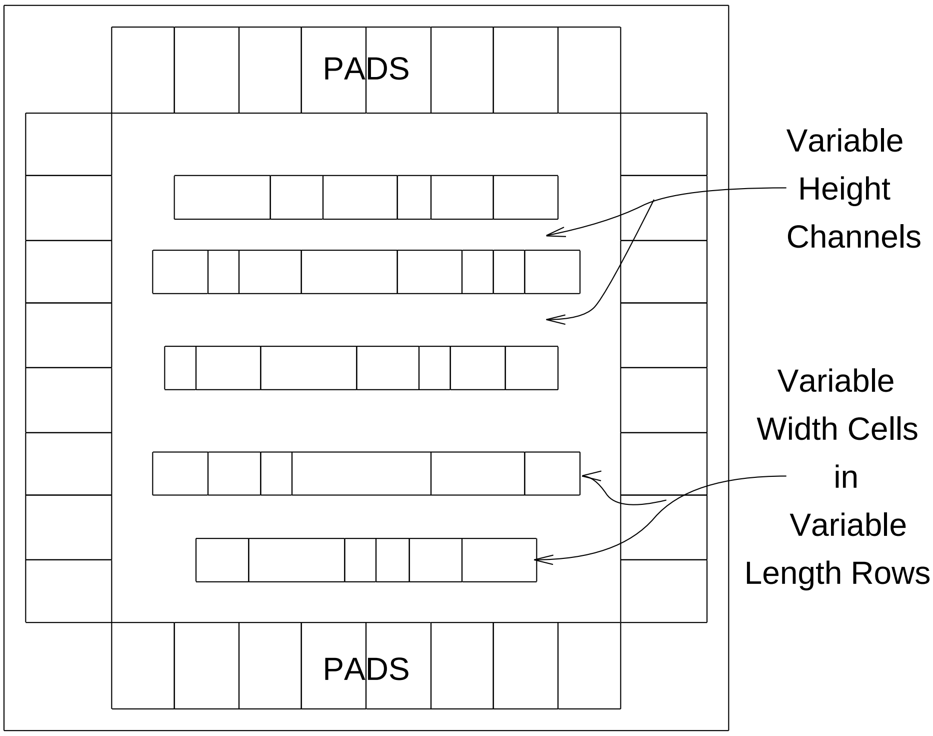


Figure 2.2. Example of a standard cell design

locations of the cells that have previously been placed. Specific examples of constructive placement methods include (1) cluster growth [12], (2) partitioning of components [13-16], (3) global placement by quadratic assignment or convex function optimization [17, 18], and (4) artificial intelligence planning [19]. Iterative methods attempt to alter a complete placement of the cells to attain any amount of improvement in the placement. Specific examples of iterative improvement placement include (1) successive overrelaxation [20], (2) simulated annealing [21-23], (3) simulated sintering [24], and (4) simulated evolution [25, 26].

Each of the above heuristics depend on the cost function employed to measure the acceptability of a current placement. Since the twofold goal of cell placement is to minimize the placement area while insuring the routability of the layout, cost functions have examined various criteria such as estimated wire length and cell congestion. One simple method for estimating the wire length is to measure the half-perimeter of a box which bounds the pins of a given net. Figure 2.3 graphically shows how the bounding box measure would be calculated. A more computationally intensive measure is to calculate the wire length of the minimal Steiner tree. One way to measure cell congestion is to calculate the number of nets that connect separate partitions of the set of cells. The goal is then to minimize the number of nets cut by a line separating the partitions. Figure 2.4 shows the high- and low-cost configurations for a small example circuit.

2.4. Parallel Cell Placement Algorithms

The majority of the research work on cell placement has been focused on developing nonparallel algorithms. These algorithms were discussed in the previous section.

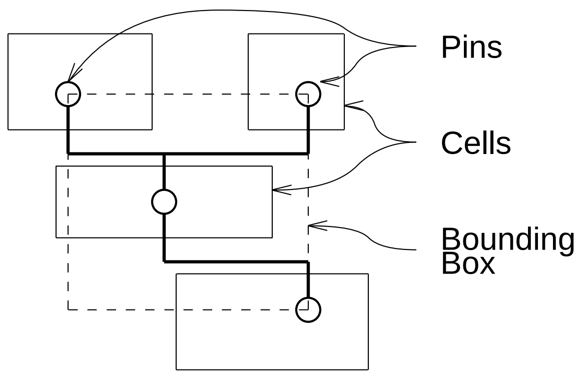


Figure 2.3. Wire length estimation by bounding box

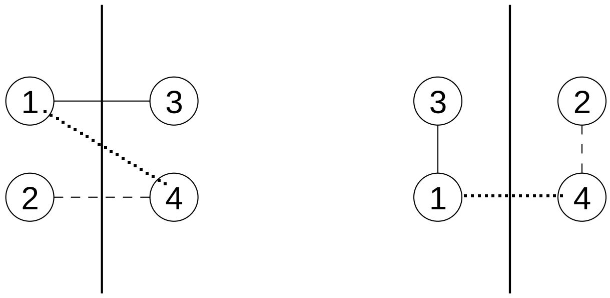


Figure 2.4. Cell congestion estimation by net cut count

However, since the placement of standard cells in a large circuit can be very time consuming, researchers have been investigating the tradeoffs of various parallel algorithms for cell placement. This work in parallel cell placement can be classified based on the architecture of the target machine. Using a message-passing hypercube multiprocessor, Banerjee et al. [27, 28] developed a parallel simulated annealing algorithm. In this algorithm, the layout area is equally divided among the processor nodes, and cells are displaced or exchanged between pairs of nodes in parallel, subject to the cost function and the simulated annealing temperature scale. After a sequence of moves, the cell location changes are broadcast to all processors to maintain current cell positions. Ravikumar and Sastry [29] reported another hypercube multiprocessor standard cell placement algorithm applying a divide-and-conquer technique. Following an initial cell placement, all clusters of cells are placed optimally within each cluster (using enumeration methods) in parallel. The clusters are then modeled as single modules and a parallel iterative improvement algorithm is applied to the clusters. Finally, a sequence of perturbations is applied to cell combinations within the clusters and between pairs of clusters in parallel.

Shared-memory computers have also been used as target multiprocessor architectures for the cell placement problem. Casotto et al. [30] proposed a parallel simulated annealing algorithm for macro cell placement on a shared-memory multiprocessor. Steps are taken to reduce the amount of error caused by the parallelization of a sequential algorithm. The shared memory is especially useful to help reduce communication overhead when updating cell locations after a move. Kravitz and Rutenbar [31] presented an algorithm for standard cell placement on shared-memory multiprocessors based on parallel simulated annealing. Two methods for extracting parallelism were analyzed: parallel move decomposition and the application of serializable subsets of moves in parallel. A serializable set of moves is any set of sequential moves which if executed in parallel would produce the same result. Move decomposition provided only a parallelism of three and speedups around two. Parallel moves were effective at the low-temperature ranges when the percentage of moves accepted was very small and a serializable set of moves was easier to attain.

Casotto and Sangiovanni-Vincentelli have proposed a parallel standard cell placement algorithm for the Connection Machine [32]. In their algorithm, sets of processing elements (PEs) are assigned to each cell and net, and are responsible for any calculations concerning those circuit elements. Unfortunately, the size of the machine and the number of PEs required for cells and nets limit the size of the standard cell circuits to around 8000 cells. Wong and Fiebrich [33] have developed a parallel algorithm for the Connection Machine using similar data structures [34].

Ueda et al. [35] have proposed a parallel cell placement algorithm for a two-dimensional processor array. The placement is performed by repeated pairwise exchanges of cells in parallel. The authors claim that the amount of interaction among the parallel exchanges reduces to almost nothing for large circuits. Finally, Kling and Banerjee [36] have implemented a simulated evolution-based standard cell placement algorithm on a network of workstations. The rows of cells are distributed among the available processors in a cyclic manner. The simulated evolution methodology is then applied to the set of cells in each processor with periodic broadcasts of the current cell locations.

2.5. Uniprocessor Global Routing Algorithms

Figure 2.5 shows a simple global routing problem for a chip with pads(P) and standard cells(C) in rows connected by nets(N). A global router must make choices among alternative paths for each net. In Figure 2.5, one such choice is between routing the net using the segment N and routing using the segment N‘. Furthermore, global routers

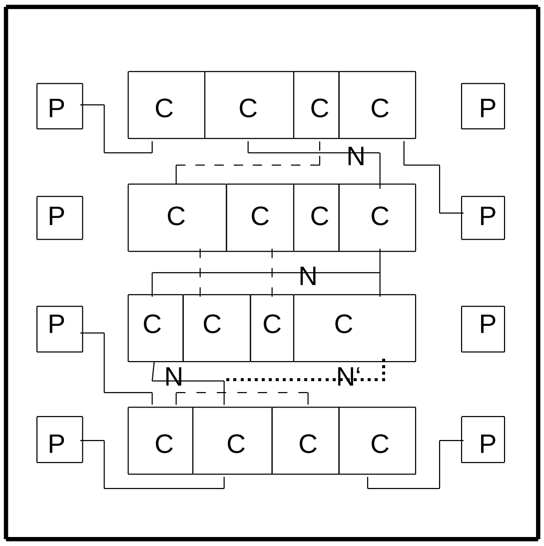


Figure 2.5. The global routing model

must determine how to connect wires from one row to another. These connections can be made by routing around the end of the row, utilizing terminals of a net on the top and bottom sides of a cell (equivalent pins), or making use of special feedthrough paths within or inserted between cells in the row.

Some criteria used to evaluate the quality of the routing include: total net length, total chip area, the number of tracks required to route the nets between the rows of cells (row-based routing), and the number of feedthroughs that had to be inserted between cells. For row-based layouts, i.e., standard cell or gate array, the output of the global router is typically used to set up the channels to be routed by a channel router.

Previous research in uniprocessor global routing can be divided basically into these categories: minimum spanning tree and other graph theory-based solutions [37-39], maze routing [40], physical analogies [41-43], and hierarchical routing [44-46]. Minimum spanning tree solutions model net connections as a spanning graph. The nodes of the graph represent the cells which the net connects and the goal is to try to reduce the graph to a tree while minimizing a cost function. In order to be effective, however, this method must handle the net ordering problem which occurs when nets are routed individually. Usually, the first nets selected for routing are given the best paths available. However, as more nets are routed, the constraints on unrouted nets build up so that the last nets routed have little chance of being routed well. A common method for dealing with the net ordering problem is to remove and then reroute sets of nets until no further improvements can be made.

Maze routing methods typically apply a line/wave expansion algorithm from a source pin to a destination pin. Since nets are usually expanded one at a time, the net ordering problem affects the quality of the results and must be addressed. Furthermore, nets must often be split into 2 pin subnets, providing a source and destination for the search algorithm, before routing can begin. This a priori splitting of nets can add unnecessary constraints to the problem and reduce the quality of the routing solution.

Physical analogy approaches have modeled the routing problem to fit into the framework of concepts such as simulated annealing, attractive and repulsive forces, and electromagnetic forces; however, the solutions generated must usually be transformed from the continuous domain to the discrete domain. Top-down and bottom-up hierarchical approaches have also been studied, usually in conjunction with one of the above approaches, to handle this complex problem.

2.6. Parallel Global Routing Algorithms

As with cell placement, the majority of the research work on global routing has been focused on the development of nonparallel algorithms; however, there have been a few projects which have utilized parallel approaches to the problem. One approach was to develop a maze routing algorithm suitable for a special purpose hardware routing machine, made up of a 2-D array of microprocessors [47]. Similarly, a maze router was implemented on the AAP-1 2-D array processor [48]. Two other algorithms for maze routing have been developed, specifically for the hypercube multiprocessor [49, 50]. A different approach, developed by Rose for shared-memory multiprocessors [51], determines the best of all possible two-bend routes for each two-pin subnet of each net.

Along with the problem of net order dependence, some of these parallel routing approaches suffer from routing quality degradation. As the number of processes used to solve the problem is altered, the quality of the final result can change dramatically.

This is because processes must assume that the current state information contained within themselves is accurate. However, some processes may be changing important state information that may not be immediately reflected in other processes. As the number of processes increases, the state information may become less accurate.

It is very important, then, to partition the tasks to be solved in parallel in such a way as to minimize the interaction among the tasks being solved simultaneously. Hierarchical methods can be used very effectively to partition a problem into independent subtasks, provided the partitioning is done carefully. Since hierarchical routing methods not only route all nets simultaneously without occurrence of routing degradation with parallelism, but also handle large and complex routing problems, we have chosen to develop a parallel top-down hierarchical router [52]. Our parallel, hierarchical routing algorithm will be discussed in the following chapter.

2.7. Combined Placement and Routing Algorithms

We have been discussing placement algorithms that use an approximation of net routings during evaluation stages. Since global routing and cell placement are both NP-hard problems, most design methodologies have separated the two problems to reduce the solution complexity. Often, lower quality placements and routings result from the separation of the two problems. A placement algorithm that takes into account the information from a global routing of the nets throughout the algorithm can better anticipate routing congestion and adjust the placement immediately. A routing algorithm can perform far better if it guides the placement of cells as they are being placed to reduce routing demands in particular regions. Most placement programs measure the quality of a cell's location by finding an approximation of the net lengths, usually the half-perimeter of a box bounding all pins of the net; however, with simultaneous placement and routing, the goodness measure of a cell can be evaluated more accurately.

Noting the benefits of combining placement and routing, other researchers have begun to develop techniques which combine algorithms for the two problems to improve the final placement. Szepieniec [53] proposed a novel hierarchy-based integrated placement and routing algorithm. The algorithm depends on having the underlying layout arranged as a slicing layout. A slicing layout is obtained by repeatedly splitting a layout with alternating horizontal and vertical parallel lines. Shragowitz et al. [54] presented a placement and routing algorithm for use in the layout of sea-of-gates style chips. The layout is dynamically divided into slices as the solution proceeds from the left side to the right side of the layout. Dai and Kuh [55] proposed an algorithm for integrated floorplanning and global routing. This algorithm is especially designed for Building Block Layouts. Igusa et al. [56] developed another sea-of-gates based floorplanning/placement/routing system. In a hierarchical fashion, floorplanning is performed on the set of cells, followed by specific placement of the cells. Next, a sequence of global routing and placement adjustment steps are repeated until convergence is achieved.

Finally, Suaris and Kedem [57] proposed an algorithm for combined placement and routing of standard cells based on quadrisection (an extension of bisection). Again in a hierarchical fashion, the cells are placed, based on the terminal propagation of each net, followed by global routing to generate a spanning tree for the nets. The spanning tree information is then used to assist the terminal propagation. This sequence is repeated in a sequential manner.

2.8. Parallel Combined Placement and Routing Algorithm

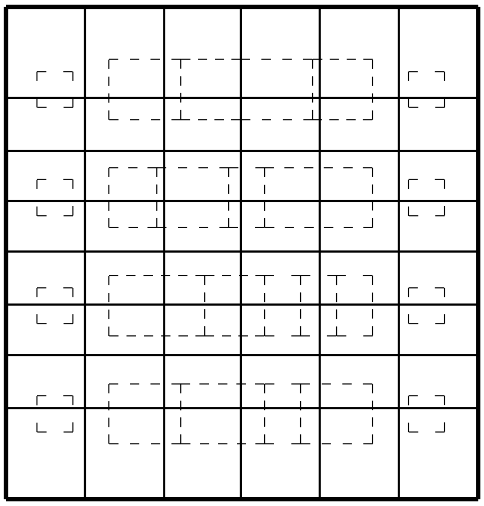
Unfortunately there has been little, if any, published work in the area of parallel algorithms for simultaneous placement and routing. As we have seen, parallel processing can be used effectively in placement algorithms to reduce the overall runtimes. We have seen that there are many benefits to combining the tasks of cell placement and global routing while taking advantage of the interaction between the two. In Chapter 4, we will present a new parallel algorithm for combined placement and routing which addresses these problems.

CHAPTER 3.

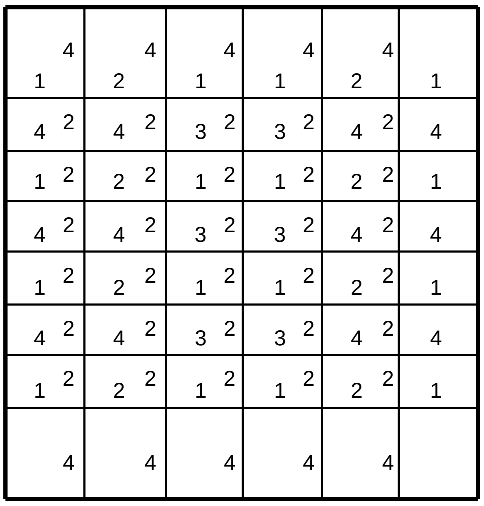
PARALLEL GLOBAL ROUTING

3.1. Global Routing Model

The global routing model we are using is similar to that of Burstein and Pelavin [44]. The entire layout area (including pads) is divided into a two-dimensional array of *routing blocks*. Each routing block is assigned routing capacity information for each of its four boundaries based on the physical dimensions of the routing block and the underlying layout. Figure 3.1 demonstrates how the routing block array and the routing capacity of



(a) Layout block setup



(b) Capacity determination

Figure 3.1. Routing block model

each block (Figure 3.1(b)) are derived from a given layout (Figure 3.1(a)). The dashed boxes represent the cells in rows and the pads along the edges of the layout. The dimensions of the routing block array are determined by the number of cell rows in the layout. The numbers along the grid lines in Figure 3.1(b) represent the wiring capacity along the vertical and horizontal edges of the routing block. The values given are based on the channel width, the number of built-in feedthroughs, and the actual size of the routing block.

In the routing capacity model, it is sufficient for each routing block to maintain capacity information for only two of its four shared edges (for example, the top and right edges). Let us denote the vertical capacity for a routing block in row *r* and column *c* as $v_{r,c}$ (across the top edge), and the horizontal capacity as $h_{r,c}$ (across the right edge). Let *L*, *R*, *T*, and *B* be the locations of the left, right, top, and bottom edges (rows and columns) of the region to be solved. Let *X* and *Y* be the locations of the vertical (y) and horizontal (x) axes, respectively, of the two-by-two bin array. Let CAP_i, $i \in A,B,C,D$ represent the capacities of the four axis segments in clockwise order around the two-by-two bin array, as shown in Figure 3.2(a). Then,

$$CAP_A = \sum_{i=Y}^{T} min(h_{i,X-1}, h_{i,X}, h_{i,X+1})$$
$$CAP_B = \sum_{i=X}^{R} min(v_{Y-1,i}, v_{Y,i}, v_{Y+1,i})$$
$$CAP_C = \sum_{i=B}^{Y} min(h_{i,X-1}, h_{i,X}, h_{i,X+1})$$
$$CAP_D = \sum_{i=L}^{X} min(v_{Y-1,i}, v_{Y,i}, v_{Y+1,i}).$$

This scheme quickly estimates the capacity of the axes, with little chance of overestimating by concentrating on the regions closest to the axis. Cases in which the routing

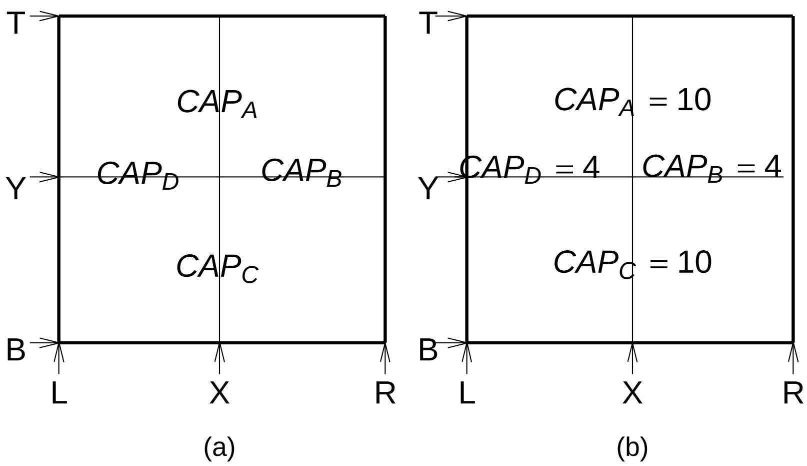


Figure 3.2. (a) Axes capacities of 2x2 bin array (b) Example

block capacities are nonuniform near an axis are handled as well. Figure 3.2(b) illustrates the capacity estimation for the example in Figure 3.1.

At the start of each level of the hierarchical decomposition, the current set of routing blocks is divided into four regions or *bins*, forming a two-by-two bin array. During each stage of the decomposition, these bins are further divided into smaller regions until one of the dimensions of the bin is equivalent to the size of a routing block.

Next, each net in the given problem is classified as one of 15 net types, based on the presence of pins in each of the four bins. Figure 3.3 shows the 11 net types consisting of two or more occupied bins, along with the set of all possible routings associated with each net type. The remaining four net types not shown in Figure 3.3 represent nets which have all pins in the same quadrant, and are unnecessary to include in the routing evaluation.

Each possible routing of the net types has been assigned a unique variable number to be used in solving a linear program (see Figure 3.3). Such a formulation was

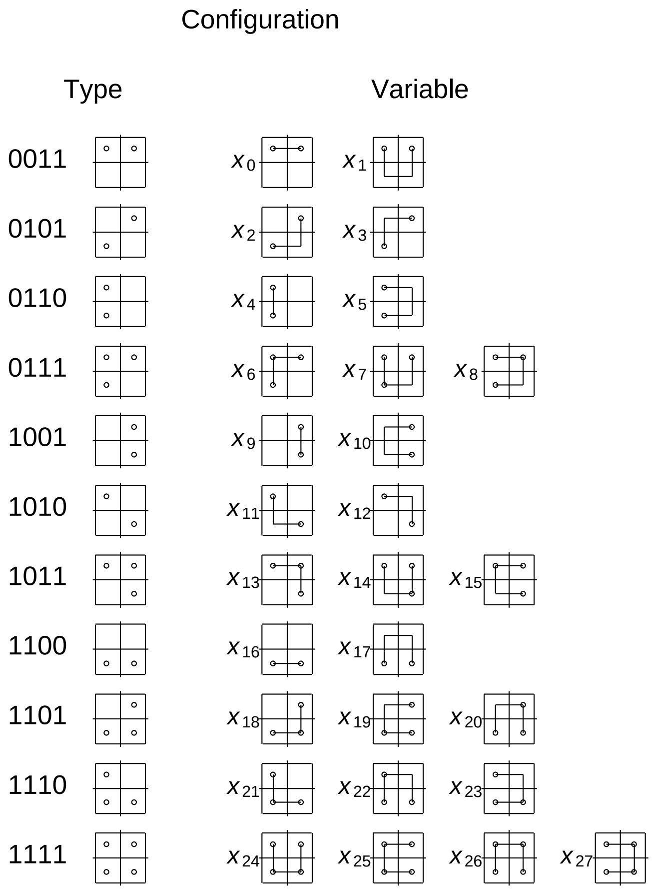


Figure 3.3. Net types and possible routings

proposed by Burstein and Pelavin [44]. We define a linear (integer) programming (LP) formulation of the problem to be

$$\text{For all } x,\ MAX\ (px)$$
$$\text{subject to } Ax \le a \text{ and } Bx = b,$$

such that x represents the variable space, p represents the objective function, A and a

represent the inequality constraints, and B and b represent any equality constraints. In our problem, the variables, x_i, $0 \le i \le 27$, represent each of the 28 possible net routings from Figure 3.3, and the set of 15 constraints is based on the available routing capacities and the types of nets being routed. Four of the constraints which limit the number of nets crossing between adjacent bins are as follows:

$$CAP_A \ge x_0+x_3+x_5+x_6+x_8+x_{10}+x_{12}+x_{13}+x_{15}+x_{17}+x_{19}+x_{20}+x_{22}+x_{23}+x_{25}+x_{26}+x_{27}$$

$$CAP_B \ge x_1+x_2+x_5+x_7+x_8+x_9+x_{12}+x_{13}+x_{14}+x_{17}+x_{18}+x_{20}+x_{22}+x_{23}+x_{24}+x_{26}+x_{27}$$

$$CAP_C \ge x_1+x_2+x_5+x_7+x_8+x_{10}+x_{11}+x_{14}+x_{15}+x_{16}+x_{18}+x_{19}+x_{21}+x_{23}+x_{24}+x_{25}+x_{27}$$

$$CAP_D \ge x_1+x_3+x_4+x_6+x_7+x_{10}+x_{11}+x_{14}+x_{15}+x_{17}+x_{19}+x_{20}+x_{21}+x_{22}+x_{24}+x_{25}+x_{26}$$

The remaining 11 constraints limit the variable values for each of the 11 net types and are as follows:

$$N_{0011} = x_0 + x_1$$

$$N_{0101} = x_2 + x_3$$

$$N_{0110} = x_4 + x_5$$

$$N_{0111} = x_6 + x_7 + x_8$$

$$N_{1001} = x_9 + x_{10}$$

$$N_{1010} = x_{11} + x_{12}$$

$$N_{1011} = x_{13} + x_{14} + x_{15}$$

$$N_{1100} = x_{16} + x_{17}$$

$$N_{1101} = x_{18} + x_{19} + x_{20}$$

$$N_{1110} = x_{21} + x_{22} + x_{23}$$

$$N_{1111} = x_{24} + x_{25} + x_{26} + x_{27}$$

where N_t is the total number of nets in each configuration t.

The objective function is designed to minimize the interconnection lengths of the nets by prioritizing the variables representing the shorter length connections more than those representing the longer ones. For example, in Figure 3.3, x_0 would have a higher weight than x_1, and x_4 would have a higher weight than x_5. The four variables representing the routing configurations of net type 15 are biased in the objective function toward the selection of the shortest length net. For instance, if the area represented by the four bins is wider than it is high, it is desirable to minimize the number of horizontal connections. Therefore, we would favor variables x_{24} and x_{26} over variables x_{25} and x_{27} by assigning them a higher weight.

The values of the variables x_i resulting from the solution of the linear program represent the number of nets routed in the particular pattern which the variable represents. After a solution to the LP is found, the nets must then be assigned to the appropriate configuration. The current implementation performs a greedy assignment of the nets.

3.1.1. Feedthrough insertion and channel width expansion

In row-based layout, feedthroughs must be inserted into the rows to make connections if no built-in feedthroughs or equivalent pins are available when connections must be made from row_i to row_{i+2} past row_{i+1}. The routing algorithm handles the problem through the simplex computations. After the problem has been set up, as long as sufficient routing facilities are available, a solution will be found, or else the simplex algorithm will terminate as having an infeasible initial problem. By analyzing the simplex state and the given routing problem, adjustments to certain capacities will provide a feasible initial problem for the simplex algorithm. Under certain simplex state conditions, these adjustments immediately generate a feasible initial problem. Otherwise, selected capacities are increased until a feasible problem is produced. Adjustments to CAP_A and CAP_C are equivalent to an increase in the channel width. Adjustments to CAP_B and CAP_D are equivalent to the insertion of feedthroughs in the row along the X-axis.

3.1.2. Hierarchical decomposition

As mentioned earlier, we are applying two-dimensional hierarchical decomposition methods to the global routing problem. At each stage of the hierarchy, we divide a larger problem into four smaller subproblems (divide and conquer). Deciding how to partition the subproblems so that they are independent of each other is very important. One critical decision involves the determination of net-crossing locations along the boundaries between the subproblems, and the determination of methods for locking these locations in place. We have investigated two approaches, which are discussed in the following sections.

3.1.2.1. Maximal boundary determination

The first strategy completely determines the net-crossing locations by recursively decomposing along the axes of interest down to the routing block level. This strategy is computationally more costly than the one to be discussed in the next section, but the advantage is that the complete boundary interface is determined hierarchically. Figure 3.4 shows the first steps in the decomposition for this strategy. The nodes of the graph represent a complete solution of a two-by-two routing problem, consisting of net analysis, linear program setup, linear program solution, and the assignment of nets to particular route types. The arcs of the graph represent dependencies from child nodes

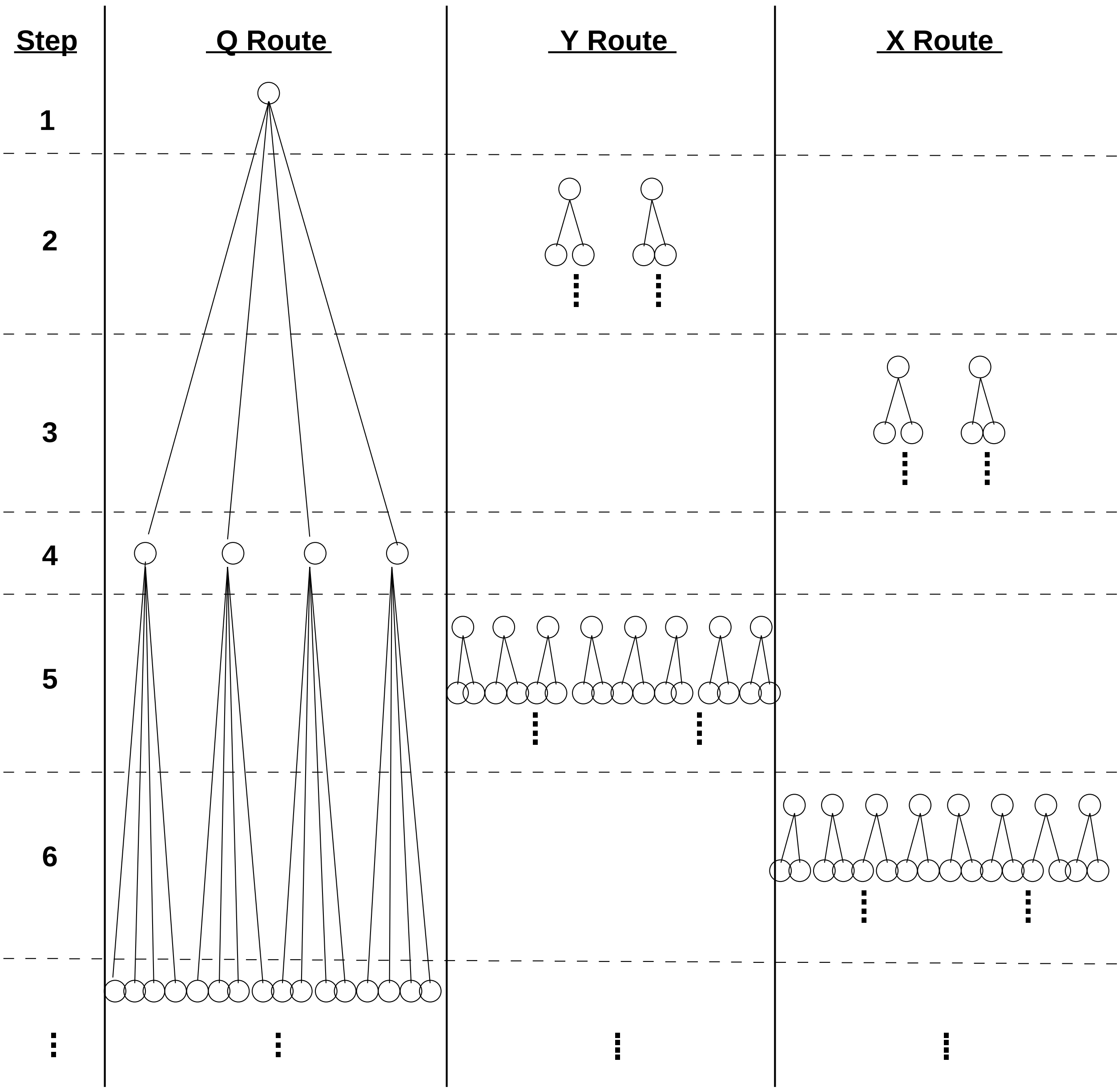


Figure 3.4. Maximal boundary determination

(below) as their parent node (above). The steps 0,4,7,... represent single two-by-two bin routings. The steps 2,3,5,6,... represent two-by-N routings of each axis from the previous step. In Step 1 and Step 2, the topmost two-by-two solution is followed first by the recursive two-by-N subdivision and solution of the X-axis down to the level of individual routing blocks, and second by the recursive two-by-N subdivision and solution of the Y-axis. After the completion of these steps, the net crossings have been completely determined and locked into place along both axes of the two-by-two bin problem, and the four subproblems for Step 3 are completely independent of each other. This sequence of steps is then recursively repeated until the size of the bin is equal to the size of the routing block, and the net crossings through all routing block edges have been determined. This strategy utilizes the maximum number of two-by-two routing solutions.

Figure 3.5 shows by example the first four decomposition steps of Figure 3.4. In this figure, the area of interest is highlighted by a box. The ellipses in the figure represent axis segments over which the routing has determined the set of crossing nets. Step 1 has decomposed the Y-axis into 2 parts, specifying the sets of nets crossing each half. Step 2 begins by decomposing the Y-axis, first into 4 parts, then into 8, and so on. Step 3 decomposes the X-axis in the same manner as Step 2. Step 4 begins with four independent routing problems since the net crossings over each border have been completely determined.

3.1.2.2. Minimal boundary determination

Figure 3.6(a) shows the first steps in the hierarchical decomposition for this second strategy. The topmost two-by-two problem is solved (Step 1), followed by quick heuristic approximations of the crossings of nets instead of the application of a two-by-N routing of each axis. The four subproblems are then completely independent in Step 2. These steps are repeated recursively until the routing block level (bin = routing block) is reached. This strategy utilizes the fewest possible two-by-two routing solutions for a

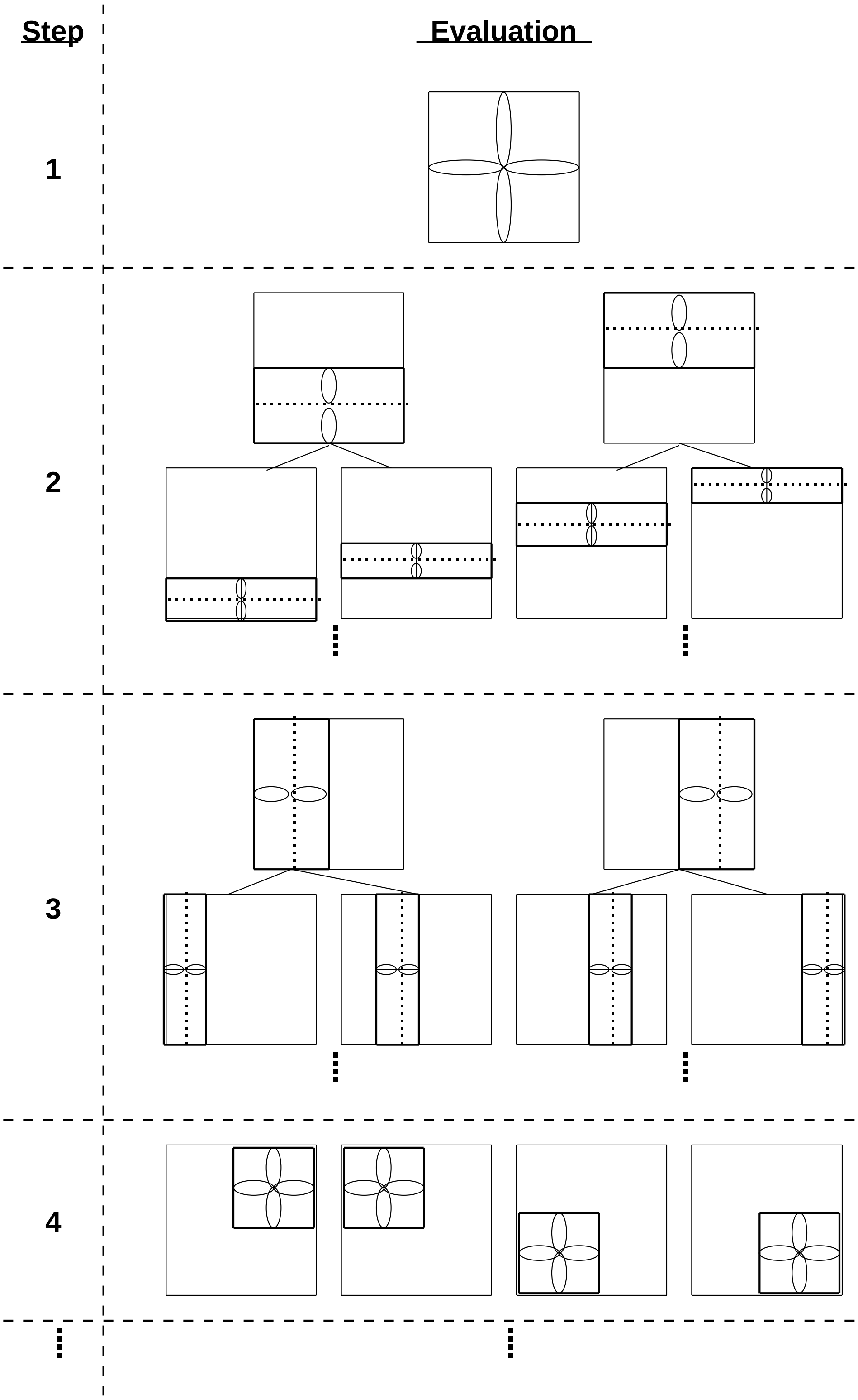
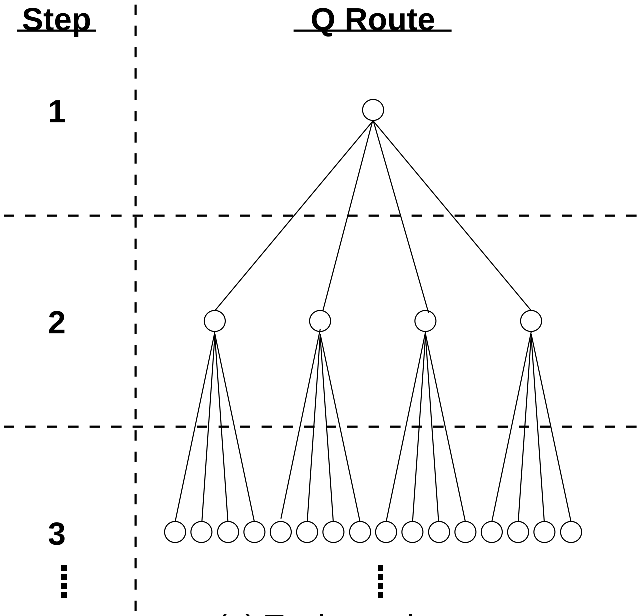
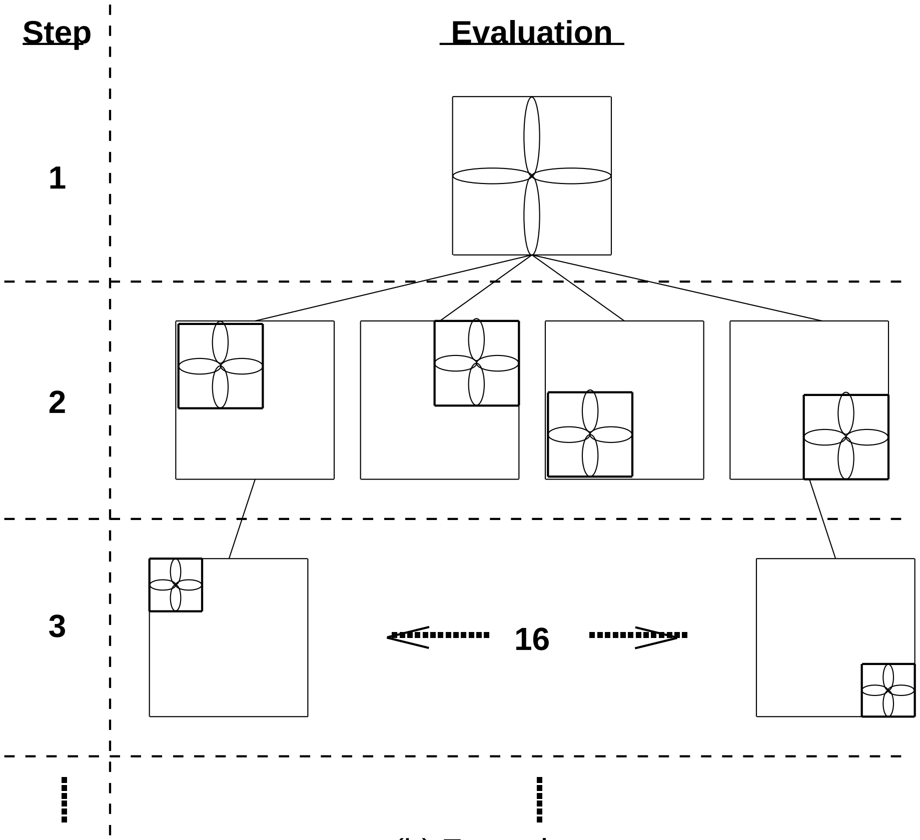


Figure 3.5. Example of maximal boundary determination



(a) Task graph



(b) Example

Figure 3.6. Minimal boundary determination

hierarchical routing. In Figure 3.6(b) we find an example of the decomposition steps in (a).

Even though the execution time for a single node of this strategy is greater than that of the previous strategy, the minimal determination of the boundary lines is faster than the previous method since the number of nodes in the graph (or solutions of two-by-two routing instances) is far less than that for the Maximal Boundary Determination strategy. However, the quality of the solution is often sacrificed for the sake of computational speed. The routing difficulty exists because without a costly complete analysis, it is extremely difficult to determine accurately the points along the boundaries at which each net should cross. Some approximations based on the pin locations of each net are used to estimate the crossing; however, if the boundaries are not well-predicted, the quality of the routing will be severely degraded, starting from the topmost two-by-two solution (Step 1). The Maximal strategy takes the extra effort to completely analyze the routing constraints along the subproblem boundaries in a hierarchical fashion.

3.2. Parallel Algorithm Overview

The term *granularity* has become accepted as a measure of the amount of work completed by a process before communicating with other processes in a parallel processing environment. Large-granularity applications would be characterized by long processing sequences interrupted by short, infrequent communication sequences. Fine-granularity applications would be characterized by very short processing sequences with a large amount of communication among processes. As will be shown, the tasks of our global routing algorithm can be considered coarse-grained, since the ratio of execution time to synchronization/communication time is very large.

3.2.1. Exploitation of coarse-grained parallelism

The parallel execution of a binary tree is a well-known paradigm. The hierarchical routing execution in our algorithm takes the form of a binary tree in which the nodes of the tree represent the LP setup, the LP solution, and the net assignments for a single two-by-two routing problem. Furthermore, each node of the tree that is currently being evaluated is completely independent of all other nodes on the same level. The local information for the current subproblem is derived from its parent node's data structures and global pin location information, which is strictly read-only. The solution of the routing subproblem causes the executing process to write the results to a global (shared) output data structure. However, since the tasks are spatially independent, there is no need for critical sections of code to lock out other processes as a process writes out its results.

After writing the results, the process creates two child routing subproblems. One child subproblem is assigned to the first idle and waiting process. The second child subproblem is then executed by the parent itself. If no processes are waiting, the parent will proceed to execute the first subproblem, followed by the second. The number of processes created and initially available for task solution is set equal to the number of processors available to the user.

The routing solution complexity and speedup under parallel execution for both decomposition strategies are estimated in the following sections.

3.2.1.1. Maximal boundary determination

Given R rows and C columns of routing blocks, the required number of evaluations to solve the vertical segments of all routing blocks in the maximal decomposition strategy is $(R-1)\times(C-1)$. Similarly, the required number of evaluations to solve the horizontal segments is $(C-1)\times(R-1)$. However, one vertical and one horizontal component is solved at each iteration, therefore, the total number of evaluations, N_{2x2}, is

$$N_{2x2} = (R-1)(C-1).$$

This expression has been verified through actual runs of the algorithm. The estimated execution time for one process is then

$$T_1 = T_{2x2}(R-1)(C-1),$$

where T_{2x2} is the average time to solve a single two-by-two routing problem as a linear function of the number of nets n. Since the estimated execution time T_P for P processes is equal to the time spent executing until all P processes are activated plus the time spent in full parallel execution, we have

$$T_P = (T_{2x2} + T_{sync})\left(2\log_2 P + \log_4 P - 2 + \frac{(R-1)(C-1) - \frac{7P-13}{3}}{P}\right),$$

where T_{sync} is an estimation of the time spent in synchronization. After simplifying the expression, we arrive at

$$T_P = (T_{2x2} + T_{sync})\left(\frac{(R-1)(C-1)}{P} - \frac{13P-13}{3P} + \frac{5}{2}\log_2 P\right).$$

The expected speedup is then

$$S_P = \frac{T_1}{T_P} = \frac{T_{2x2}}{(T_{2x2} + T_{sync})}\,\frac{6P(R-1)(C-1)}{6(R-1)(C-1) - 26P + 26 + 15P\log_2 P}.$$

3.2.1.2. Minimal boundary determination

Again, given R rows and C columns of routing blocks, $Z = min(R,C)$, the required number of node tasks to solve is

$$N_{2x2} \leq \sum_{i=0}^{\log_2 Z - 1} 4^i \approx \frac{Z^2 - 1}{3},$$

in which equality holds for cases in which $\log_2 Z$ is an integer. The estimated time for completion for one process is $N_{22} \times T_{22}$. Again, since the estimated execution time for P processes is equal to the time spent executing until all P processes are activated plus the time spent in full parallel execution, we have

$$T_P = (T_{2x2} + T_{sync})\left(\log_4 P + \frac{\frac{Z^2 - 1}{3} - \frac{P - 1}{3}}{P}\right).$$

After simplifying the expression, we arrive at

$$T_P = (T_{2x2} + T_{sync})\left(\frac{Z^2 - P}{3P} + \frac{1}{2}\log_2 P\right).$$

The expected speedup is then

$$S_P = \frac{T_1}{T_P} = \frac{T_{2x2}}{(T_{2x2} + T_{sync})} \frac{2P(Z^2 - 1)}{2Z^2 - 2P + 3P\log_2 P}.$$

Figure 3.7 provides a graphical look at the two equations for S_P assuming $\frac{T_{sync}}{T_{2x2}} = 0.1$. Included in the plot is an estimate of process efficiency (useful time/ total time) ranging from 0.95 for $P = 2$ to 0.6 for $P = 16$, based on measurement extrapolation, to model the effect of the task scheduling mechanism on the speedup. The current implementation provides dynamic task scheduling based on process availability. An idle process can acquire a task only immediately after another process generates it. To eliminate the need for barrier synchronization of the processes, a task queue is replaced by a process idle scoreboard. Thus, due to task granularity, there will be times when a process waits idle for a new task to be generated. As the number of processes increases, the process efficiency is expected to decrease.

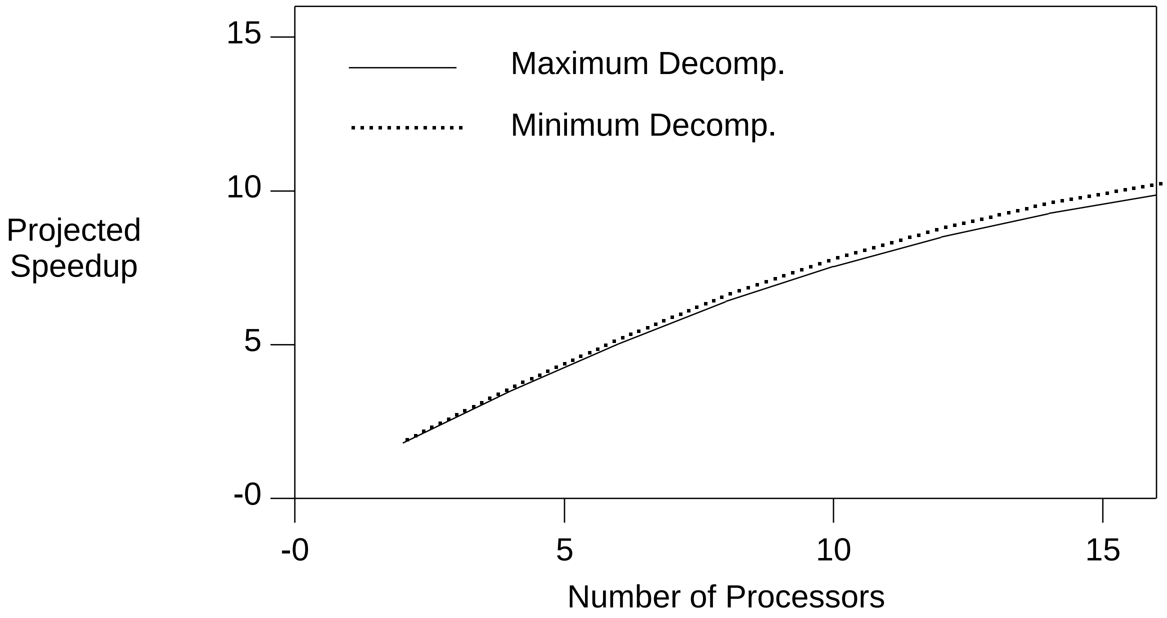


Figure 3.7. Plot of projected speedup vs. number of processes

3.2.2. Exploitation of fine-grained parallelism

There are three specific subtasks which can be executed in parallel at a fine-grained level. First, during the LP setup, the type for each net of the current two-by-two problem is determined. Since each net is independent, the nets may be divided among available processes and evaluated in parallel. Second, the exchange operations required to solve the linear/integer program may also be divided among available processes for parallel execution. Finally, the assignment of nets could be done in parallel, based on specific net types. Each of these areas of parallelism is orthogonal to each other.

However, since the amount of parallelism available at the task level (coarse-grained) is so great, the exploitation of parallelism at the fine-grained level would not provide significant improvement. Only during the startup phase of the execution tree will specific processes be idle. Figure 3.8 shows the percentage of the number of two-by-two solutions in the startup phase in relation to the total number of two-by-two solutions for routing problems with $R = C = Z$ and $P = 16$. As is clear from the figure, the part of the execution in large problems for which fine-grained parallelism can be useful is extremely small. Furthermore, parallelism of the simplex solution would not be effective since the average number of pivoting operations for solution has been measured to be less than 6. Therefore, we determined that is was unnecessary to evaluate these tasks in parallel at such a fine-grained level.

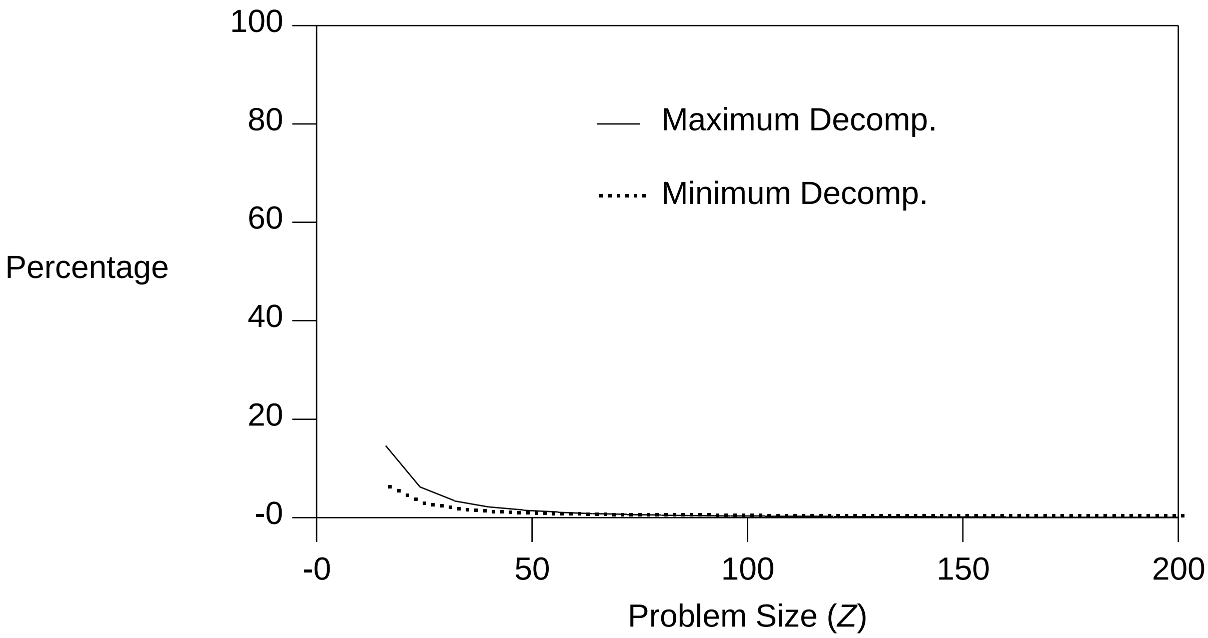


Figure 3.8. Percentage of tasks in startup phase

3.2.3. Task complexity

In the previous sections, we have discussed some of the basic elements of the two-by-two routing task. These are summarized as follows:

1. Evaluation of net types.
2. Setup of linear programming formulation.
3. Solution of linear/integer program.
4. Assignment of routing pattern to each net.
5. Subdivision of area for next level of hierarchy.
6. Repetition with child nodes.

LEMMA 1:

The complexity of a single solution of a two-by-two routing *task* is $O(n)$, where n is the number of nets.

Proof:

We will show that each subtask solution is $O(n)$ in the worst case. A circuit is assumed to have $p \leq kn$, where p is the number of pins or net terminals, k is a constant equal to the maximum number of pins per net, and n is the number of nets in the circuit. Thus p is $O(n)$.

1. To evaluate each net type requires a search for pins in the current region. This operation is $O(p) \leq O(n)$.

2. Each net is assigned to a specific linear program variable based on the characteristics of the net's pins. This subtask is $O(n)$.

3. The simplex solution of a linear program (with 28 variables, a fixed number independent of the problem size) can be shown to terminate in a finite number of pivots (steps) provided proper pivoting techniques are used. We are also applying cutting plane methods to convert the linear program solution into an integer solution

[58]. Measurements taken show the average number of pivots in the simplex solution to be less than 6.

4. The current implementation utilizes a straightforward assignment algorithm which runs in $O(n)$.

5. Subdivision of the current two-by-two region and setup for the next level of the decomposition can be done in constant time.

Thus, the complexity of a single task solution is $O(n)$.

QED

THEOREM 1:

The complexity of the parallel global routing algorithm is $O(nM)$, where M is the number of routing blocks.

Proof:

The total complexity of each strategy is the product of the task complexity and the total number of tasks (nodes). From Lemma 1 we know that the single task complexity is $O(n)$. For the worst-case Maximal Decomposition strategy, we determined in Section 3.2.1 that the number of tasks (N_{2X2}) is slightly less than the total number of routing blocks ($M = RC$). Thus, the complexity of the algorithm is $O(nM)$.

QED

3.2.4. Experimental results on 2X2 routing task complexity

For the following figures, the measurements were taken on the Encore Multimax, executing the Maximal Strategy on the Primary 1 benchmark. The iteration number refers to the task solution number in a depth-first trace of the execution graph. Figure 3.9 shows the time taken to set up the LP problem for each of the task solutions. The average time is 12.9 ms; the standard deviation is 1.2 ms. Figure 3.10 shows the time taken to solve the given LP problem for each task solution. The average time is 5.7 ms; the standard deviation is 5.3 ms. Figure 3.11 shows the execution time to assign the net types to a specific configuration for each task solution. The average time is 1.0 ms; the standard deviation is 0.6 ms. Figure 3.12 shows the total execution time (T_{2X2}) for each task solution. The average time is 19.6 ms; the standard deviation is 5.6 ms.

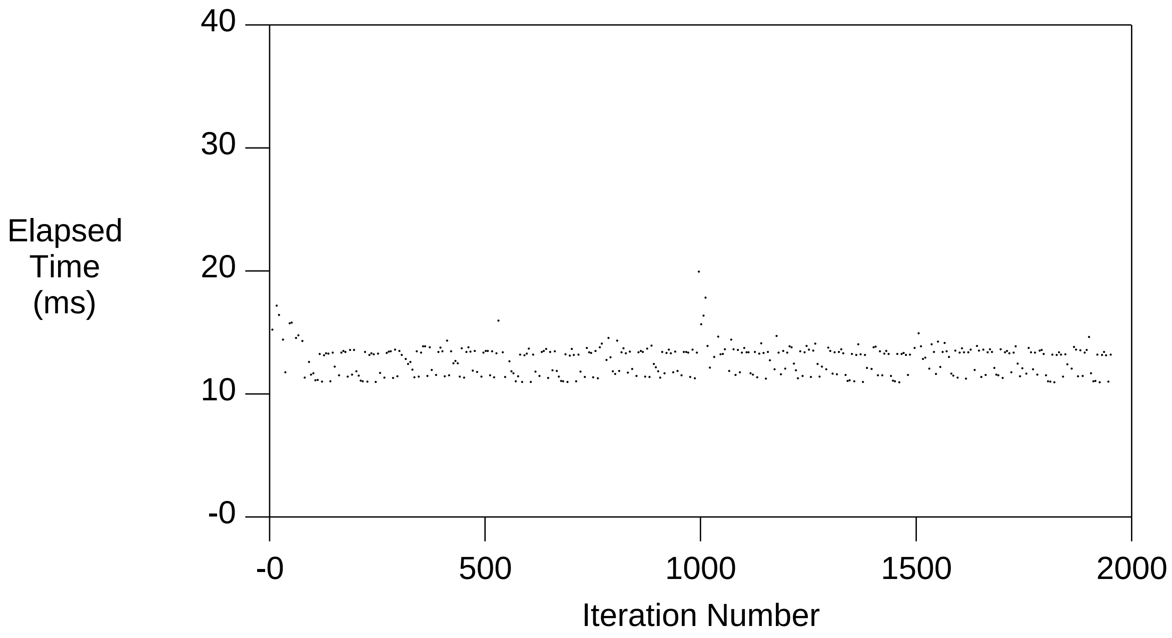


Figure 3.9. Net setup time vs. iteration number

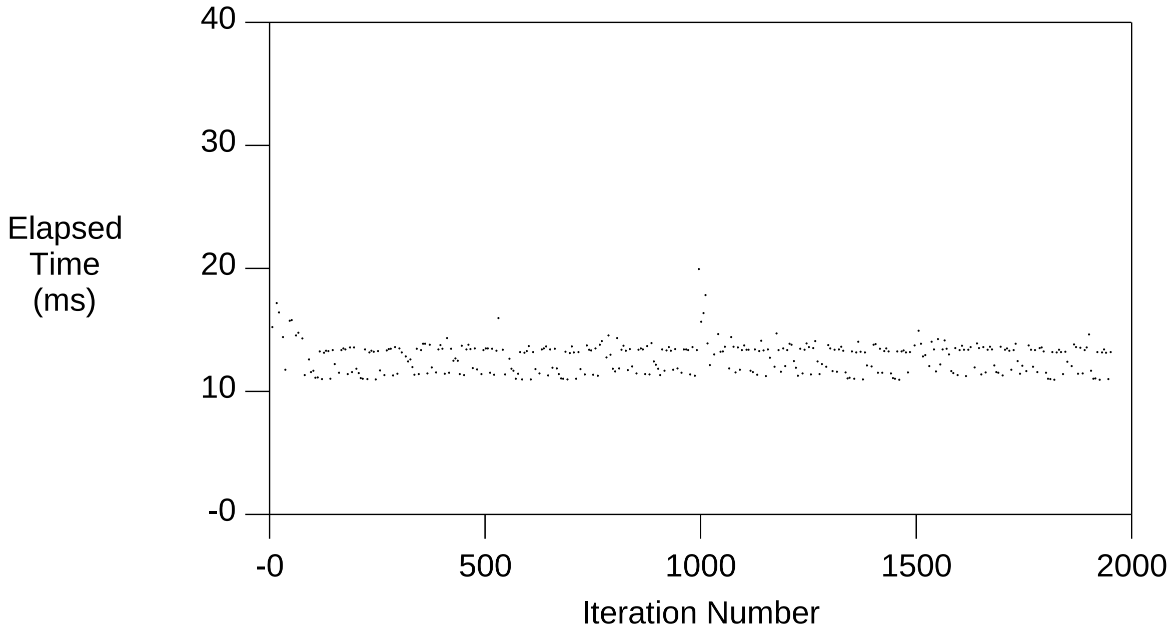


Figure 3.10. LP solution time vs. iteration number

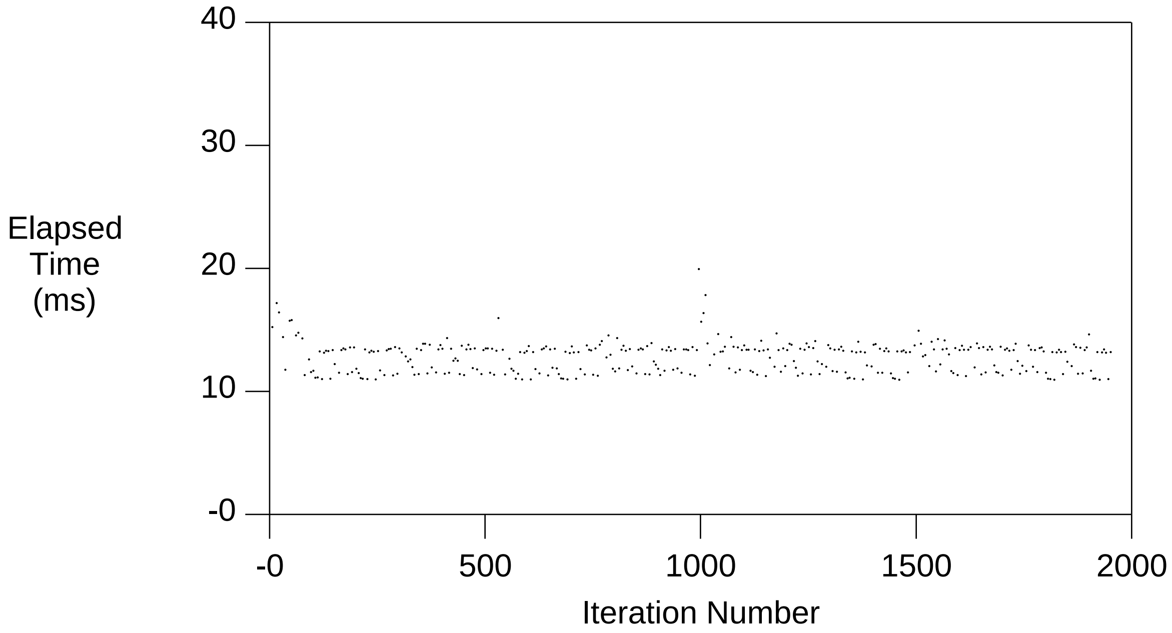


Figure 3.11. Net assignment time vs. iteration number

3.3. Implementation

The algorithm was implemented as *PHIGURE* (Parallel HIerarchical Global RoutEr) using approximately 5000 lines of C code on an eight-processor Encore Multimax 510 (shared-memory multiprocessor). Experiments were performed on a few of the placement and routing benchmarks from the MCNC Workshop on Placement and Routing, along with a number of other circuits. Testing was done for a single process, two processes, four processes, and eight processes.

A flow chart for the master (MP) and slave processes (SP) is shown in Figure 3.13. The master process begins by initializing shared and local data elements, including the *idle processor scoreboard*. The scoreboard is used to indicate the busy/idle state of each process and to pass pointers to new tasks for evaluation. Next, the master

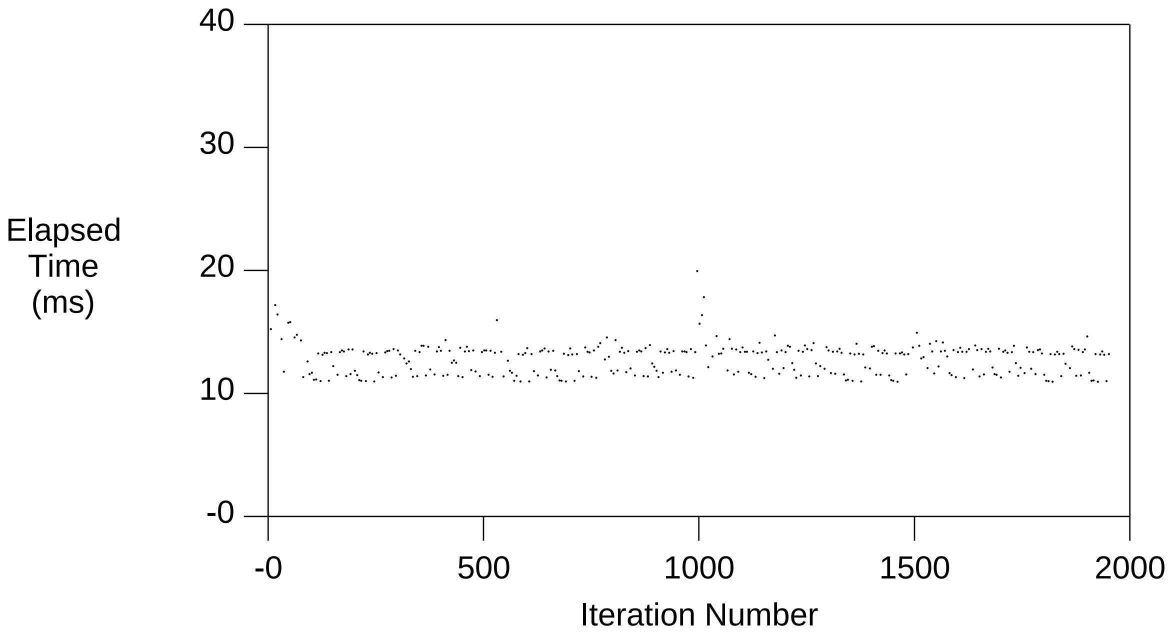


Figure 3.12. Total time vs. iteration number

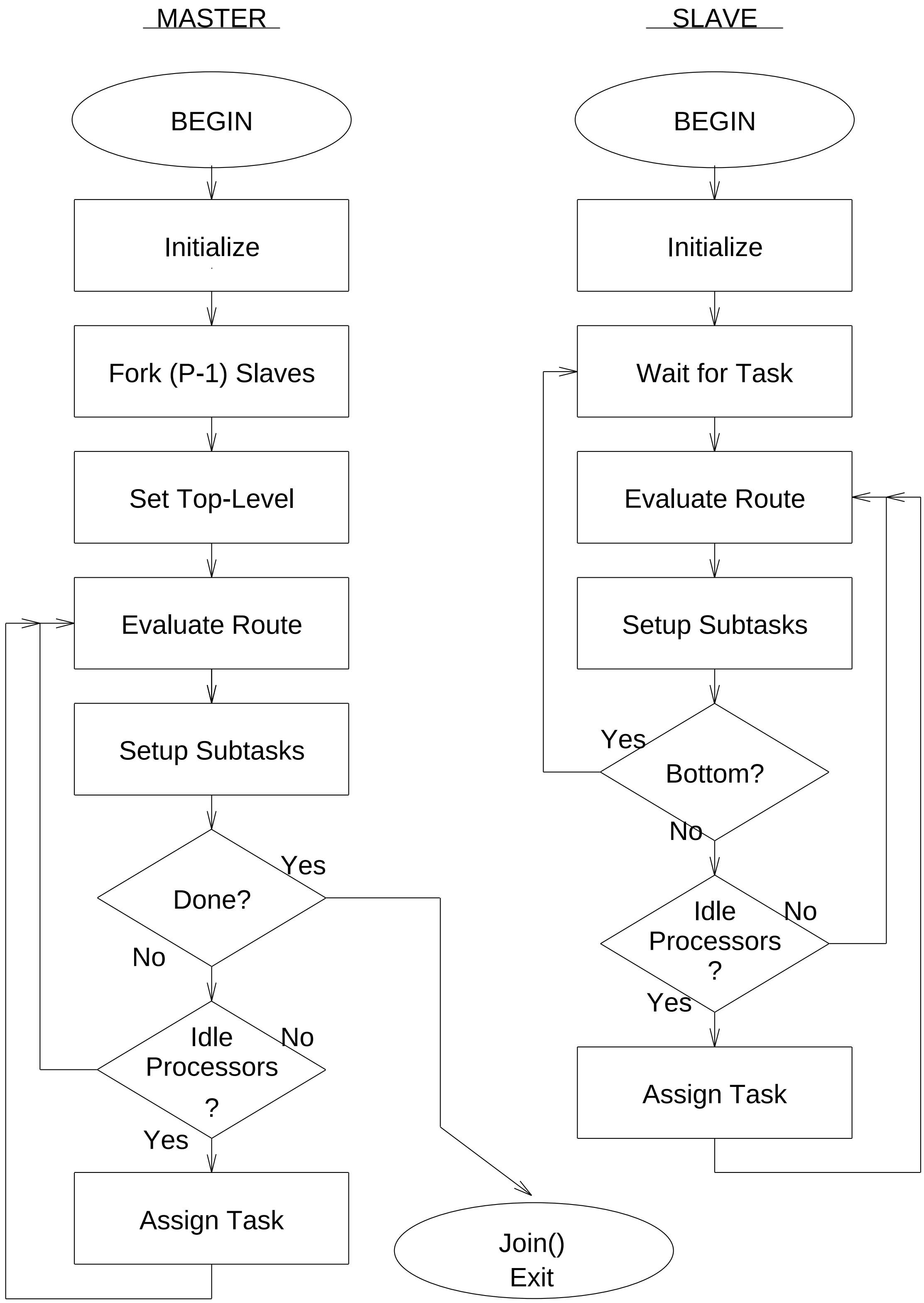


Figure 3.13. Parallel global routing flowchart

process forks off *NumProcesses* –1 processes. These processes receive copies of any fixed data (not to be changed) and share memory space for the data to be used by all processes. Following the solution of any routing tasks, new tasks are created (children of the node in the execution graph). The scoreboard is checked for idle processors. If there is an idle process, one of the new tasks is passed to it by way of the *shared task pool*; otherwise, the current process continues with the evaluation. The shared task pool is an array of pointers in shared-memory space. Processors enter a critical code section to place a task pointer in the pool or to remove a task from the pool.

While the SPs wait, the MP creates and solves the first routing task. After completion, idle SPs can begin to execute tasks in parallel. Processes which reach the bottom level of the hierarchical decomposition and are unable to create new subtasks set a flag on the idle processor scoreboard indicating their idle state and wait until a new task is provided. Finally, when the hierarchical routing is completed, the MP eliminates the SPs and writes the output to files.

3.4. Results

Table 3.1 compares the routing results of the algorithm to actual runs of the TimberWolf 5.4 global router (TW) [37] using the same placement and some of the recently

Table 3.1. Routing quality comparison

	Number of Trks					
Circuit	PHIGURE	TW5.4(Mea.)	TW5.4(Pub.)	UTMC	CP	LR
Primary1	210	163	166	177	190	262
Primary2	488	432	401	447	449	563

published results for the UTMC router (UT) [37], a router by Cong and Preas (CP) [38], and Locusroute (LR) [51]. This table shows that the algorithm performs well within the range of some recently published routers. Table 3.2 compares the uniprocessor run-times for the TimberWolf 5.4 router with those of the algorithm. These measurements were also taken on an Encore Multimax.

Table 3.3 shows the results for two of the Placement and Routing Workshop benchmark circuits and three other standard cell circuits. For each circuit, the table gives the number of tracks used, as estimated by the maximum channel density across the routing block edges, and the average execution times in seconds (real time, including process creation) for one, two, four, and eight processes using the Minimal and Maximal decomposition strategies. Cell placements for all of the circuits were performed by TimberWolf 5.4. As is clear from the table, there is no degradation in routing quality when going from a single process to many processes, and very good speedups were achieved (>6 for 8 processes). Since the hierarchical decomposition creates a large number of jobs after the first few steps, our algorithm is scalable for a large number of processes.

Table 3.2. Uniprocessor runtime comparison

	Runtime (s)	
Circuit	TimberWolf 5.4	PHIGURE
P1	221	153
P2	1326	565

Table 3.3. Parallel algorithm results

Circuit (Nets)	P	Min Decomp Trks	Min Decomp Time(s)	Min Decomp SpdUp	Max Decomp Trks	Max Decomp Time(s)	Max Decomp SpdUp
Primary1 (1185)	1	348	33	1.0	210	154	1.0
	2	348	17	1.9	210	81	1.9
	4	348	9	3.7	210	52	3.0
	8	348	6	5.5	210	35	4.4
Primary2 (3710)	1	817	187	1.0	488	565	1.0
	2	817	97	1.9	488	287	1.9
	4	817	52	3.6	488	163	3.5
	8	817	30	6.2	488	93	6.1
Circuit X1 (1979)	1	641	189	1.0	532	351	1.0
	2	641	92	2.0	532	174	2.0
	4	641	47	4.0	532	91	3.8
	8	641	29	6.5	532	55	6.4
Circuit X2 (3013)	1	709	254	1.0	596	389	1.0
	2	709	139	1.8	596	193	2.0
	4	709	74	3.4	596	103	3.8
	8	709	44	5.7	596	64	6.1
Circuit X3 (3258)	1	742	192	1.0	515	645	1.0
	2	742	97	1.9	515	325	2.0
	4	742	52	3.7	515	183	3.5
	8	742	30	6.4	515	97	6.6

3.5. Conclusions

In this chapter we have presented a new algorithm for parallel global routing. This algorithm applies hierarchical routing and decomposition techniques to create independent subproblems which can be evaluated in parallel. Even though parallelization of the original hierarchical algorithm might appear straightforward, we have demonstrated that one needs to decompose the problems in the parallel processing environment in such a way as to create less interaction among processes and therefore avoid contention. We have illustrated this through two approaches -- maximal and minimal decomposition. Results were presented which compare these two strategies for decomposing the routing problem and show that high-quality routings are attainable for one strategy. Most importantly, the routing quality is not degraded by decomposing in parallel.

The primary goal of this project was to be a stepping-stone for the work to be presented in the following chapter. There are numerous issues that could still be addressed; however, since the scope of this project was limited, we decided to proceed on with new work.

CHAPTER 4.

PARALLEL PLACEMENT AND ROUTING

4.1. Overview

In this chapter, we will discuss a parallel algorithm for placement and routing. The specifics of the algorithm presented refer to standard cell layouts, but, with slight alterations they can be applied also to other row-based layouts. Figure 4.1 shows the main steps of the algorithm. Each of these steps will be presented, followed by discussions of the complexity of the algorithm, the expected speedups, and the experiments to measure the effectiveness of the algorithm and the quality of the results.

Our goals in developing our placement and routing algorithm were to produce high-quality layouts, be able to interface the routing of the nets to the placement of the cells, limit the complexity of the algorithm, especially when considering large problem sizes, and be able to decompose the problem into a large number of independent tasks that can be executed in parallel. After considering the algorithms currently employed for placement and for routing, we proceeded to develop a combined placement and routing algorithm that utilizes two-dimensional hierarchical decomposition methods in both the placement of the cells and the routing of the nets. This approach especially avoids the complexity problems of many "flat" placement algorithms and provides many inherent parallelisms.

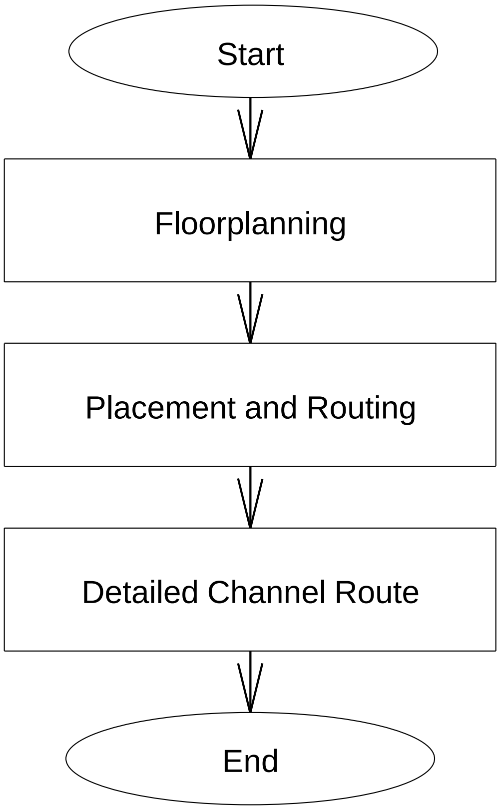


Figure 4.1. Overview of the placement and routing algorithm

4.2. Floorplanning Step

In the first step of the algorithm, floorplanning, the overall layout of the chip is evaluated. Based upon the total area of the cells that have been read from the input file, the aspect ratio (the ratio of the width of the chip to the height of the chip) can be user specified, and the default channel region height, the number of rows to contain the cells and their lengths are determined. The pads are also arranged around the periphery of the cell row region.

Next, the entire layout (including the pads) is divided up into a two-dimensional array of blocks, called *layout blocks*. In the cell row region, each layout block encompasses a portion of a row and the corresponding channel area above the row, as shown in Figure 4.2. Finally, estimates are made of the routing capacities for the routing areas of the layout (channels and the area between the pads and rows of cells) and assigned to the edges of the layout blocks.

4.3. Placement and Routing

The placement and routing step consists of a number of operations which are executed in a certain sequence at each level of the hierarchical decomposition. The operations are (1) the placement of the current set of cells, based on the quadrisection algorithm of Suaris and Kedem [16], (2) the routing of nets for the quadrisection placement,

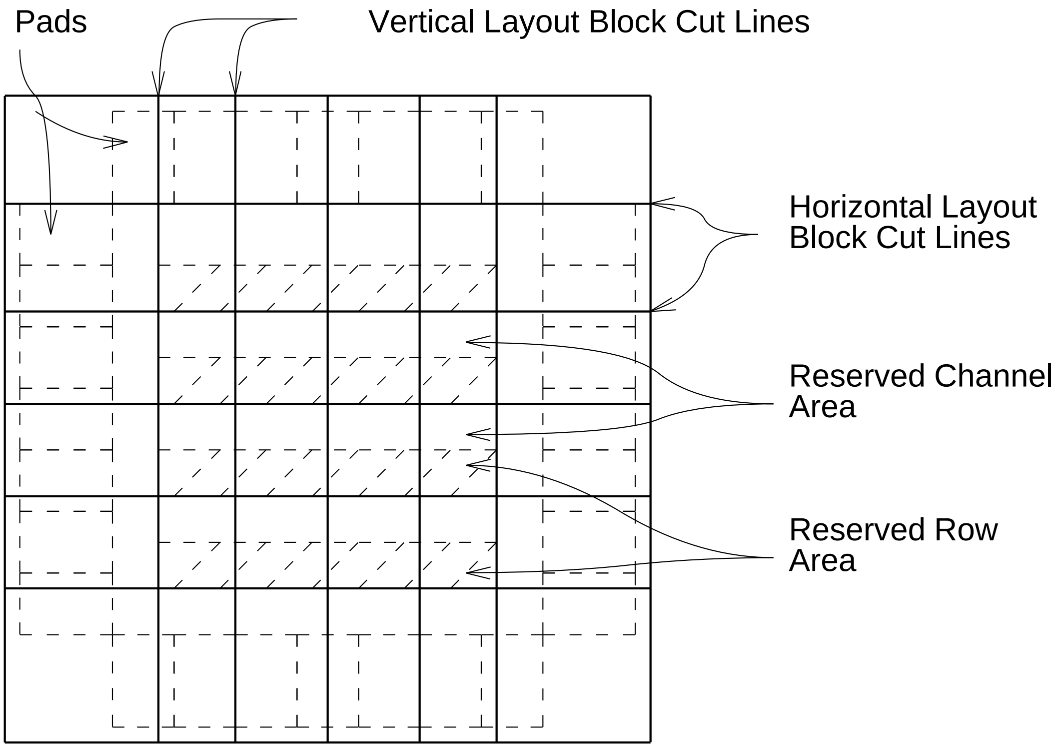


Figure 4.2. Determination of layout block array

similar to the algorithm described in Chapter 3, (3) the restricted global bisection of cells, and (4) the *two–by–N* routing of the nets in the bisection. Since we have tightly combined these placement and routing tasks, each operation intimately depends on the results of the other operations.

4.3.1. Quadrisection-based placement

Placement methods that are based on a partitioning strategy usually have a goal to minimize the number of nets crossing over the partition boundaries. The bisection (or min-cut) method partitions the layout (i.e., the circuit cells) into two groups, performing cell swaps between the groups until the number of nets crossing the single boundary is minimized. Figure 4.3 demonstrates the bisection partitioning algorithm in which the two groups of cells are connected across the boundary line. Let C_h, $h \in \{0,1\}$ be the set of

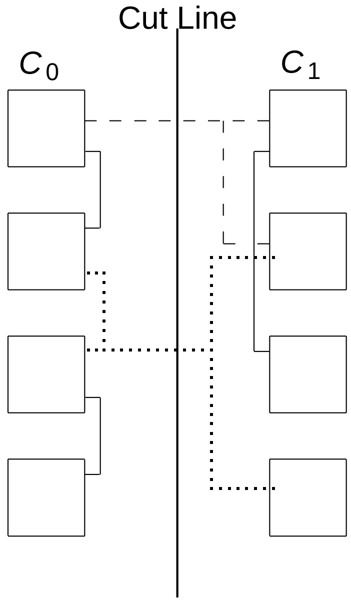


Figure 4.3. Min-cut partitioning

cells located in half h. Cells are repeatedly swapped between C_0 and C_1 to minimize the number of nets crossing the cut line while maintaining a balance in the cell area of both halves.

In the quadrisection method of Suaris and Kedem, the layout is partitioned into four groups (a two-by-two array of bins) instead of two groups, and cell movements occur among any of the four groups. An extension of the bisection heuristic for the selection of the cells is applied, which minimizes the net cuts over all four boundary segments of the two-by-two bin array through the movement of the selected cells. By approaching the layout problem in two dimensions instead of one, the authors have demonstrated results much better than those attained with the use of bisection placement. At each level of the quadrisection decomposition, a portion of the layout is selected and divided into four quadrants. Figure 4.4 shows the quadrisection algorithm in which the four groups of cells have net connections across the four boundaries. At level k in the decomposition, the entire layout has been divided up into a $2^k \times 2^k$ array of quadrisection regions (Figure 4.5). Notice that the cut lines used at level k become the boundary lines for the various quadrisection regions at level $k+1$.

In our quadrisection algorithm, we label the four quadrants as 0–3 and the four quadrant boundary segments as A–D, as shown in Figure 4.4. Let C_q, $q \in \{0,1,2,3\}$ be the set of cells located in quadrant q. Each net is assigned a residency flag for each quadrant, specified as 1 if a pin of the net is located in the quadrant and 0 if no pins are located in the quadrant. If a net connection from the area outside of the layout portion must enter into one of the quadrants, a *pseudo pin* is fixed in that quadrant for the net and is included in the residency vector. These pseudo pins are the result of previous

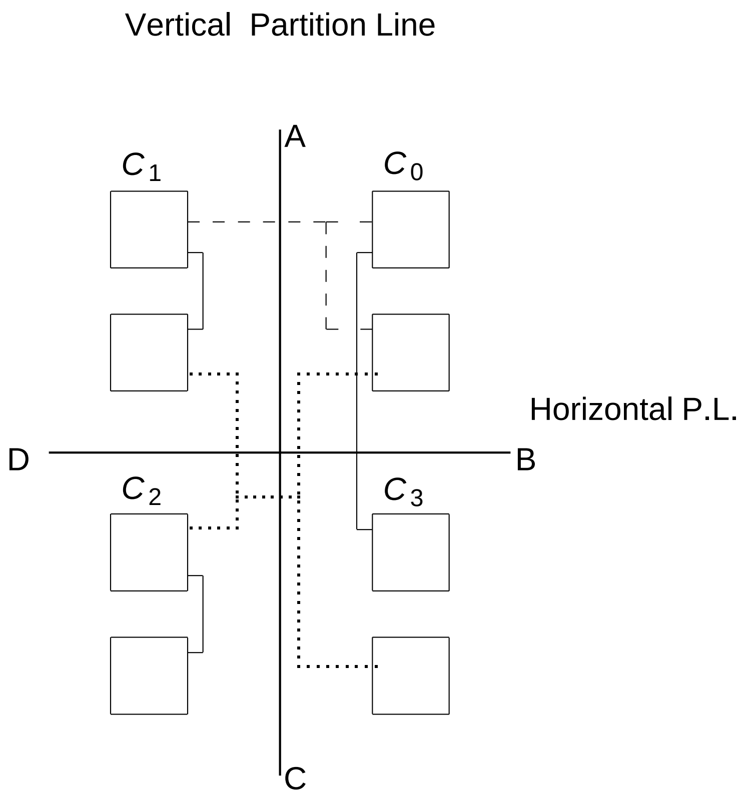


Figure 4.4. Quadrisection-based partitioning

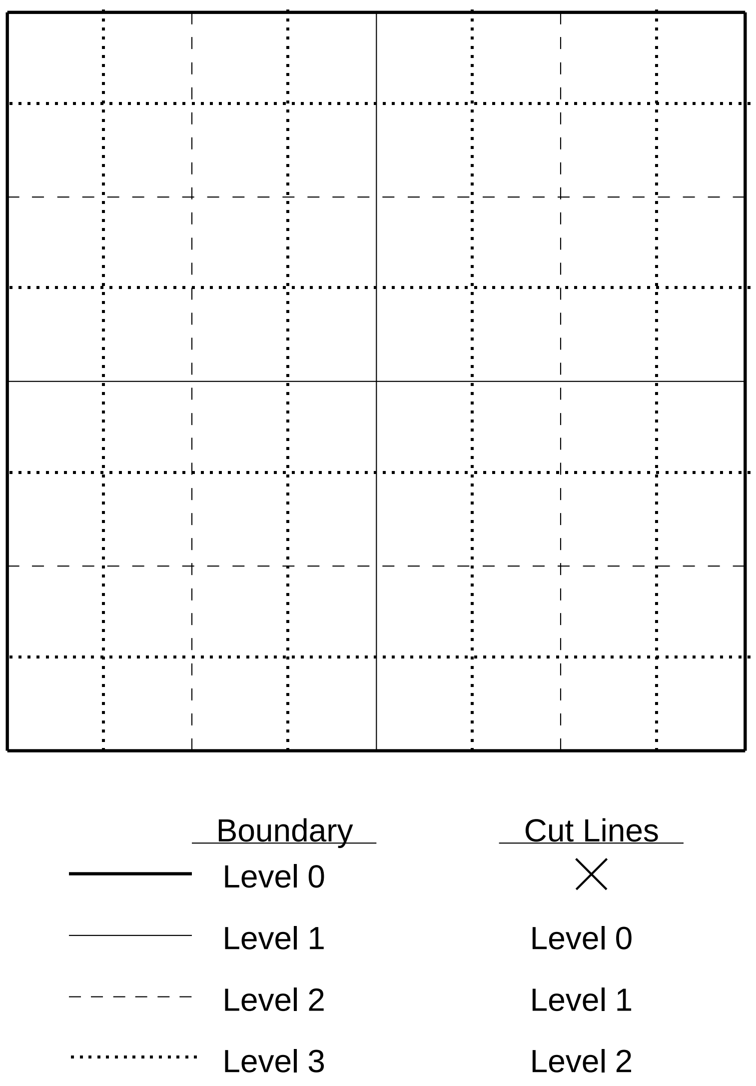


Figure 4.5. Partition and cut lines for different quadrisection levels

routing evaluations which have determined that certain nets cross into the layout portion through specific segments of the quadrisection outer boundary. The quadrant associated with the boundary segment receives the pseudo pin.

According to [16], each net can be associated with a cost which is calculated as a function of the net's residency vector for this set of quadrants. This cost function, which is shown in Figure 4.6, assumes that the shortest path is always available for connecting

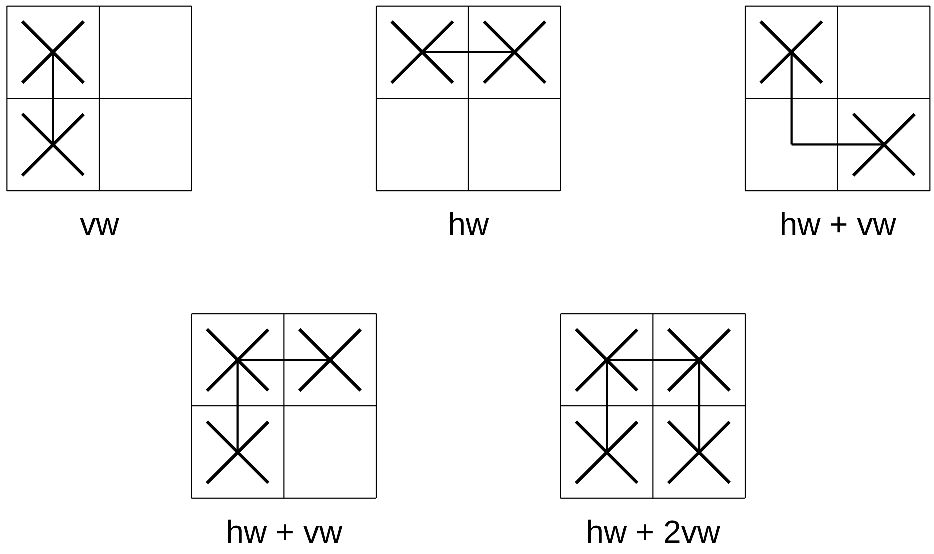


Figure 4.6. Simple quadrisection net cost function

the pins in the quadrants. The horizontal (hw) and vertical (vw) weights are used to account for differences in the costs for routing different directions, and are usually specified by the user.

We propose a better cost function which determines how each net is routed and calculates each net's cost based on the routing crossings of the four boundary segments A–D. Figure 4.7 shows the net cost of various routing alternatives for a few pin configurations. Similar to the simple cost function, the boundary crossing information for each net, which is determined after a global routing is performed, can be stored as a vector of residency flags, 0 if the net does not cross the boundary and 1 if the net does cross, for each boundary segment A–D. As in the standard cost function, the cost function can be evaluated in $O(1)$ time.

If a given cell c in quadrant q were to be moved to quadrant r, the nets associated with c may have to be rerouted to make the connections to the new pin in r.

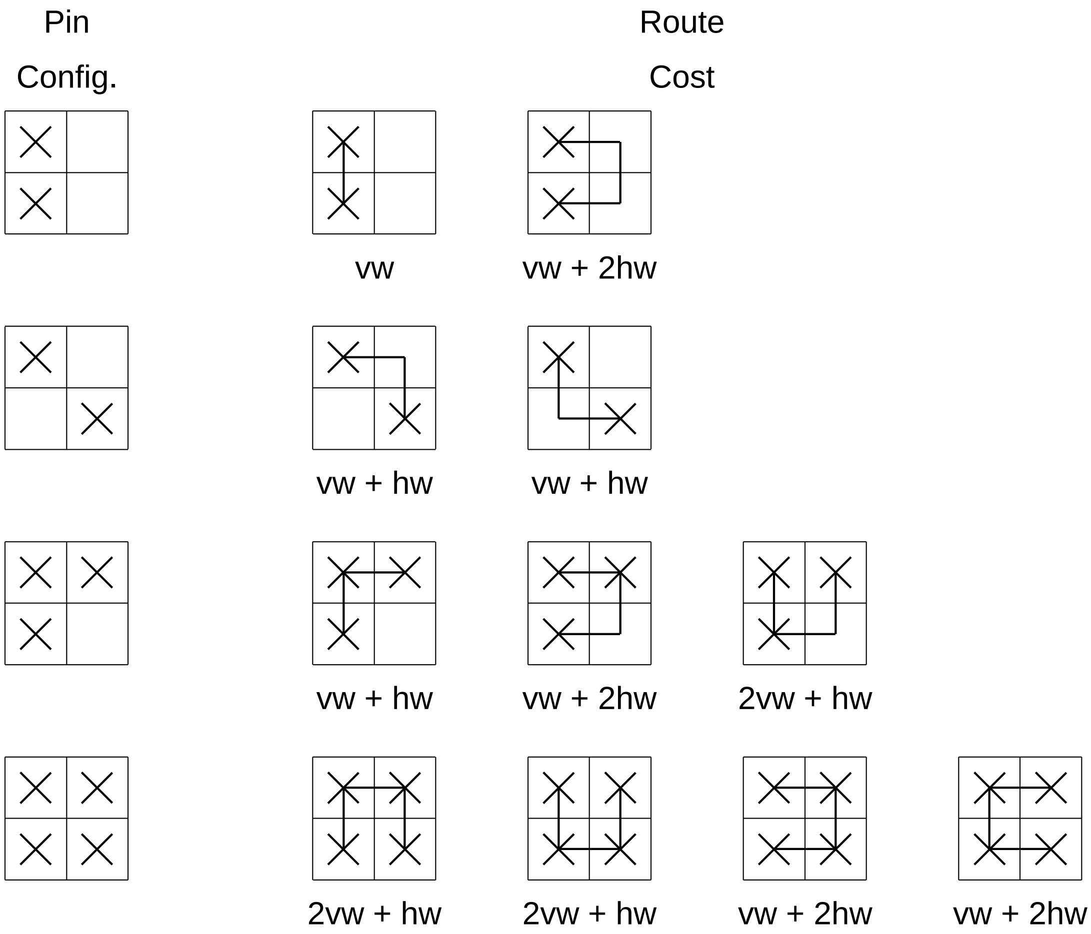


Figure 4.7. Improved quadrisection net cost function

Furthermore, if c were the only connection for a net n in q, the connections to q for n may be removed also. These changes or reroutings of the nets cause changes in the calculated cost of the net. In order to account for the change in cost, a system of *gain tables* is used which reflects the change (gain) in cost of the nets with respect to movements of cells from one quadrant to another. A separate gain table is used for each of the twelve combinations of $q, r \in \{0,1,2,3\}$ such that $q \neq r$. Figure 4.8 shows the twelve combinations of q and r and the associated movement of a cell c from quadrant q to r.

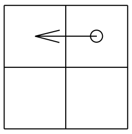
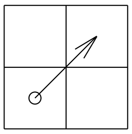
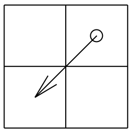
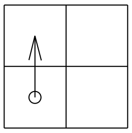
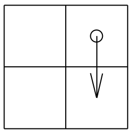
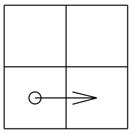
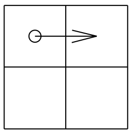
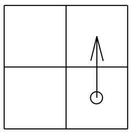
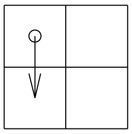
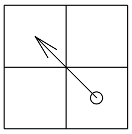
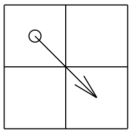
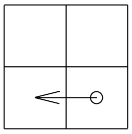
(q , r)	Movement	(q , r)	Movement
(0 , 1)		(2 , 0)	
(0 , 2)		(2 , 1)	
(0 , 3)		(2 , 3)	
(1 , 0)		(3 , 0)	
(1 , 2)		(3 , 1)	
(1 , 3)		(3 , 2)	

Figure 4.8. Quadrisection gain tables

Each gain table contains a list of the movable cells currently located in quadrant q. Each cell c has associated with it a cost value, determined by summing up the expected change in cost for each of the nets connected to c, if c were to be moved to quadrant r. Since our goal is to minimize the net length and cost, a cell is selected for movement from the gain table when it has the best or smallest cost gain. To efficiently select the cells to be swapped or moved, we utilize the same data structure (Figure 4.9) as Suaris and Kedem, which is derived from the data structures of Fiduccia and Mattheyses [59]. In this data structure, sets of cells with the same gain value are placed in doubly-linked lists called *buckets*. These buckets are indexed by the gain value, with the smallest gain value denoted as *CurrMinGain*. The Cell List Pointer Array provides $O(1)$ access to any entry in the bucket lists, and the doubly-linked lists provide for $O(1)$ insertion and deletion of bucket entries. For a more detailed description of how to determine the gain of each cell in the gain tables, see [16].

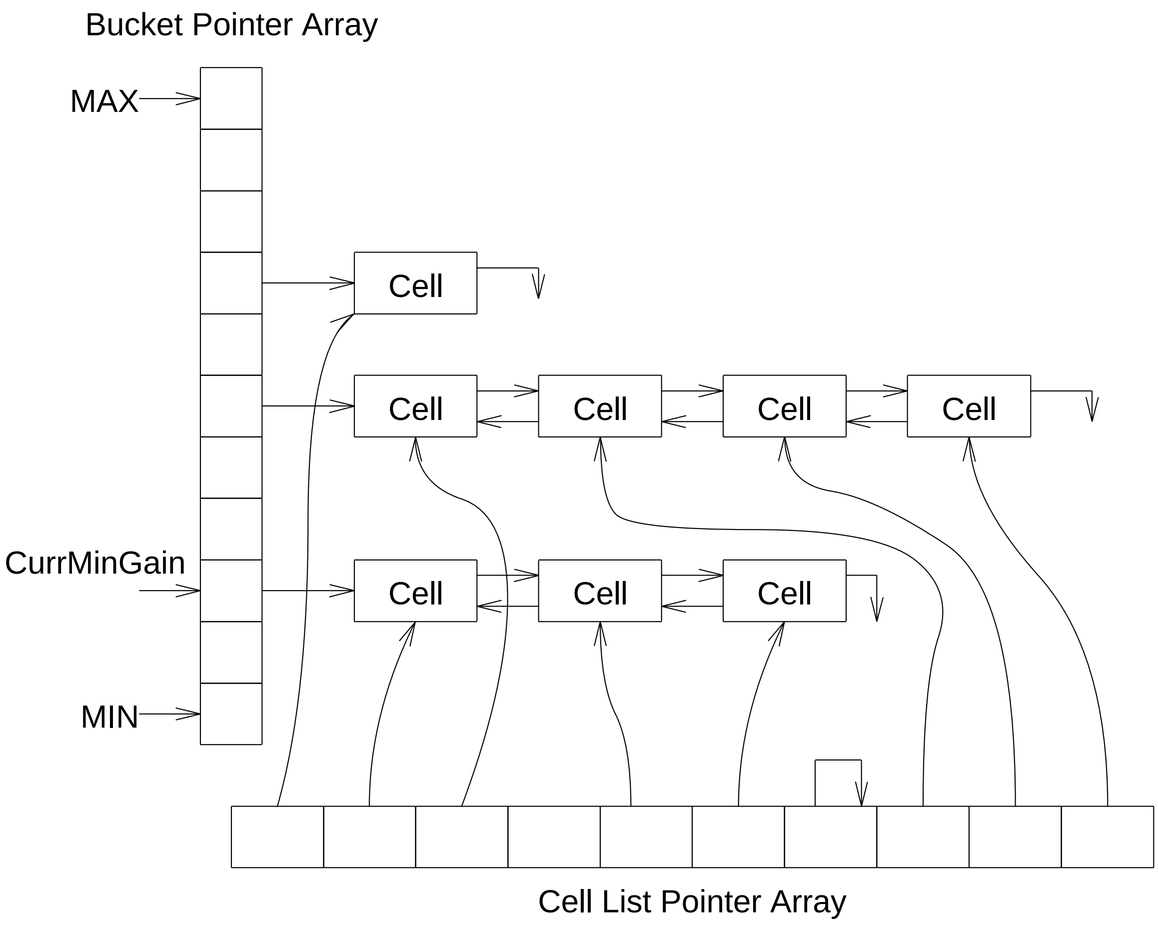


Figure 4.9. Gain table data structure

In addition to the determination of the minimum gain cell, another important criterion in the selection of cells is the determination of whether the movement of the cell would cause an imbalance in the total area of the cells ($CellArea$) occupied by each quadrant. A minimal cut would be achieved if all cells were in one quadrant; however, this is clearly no closer to the solution. A maximum cell area value ($MaxArea$) and a minimum cell area value ($MinArea$) are determined for each quadrant. If $CellArea_q - size(c) \geq MinArea_q$ and $CellArea_r + size(c) \leq MaxArea_r$, then c may be moved from q to r.

We propose that another important enhancement to the Suaris-Kedem Quadrisection algorithm would be the ability to swap cells. Size restrictions can place a tight limitation on the set of cells allowed to be moved; often, minimum gain cells fall into this category. We avoid this common problem by allowing cells of equal size to be swapped. A secondary restriction on the selection of the second cell for the swapping is that the cell must be in the quadrant r, have a cost gain of 0, and have no nets in common with the first cell selected. This is necessary to maintain the proper gain values.

After a cell is moved from one quadrant to another, the cell is locked in place, the cell's bucket entry is removed, and the current state is stored on a stack. The sequence of selecting and moving cells is repeated until no cells can be selected for movement or when a sequence of k_s selections of cells with gains > 0 has taken place. The stored state information is then used to backtrack and undo cell movements which have only worsened the net states and the partition of the cells into quadrants. The steps of cell selection followed by backtracking are called a *pass* and are repeated k_{pass} times, or until no gains are made on consecutive passes.

4.3.2. Routing of the quadrisection

At the end of the quadrisection operation, the cells of the portion of the layout have been placed in one of the four quadrants while minimizing the net crossings over the boundaries between the quadrants. A quadrisection routing operation is then used to verify and lock in place the routing of the nets across the four boundary segments. A single iteration of the algorithm presented in Chapter 3 for determining the routing of the nets in a two-by-two array of routing blocks is used, since each quadrant matches one block of the two-by-two array. This operation is $O(n)$, where n is the number of nets, and must be done once for each quadrisection operation completed.

If the route-based cost function is used, it is necessary to know the routing of the nets in the quadrisection region before quadrisection can take place. The routing must be based on the current placement of the cells at the beginning of quadrisection. Thus, one iteration of the two-by-two routing algorithm will be performed before as well as after the quadrisection when the route-based cost function is used.

To determine the best routing of the nets, an accurate measure of the routing capacities across the four quadrisection boundaries must be made. Since the exact locations of the cells is not known until the placement algorithm completes, we measure the routing capacity along the horizontal boundaries as the average number of feedthroughs available divided by the number of rows over which the cells are to be placed. The simplex computations can then insert feedthrough cells in the rows or increase the channel height, if needed. As cells are moved, the horizontal capacity measure can vary and must be recalculated before every routing. The vertical boundary capacities are an average of the number of tracks available in the channels intersecting the boundary.

4.3.3. Initial placement for quadrisection

In the discussion of the quadrisection placement algorithm, we mentioned that the cells to be placed are initially divided into four groups. In [16] a two-stage min-cut scheme is used to generate the initial partition, or seed, for the quadrisection. In this section we propose a new method called *Restricted Global Bisection* for providing the initial partition for the quadrisection-based placement.

4.3.3.1. The X-dimension restricted global bisection

The bisection is performed separately in the X-dimension and the Y-dimension. The X-dimension bisection consists of the set of cells between the coordinates x_{lo} and x_{hi} and the bottom and top borders of the layout. The values for x_{lo} and x_{hi} are determined by the quadrisections at the previous level in the hierarchical decomposition. The vertical lines separating the quadrisection regions and the vertical lines which cut down the middle of a column of quadrisection regions are used as the domain of the values of x_{lo} and x_{hi}, thus giving 2^k separate X-dimension bisection regions, where k is the current hierarchy level (Figure 4.10). Given x_{lo} and x_{hi} for a bisection, the set of cells is then partitioned into two groups, separated by a line (x_{mid}) halfway between x_{lo} and x_{hi}.

The bisection region is further divided up vertically, with the number of partitions $PRT_{bsect} = 2^k$. Throughout the bisection algorithm, the cells are restricted to horizontal movements only. Thus, every cell stays in the same vertical partition (Figure 4.10) and each cell's y-coordinate remains untouched. At the start of the bisection, the set of cells in each partition is split in two halves using a clustering partition algorithm. This becomes the seed for the bisection algorithm.

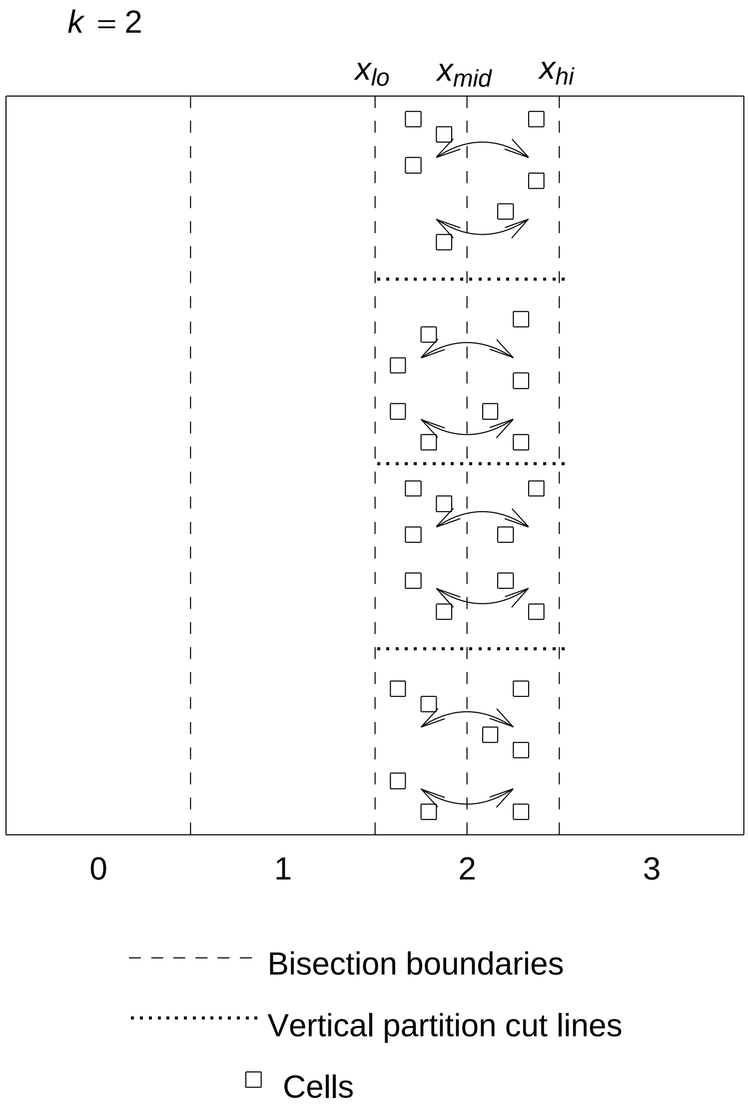


Figure 4.10. Partitioning for X-dimension restricted global bisection

In the same way as [59], cells are assigned a cost function based on the net connections and are moved or swapped between the bisection halves to reduce the overall cost. To model the restricted movement of cells in the cost function, we have devised a cost function based on the number of crossings by a minimal length net over the partition line (x_{mid}). Figure 4.11 shows an example in which moving the highlighted cell to the other half would decrease the number of crossings over x_{mid}. It is very important to

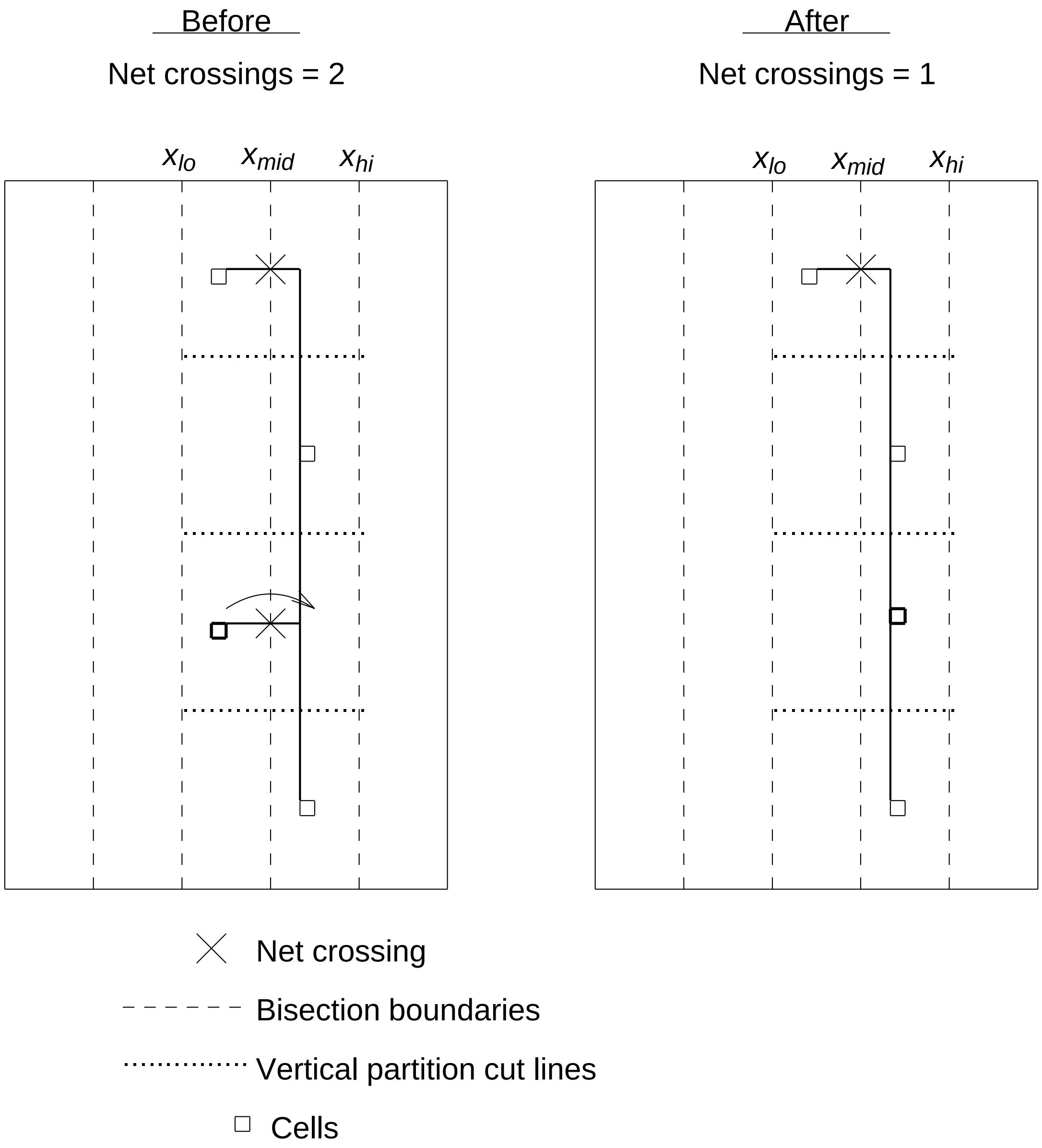


Figure 4.11. Bisection cost function example

note that the net cost function is based on the locations of the cells from the top to the bottom of the layout, not only in a small section. By evaluating the full height of the layout, we are able to line up nets which pass vertically over many rows and, thereby, reduce the demand of nets to occupy track space in the channels between the rows.

The cells in the two halves are assigned a cost equal to the sum of the costs of each net attached to the cell. These costs are assigned in gain tables similar to the quadrisection algorithm and to [59]. Cells are selected from the two gain tables (cells move only from one half to the other) so as to minimize the net cost gain. Similar to our quadrisection algorithm, cells may be selected for swapping if the constraints are met.

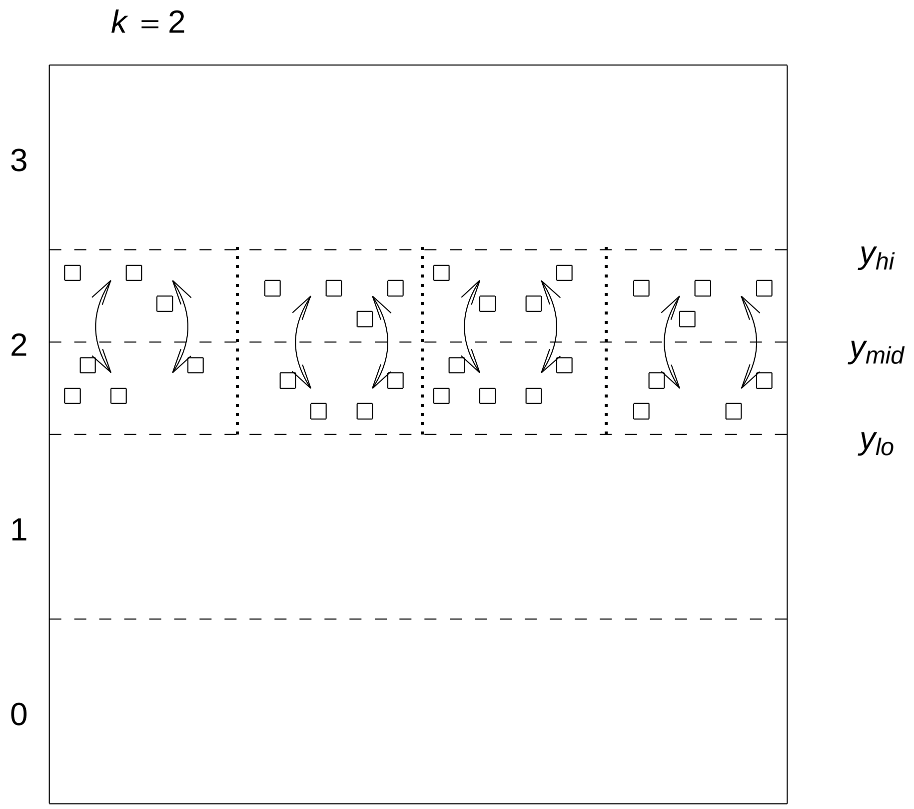
4.3.3.2. The Y-dimension restricted global bisection

Alternately, the Y-dimension restricted global bisection algorithm partitions the layout into horizontal strips the width of the layout area. By applying the cost function horizontally, we reduce the demand of the nets for a high number of feedthroughs and route each net in as few channels as possible. Figure 4.12 shows the configuration for the Y-dimension bisection operation.

The cell size restrictions on movement are similar to quadrisection, but consideration is given to the area available on each half. In the same manner as quadrisection, the sequence of cell selections and movements until no more moves are possible is called a *pass* and is followed by a backtrack to the last best state. A sequence of *pass*es is performed until either a limit is reached or until no further gains can be made.

4.3.3.3. Combining X- and Y-dimension bisectioning

Since the X- and Y-bisection algorithms exclusively alter the $x-$ and $y-$coordinates of the cells (respectively), they are independent of each other, and the two dimensions can be evaluated simultaneously. Following both evaluations, the cells have been preplaced in one of the quadrants of the quadrisection to be involved in the current level of the decomposition. Effectively, the bisections perform an initial placement of the cells



Bisection boundaries
Horizontal partition cut lines
Cells

Figure 4.12. Y-dimension restricted global bisection

for the subsequent quadrisection operations, based on the positions of all cells in the same layout block row and columns. However, since the x– and y–coordinates of each cell are set independently, the balance of cell areas may not be valid. Therefore, we move selected cells from the fullest quadrant to the least full one at the beginning of the quadrisection algorithm.

4.3.4. Two-by-N global routing

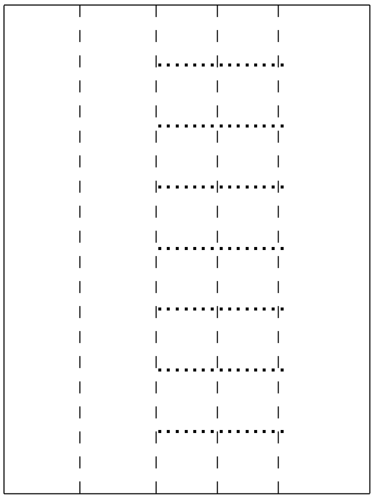
Following the bisection operation, a two-by-N routing of each bisection region is performed, again using the algorithm presented in Chapter 3 for determining the cuts along the $X-$ and $Y-$axes. The goal of the routing is to determine the sets of nets crossing each half of the partition lines running perpendicular to the the bisection line at x_{mid} (or y_{mid}). Figure 4.13 shows the steps in the evaluation of a two-by-N routing, which takes the form of a binary tree execution. The dashed lines denote the boundaries and cut line for the bisection placement. The dotted lines denote the axis lines to be determined. The depth of the tree is equal to the current level number in the decomposition hierarchy.

Although the bisection routing was introduced as immediately following the bisection placement, it is necessary to perform a bisection routing immediately after the quadrisection placement also. The two-by-N bisection routing following the quadrisection placement is necessary not only because the balancing of cell areas may change the best routing between quadrisections evaluated in parallel, but also to optimize the routing connections following movements of cells among the quadrants. Thus, at each level of the hierarchical decomposition, the bisection routing algorithm is effectively applied twice.

4.4. Algorithm Outline

In Figure 4.14, a graphical description of the placement and routing algorithm is shown. In this figure, each operation performed at each level of the hierarchical decomposition is denoted by a set of circles between a pair of horizontal dashed lines intersecting the appropriate column. The circles represent instances of the operations to be

Initial Bisection Problem



2-by-N Routing Evaluation Steps

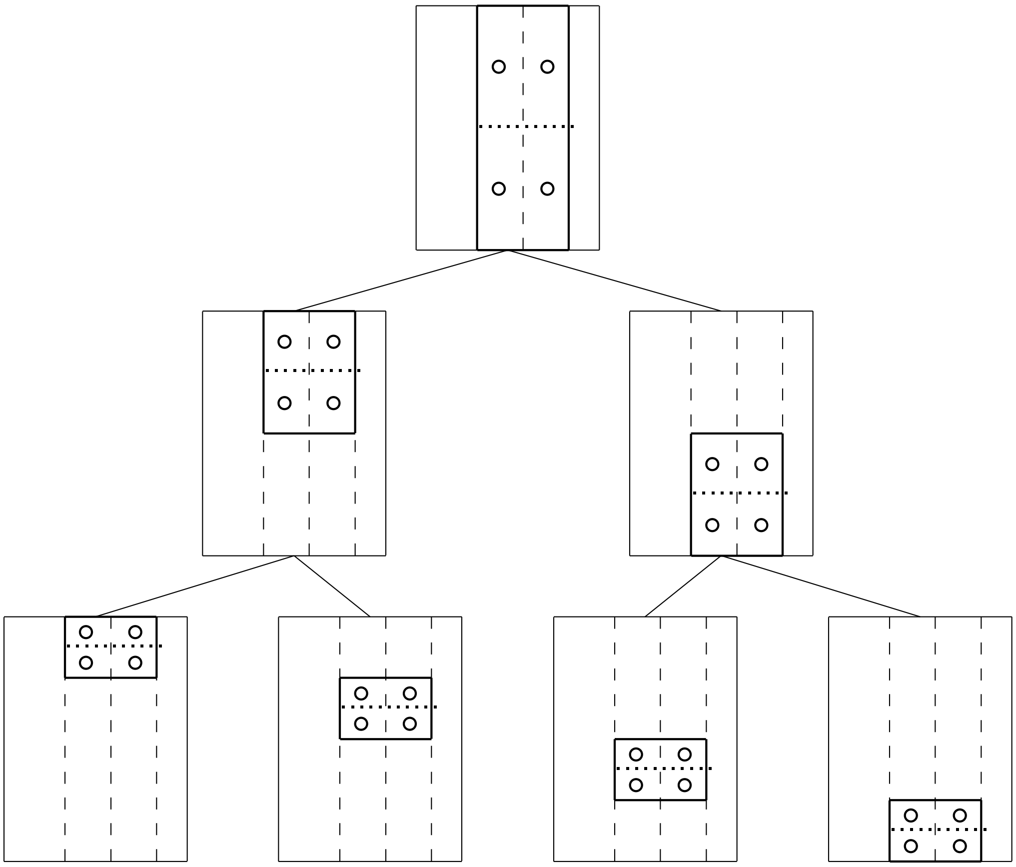


Figure 4.13. Two-by-N routing of a bisection region

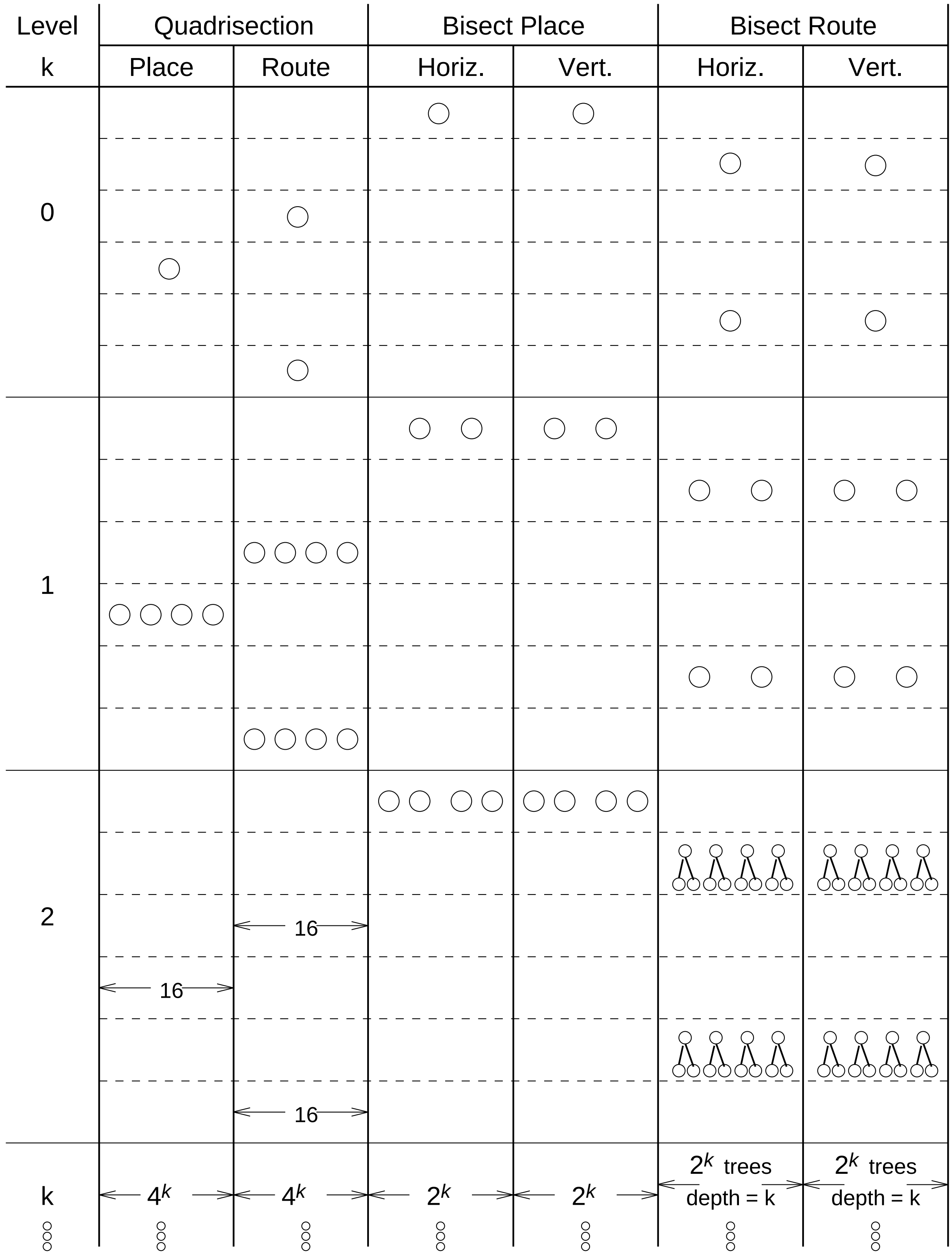
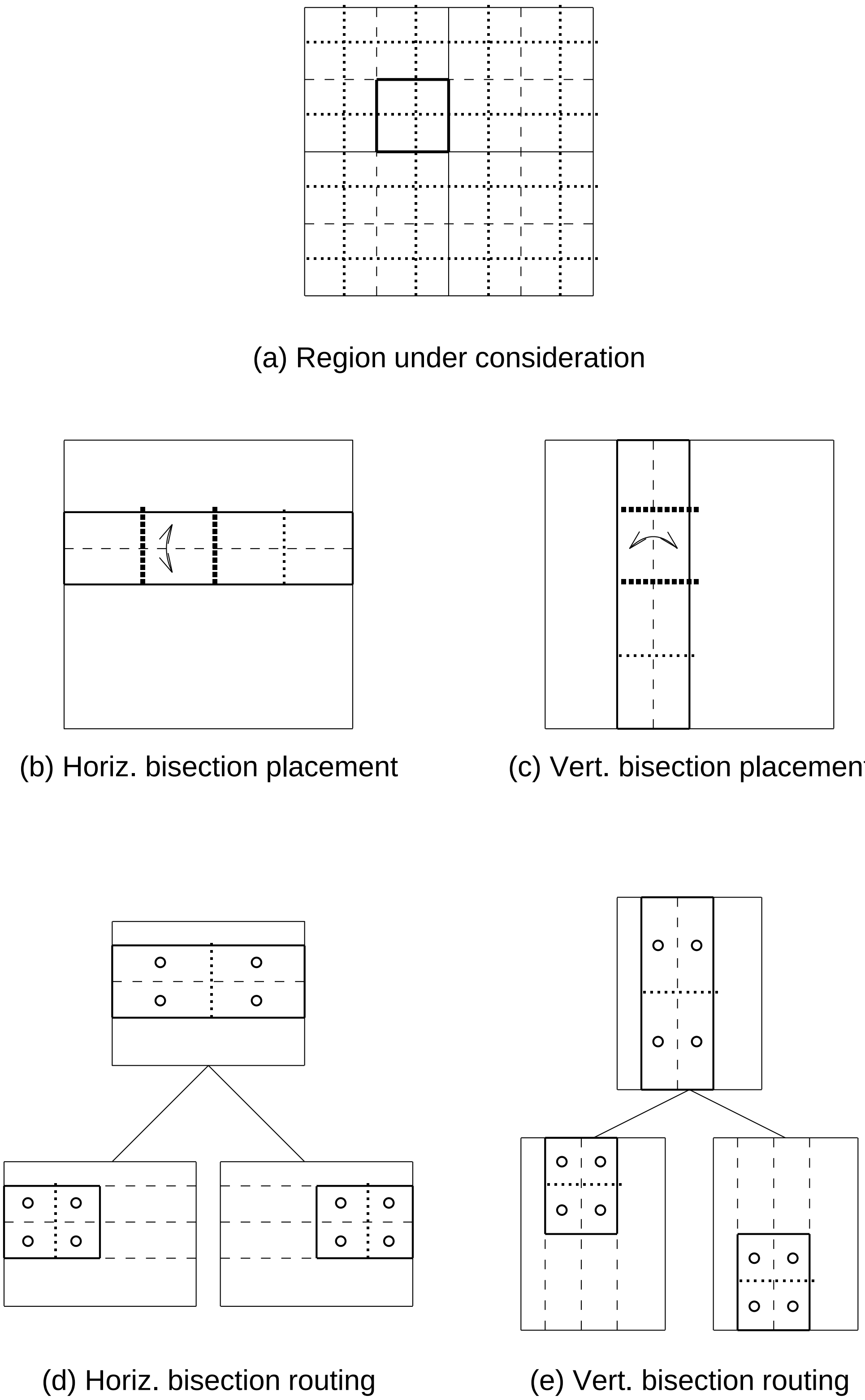


Figure 4.14. Placement and routing decomposition

performed on a portion of the layout. For example, in the quadrisection placement column, one circle at hierarchy Level 0 represents a quadrisection covering the entire layout. Four circles at Level 1 represent the four quadrisections, each covering one-fourth of the layout.

At each level of the decomposition, the cells are initially placed using the global X- and Y-bisection placement algorithm. This is immediately followed by a two-by-N routing of the same regions to determine the net crossings for the boundaries of each quadrisection placement region on that level. Next, a two-by-two routing of each quadrisection placement region is performed, taking into account the nets crossing through and ending in the region, to set up the current routing configuration for each net to be used in the quadrisection cost function. The quadrisection placement algorithm is then used to improve the current locations of the cells (provided by the previous bisection placements). Since the previous two-by-N routing may need to be modified due to movements of the cells inside the quadrisection region, the routing algorithm is repeated. Finally in the hierarchy level, a two-by-two routing is applied to each quadrisection region to fix the crossing locations of the nets on the four cut lines (A–D) separating the four quadrants.

An example showing the operations at Level 2 in the decomposition for one region of the layout is shown in Figure 4.15. In Figure 4.15(a), the region under consideration is the square outlined in bold. The dashed lines denote the boundaries of the bisection region for Level 2. The dotted lines represent the internal axis lines of the bisections and quadrisections for Level 2. At Level 3 the dotted lines would represent the bisection and quadrisection borders. Figures 4.15(b) and (c) show the horizontal and vertical



(a) Region under consideration

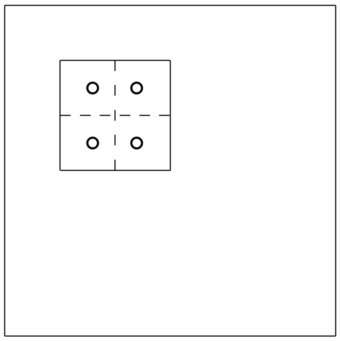
(b) Horiz. bisection placement

(c) Vert. bisection placement

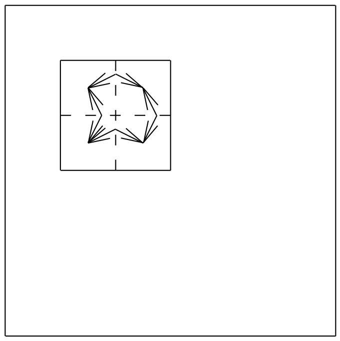
(d) Horiz. bisection routing

(e) Vert. bisection routing

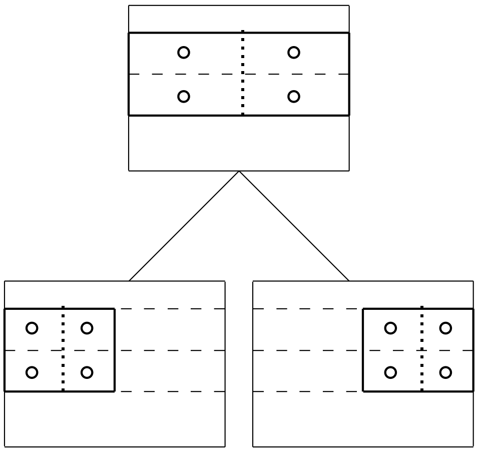
Figure 4.15. Decomposition example



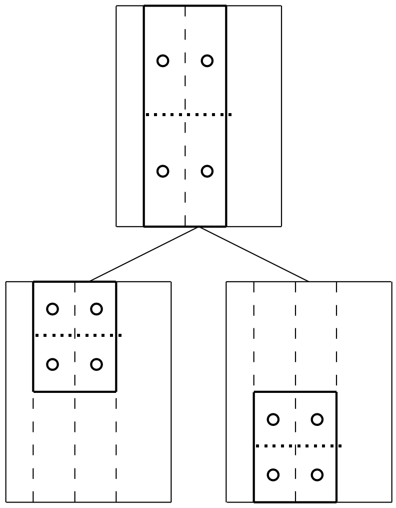
(f) Quadrisection routing



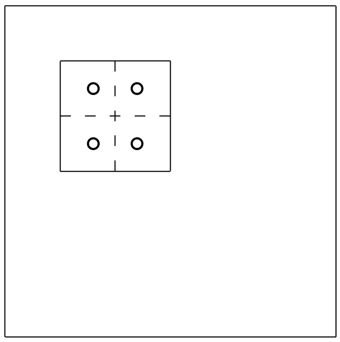
(g) Quadrisection placement



(h) Horiz. bisection routing



(i) Vert. bisection routing



(j) Quadrisection routing

Figure 4.15. Continued

bisections, respectively. The cells being displaced must remain between the pairs of bold dotted lines. Figures 4.15(d) and (e) show the horizontal and vertical 2x4 routing of the bisection regions. Each 2x4 routing requires the solution of three 2x2 routing instances. Figure 4.15(f) shows the quadrisection routing that is necessary before route-based quadrisection can be performed (Figure 4.15(g)). Figures 4.15(h), (i), and (j) show the repetition of the routing operations performed earlier.

4.5. Detailed Routing

At the end of the placement and routing step, each layout block contains a set of cells and lists of the nets crossing each of its borders. This information is then processed into a list of cells and feedthroughs in each row and a list of net segments in each channel. The final step of the layout process, then, is to take the cell positions and global routing information, set up each of the channel routing problems, and solve the channel routes using a standard channel routing algorithm. Once the channel routing problems are set up, each is independent of the others and can be evaluated in parallel.

4.6. Parallelisms and Algorithm Complexities

The placement and routing operations described in the previous sections must be performed in a sequential manner at each level of the hierarchical decomposition; however, within each level, we can take advantage of many parallelisms. Within the operations, shown in Figure 4.14, the instances are completely independent of each other, except that the child XY routing instances are dependent on their respective parent node. For example, the Level 1 quadrisection operation consists of four instances of the quadrisection problem, each covering one fourth of the layout area. Each instance is independent of the other three, since the bisection and routing steps of Level 0 have determined the locations at which nets cross the boundaries into the layout area of the Level 1 quadrisection instance. Furthermore, since the bisection placement instances alter only one of a cell's two coordinates, two or more bisections overlapping in different directions can be evaluated simultaneously. From Figure 4.14, it is clear that after the first two hierarchy levels, the available parallelism is very great.

4.6.1. Complexity evaluation

The complexity of one *pass* of the quadrisection placement algorithm using the standard cost function has been shown to be $O(m)$ in [16], where m is the number of pins in the circuit. Since k_{pass} is O(1), an instance of the quadrisection placement algorithm is O(m). The use of the route-based cost function will not affect the complexity since the operations are almost identical to those for the standard cost function. Furthermore, the addition of the swapping of cells does not change the overall complexity since the operation consists of scanning the zero-gain bucket list from one gain table until a match is found and since swapping is used only under certain conditions.

Let R be the number of layout block rows, C be the number of layout block columns, and $Z = MIN(R, C)$. Since the total number of quadrisection placement instances, N_{Qplace}, is equal to the number of nodes in a $\log_2 Z$ level quad-tree (the placement is performed at each hierarchy level), we have

$$N_{QP} \approx \sum_{i=0}^{\log_2 Z - 1} 4^i = \frac{Z^2 - 1}{3}.$$

The complexity of the quadrisection routing is $O(n)$, where n is the number of nets, and since the number of instances N_{QR} is equal to two times the number of quadrisection placement instances (each placement has an associated routing before and after), we have

$$N_{QR} \approx \frac{2(Z^2 - 1)}{3}.$$

In the same way as the quadrisection *pass*, the complexity of a single bisection *pass* is O(m), and the complexity of a bisection placement instance is O(m); since the number of bisection evaluations N_{BP} is equal to the number of nodes in a binary tree of depth $\log_2 R$ for the X-dimension bisection routing plus the number of nodes in a binary tree of depth $\log_2 C$ for the Y-dimension bisection routing, we have

$$N_{BP} \approx \sum_{i=0}^{\log_2 C - 1} 2^i + \sum_{i=0}^{\log_2 R - 1} 2^i.$$

This expression can be simplified to

$$N_{BP} \approx R + C - 2.$$

Since each level's bisection routing operation is repeated, the number of two-by-two routing instances N_{BR} required to evaluate all of the bisection routes is equal to twice the number of nodes in a binary tree of binary trees. The summation can be written as follows:

$$N_{BR} \approx 2(N_{BRX} + N_{BRY}),$$

where

$$N_{BRX} \approx \sum_{i=0}^{\log_2 C - 1} 2^k (2^k - 1) = \frac{1}{3} C^2 - C + \frac{5}{3}$$

and

$$N_{BRY} \approx \sum_{i=0}^{\log_2 R - 1} 2^k (2^k - 1) = \frac{1}{3} R^2 - R + \frac{5}{3}$$

After combining the expressions we have

$$N_{BR} \approx \frac{2}{3}(C^2 + R^2 - 3C - 3R + 10).$$

Note that the above expressions give approximations. The expressions become equalities when R and C are powers of 2.

Since synchronization between processes is necessary after each operation (e.g., quadrisection placement and bisection routing) in a parallel environment, it is difficult to evaluate exact expressions for the expected speedup as a function of the number of processes (P). Let $T_{QP}(k)$, $T_{QR}(k)$, $T_{BP}(k)$, and $T_{BR}(k)$ be the average execution times for each of the respective operations as a function of the hierarchical level. As the algorithm proceeds, the size of the problem to be solved is proportional to the area under consideration. Note that $T_{QR}(k) = T_{BR}(k)$ is the time to evaluate a single two-by-two routing instance. Let T_{sync} be the performance loss of time due to synchronization as processes remove tasks from the various queues, and let T_{barr} be the average time spent waiting for other processes to finish the tasks of the current operation.

The expected time $T(P) = T_{start} + T_{full}$ for P processors is the number of time steps before all processors have a constant supply of jobs, plus the time to evaluate the remaining tasks divided by P. The number of instances of the six operations for hierarchy level k is

$$N_{op}(level) = 4^k + 2(4^k) + 2^k + 2(2^{2k} - 2^k),$$

which is the sum of the number of QP, QR, BP, and BR instances on the level. The startup time for P processors is

$$T_{start} = \sum_{k=0}^{\log_2 P - 1} T_{BP}(k) + T_{BR}(k) +$$

$$\frac{T_{QR}(k)}{2^k} + \frac{T_{QP}(k)}{2^k} + T_{BR}(k) +$$

$$\frac{T_{QR}(k)}{2^k} + 5T_{barr} + T_{sync}N_{op}(k).$$

Now let us define the time spent evaluating quadrisection (placement and routing) operation instances on a level as

$$T_Q(k) = 4^k(2T_{QR}(k) + T_{QP}(k) + 3T_{sync}) + 2T_{barr}.$$

Note that the synchronization operations are necessary each time a task is taken from the queues. Further note that only two synchronization barriers are required after the quadrisection operations since the quadrisection routing immediately preceding the quadrisection placement can be merged for execution in the same task as the placement. The time spent evaluating bisection (placement and routing) operation instances on a level for one dimension is

$$T_B(k) = 2^k(T_{BP}(k) + T_{sync}) + T_{barr} + (2^{2k} - 2^k)(T_{BR}(k) + T_{sync}) + 2T_{barr}.$$

Note in this case, syncronization operations are required for removing each task from the queues as well as a synchronization barrier following each bisection operation. Thus, the time spent in full parallel execution is

$$T_{full} = \frac{1}{P}\sum_{k=\log_2 P}^{\log_2 Z} T_Q(k) + \frac{1}{P}\sum_{k=\log_2 P}^{\log_2 C - 1} T_B(k) + \frac{1}{P}\sum_{k=\log_2 P}^{\log_2 R - 1} T_B(k),$$

and thus the execution time for P processors is

$$T_P = T_{start} + T_{full}.$$

The complexity of the tasks is proportional to the area of the evaluation. For QP and QR evaluations, $T(k+1) \approx \frac{1}{4}T(k)$, and for BP and BR evaluations, $T(k+1) \approx \frac{1}{2}T(k)$. In other words, the magnitude of time spent at each hierarchical level is approximately the same. Therefore, T_Q can be rewritten as

$$T_Q(k) = (2T_{QR}(0) + T_{QP}(0) + 3T_{sync}),$$

and T_B can be rewritten as

$$T_B(k) = (T_{BP}(0) + T_{sync}) + 2^k(T_{BR}(0) + T_{sync}).$$

Finally, the expected speedup S_P for P processors is

$$S_P = \frac{T_1}{T_P}.$$

4.7. Results

There are many aspects of the parallel placement and routing algorithm which could be evaluated. In this section we examine a few of these aspects including the following: the effect of route-based quadrisection, the effect of bisection for initial placement before quadrisection, the solution quality, and the parallel performance on a number of example circuits.

4.7.1. Implementation

The parallel algorithm for placement and routing has been implemented in the *PARAGRAPH* (PARallel Algorithm for Global Routing and Placement Hierarchically) system using approximately 12,000 lines of C language code. The code has been compiled for various machines, but the target machine of particular interest is an Encore Multimax. The Multimax features eight NS32532 processors (rated at 5 MIPS) utilizing up to 64 Megabytes of shared memory. The code is written to make use of the *fork()* and *join()* function calls for creating slave processes, *share_malloc()* and the *shar* data type for creating and using shared memory, and the semaphore function calls to provide a locking mechanism during critical sections of code (especially in the scheduler). All queue modifications require critical sections of code to prevent multiple processes from simultaneously accessing the data.

In the parallel processing mode, each operation type is provided a unique task queue. As tasks are created, they are placed on the end of the proper task queue and await execution by any of the processors that become available. Synchronization among the processes between operations is achieved by having the master process monitor (MP) the current task queue, monitor the state of idleness of each slave process (SP), and control an "operation indicator." Since there are dependencies from one operation to the next, it is necessary to synchronize after every operation to guarantee the correctness of solution. For example, the quadrisection route depends on the preceding bisection route in order to properly establish the sets of nets crossing its boundaries. After synchronization, the next operation is enabled by the MP through the operation indicator, and waiting processes are allowed to take tasks from the new operation's task queue.

Figure 4.16 provides a high-level look at the commands for the master process (MP) and slave processes (SP). Note that in this figure we have collapsed the quadrisection routing (used with route-based quadrisection before the quadrisection placement) into a single quadrisection routing and placement (QRP) operation. The SP are continually checking the various queues for the quadrisection routing and placement (QRP), bisection routing (BR), quadrisection routing (QR), and bisection placement (BP) tasks. Along with executing any tasks available, the MP is responsible for creating the initial top-level task, making the transitions between the operation TYPES using the shared operation indicator variable, and eliminating the SP after completion of the placement and routing algorithm.

MASTER

```
Initialize();
DoFork(NumProcs-1);
SetTopLevel();
Level = 0;
TYPE = QRP;
WHILE (!DONE) {
     WHILE (GetTask(QRP))
          Eval(QRP,Level);
     BARRIER(TYPE = BR);
     IF (! TopLev) {
          WHILE (GetTask(BR))
               Eval(BR,Level);
     }
     BARRIER(TYPE = QR);
     WHILE (GetTask(QR))
          Eval(QR,Level);
     Level++;
     BARRIER(TYPE = BP);
     WHILE (GetTask(BP))
          Eval(BP,Level);
     BARRIER(TYPE = BR);
     WHILE (GetTask(BR))
          Eval(BR,Level);
     BARRIER(TYPE = QRP);
     DONE = CheckIfDone();
}
DoJoin();
```

SLAVE

```
Initialize();
WHILE (TRUE) {
     If (GetTask(TYPE))
          Eval(TYPE,Level);
}
```

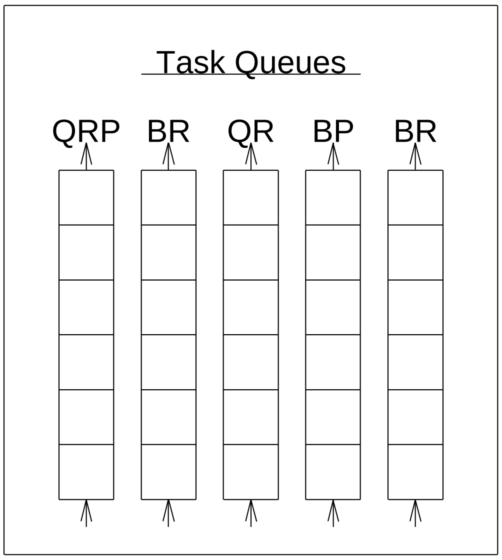


Figure 4.16. Parallel placement and routing pseudocode

4.7.2. Benchmark circuits

The parallel placement and routing algorithm was evaluated on six placement problems. Two of the circuits are the *Primary1* (P1) and *Primary2* (P2) benchmarks from the Microelectronics Center of North Carolina (MCNC). The remaining four are other standard cell circuits of varying sizes. Table 4.1 provides the number of cells, the number of pads, and the number of nets in each of the circuits.

4.7.3. Evaluation of net cost function

In the following tables, we evaluate the effect of the various algorithm options discussed in the chapter on the placement quality. The total length of the net segments in the channels is denoted as WL, the number of routing tracks as determined by summing up the maximum channel density for all channels is denoted as TC, and the layout area of the rows of cells and the channels is denoted as LA.

The first comparison we make is among the results from different executions of PARAGRAPH using different weightings of the horizontal and vertical net segments. The net weightings are used in the quadrisection and bisection placement operations to minimize the net cut length. Table 4.2 compares the results for seven combinations of

Table 4.1. Benchmark statistics

Circuit	Cells	Pads	Nets
Z1	469	37	494
Z2	1691	61	1979
Z3	2776	64	3258
Z4	2976	62	4207
P1	752	81	1185
P2	2907	107	3710

cost parameters for the six example circuits. In this set of experiments, the route-based cost function, initial placement by bisection, and cell swapping methods were all enabled during the executions. From this table, we notice that the results vary widely for the various combinations, depending on the circuit. Therefore, we are unable to set forth a combination that clearly outperforms the others.

Table 4.2. Comparison of net cost parameters

Circuit	HW	VW	WL	TC	LA
z1	1	1	563	218	16256
	2	7	518	201	15787
	5	2	570	230	16910
	9	2	594	227	16662
z2	1	1	14540	1305	194370
	2	7	17272	1388	232974
	5	2	15149	1365	206899
	9	2	15174	1405	199214
z3	1	1	9970	1426	69092
	2	7	13224	1716	84252
	5	2	7681	1223	53442
	9	2	8197	1165	54453
z4	1	1	22113	1881	93690
	2	7	9707	1457	63789
	5	2	15435	1769	89846
	9	2	20165	1879	94198
p1	1	1	1856	381	28195
	2	7	1917	382	28531
	5	2	2331	392	29293
	9	2	1929	405	29576
p2	1	1	13800	1389	111703
	2	7	16080	1471	116653
	5	2	16958	1536	121556
	9	2	16282	1539	118352

4.7.4. Evaluation of initial placement by global bisection

Table 4.3 contrasts the quality of the final results, with and without the use of the restricted global bisection algorithm as an initial placement for the quadrisection algorithm. For all executions in this set of experiments, the net cost parameters were identical, route-based cost functions were used, and cell swapping was enabled. From this table, it is very clear that the bisection placement algorithm is important in setting up the quadrisection placement since it considers nets and cells across the width or height of the layout.

4.7.5. Evaluation of cell swapping

A comparison is made in Table 4.4 of the quality of the placement using cell swapping with placement without cell swapping (displacement only). Again, the measure of quality used for the comparison is the total wirelength, track count, and layout area. For all executions in this set of experiments, the net cost parameters were identical, bisectioning was used for the initial placement of the cells, and route-based cost functions were used. The table shows that in nearly every case, the placement algorithm allowing

Table 4.3. Initial placement alternatives comparison

	With Bisect Placement			Without Bisect Placement		
Circuit	WL	TC	LA	WL	TC	LA
z1	518	201	15787	903	291	20807
z2	17272	1388	232974	19730	1474	259812
z3	7934	1295	57320	35038	2411	127789
z4	9707	1457	63789	33333	2148	109720
p1	1918	382	28531	2460	440	32917
p2	16080	1471	116653	32478	2093	148902

Table 4.4. Effect of cell swapping

Circuit	With Cell Swapping			Without Cell Swapping		
	WL	TC	LA	WL	TC	LA
z1	518	201	15787	549	221	16483
z2	17272	1388	232974	15022	1359	203940
z3	7934	1295	57320	9119	1515	66344
z4	9707	1457	63789	12359	1534	81264
p1	1917	382	28531	2897	476	36579
p2	16080	1471	116653	27630	2043	135304

cell swapping achieves a better result than the placement depending on the displacement of cells.

4.7.6. Route-based placement evaluation

A comparison was made earlier in the chapter between standard quadrisection cost function and a cost function that is based on the actual routing of the nets. In Table 4.5, we compare the two cost functions based on the wirelength, the track count, and the final layout area. For this set of experiments, the net cost parameters were identical, bisectioning was used for the initial placement of the cells, and cell swapping was

Table 4.5. Route-based vs. standard cost functions

Circuit	Route-Based Cost			Standard Cost		
	WL(x1000)	TC	LA(x1000)	WL	TC	LA
z1	518	201	15787	567	224	17252
z2	17272	1388	232974	16873	1392	222567
z3	7934	1295	57320	8751	1534	69953
z4	9707	1457	63789	14416	1777	82637
p1	1918	382	28531	1928	388	28341
p2	16080	1471	116653	15705	1464	117018

enabled. From the table, it is clear that in the majority of cases the route-based placement performs better than the standard cost function. This is especially important for layouts with very limited routing resources.

4.7.7. Comparison to TimberWolf 5.4

Finally, Table 4.6 compares the solution quality of the sequential algorithm for placement and routing (PARAGRAPH) versus the Timberwolf 5.4 placement and routing package. All of these experiments were made on a single processor of the Encore Multimax, and the suggested parameters were supplied to TimberWolf. For PARAGRAPH, bisectioning was used for the initial cell placement, route-based cost functions were used, and cell swapping was enabled. The table shows the runtime and final layout area measurement for the example circuits along with the uniprocessor execution time as measured by *getrusage()* for TimberWolf and elapsed real time (should be greater than or equal to the *getrusage()* values) for PARAGRAPH. The runtime (RT) values are measured in seconds and the wire length (WL) and layout area (LA) are to be multiplied by 1000. The TimberWolf placement algorithm is based on simulated annealing [21]

Table 4.6. Comparison to TimberWolf

	TimberWolf 5.4				PARAGRAPH			
Circuit	RT	WL	TC	LA	RT	WL	TC	LA
z1	1913	208	90	10697	124	518	201	15787
z2	14977	3957	558	94346	1648	14540	1305	194370
z3	----	-----	---	----	3062	7934	1295	57320
z4	31713	2054	597	14100	4215	9707	1457	63789
p1	4636	607	169	16121	270	1856	381	28195
p2	30132	3821	487	58517	3988	13800	1389	111703

and the global routing algorithm is based on Steiner tree minimization [37]. TimberWolf has been improved over the last several years to the point where it produces very good results and has become a standard for layout quality comparison. Unfortunately, the runtime for TimberWolf often exceeds several hours for average-sized circuits.

According to the table, TimberWolf 5.4 is able to produce extremely high-quality placements, but requires a large amount of processor time. The blank entries in the table for circuit *z3* are due to the fact that we were unable to place the specific circuit using TimberWolf 5.4. Although the solution quality of our algorithm is less than TimberWolf5.4 for these example circuits, we feel that our approach has a number of benefits. First of all, the execution time for a combined placement and routing algorithm is nearly an order of magnitude less than TimberWolf when run in the uniprocessor mode. Second, there are a number of enhancements that can be made to our algorithm and implementation to improve the quality of the results. Suaris and Kedem [16] have already demonstrated that the quadrisection approach to cell placement is very competitive with simulated annealing techniques and, with modifications to our implementation and the enhancements we have proposed, we expect to achieve similar results. Third, in any hierarchical routing algorithm, under-estimations or over-estimations of the available routing resources at the topmost hierarchy levels can adversely affect the ability of the router to achieve good results at lower levels of the hierarchy. Addressing this problem and making changes in the algorithm for assigning the linear program variables to each net configuration should provide a fair improvement in the global routing algorithm, and in our results. Fourth, hierarchical techniques are well-suited for larger and larger circuits of the future and have been used throughout our algorithm. And finally, as a result of the decomposition methods we employed, we have been able to develop a parallel algorithm for placement and routing. Through this parallelism, we are able to reduce the runtime further.

4.7.8. Process efficiency

One measure of how well the parallel processes are utilized is the process efficiency. The efficiency of a process is defined to be the ratio of time spent solving the problem over the total amount of time the process was dedicated to the problem. We measured the amount of time each process was spending executing the algorithm and the amount of time the process was spending waiting for other processes to finish their tasks. Moments of waiting occur primarily at two places. The first is during the topmost decomposition levels in which the number of parallel tasks available is less than the number of processors available. The second place is at the barrier synchronizations between operations. If processor loads are not balanced, one task may hold up the rest if it takes longer to complete. Table 4.7 gives some runtime data on the efficiency of various numbers of processes for the example circuits. In this table, the minimum, maximum, and average process efficiency values are listed. The wide ranges of maximum to minimum efficiency is due primarily to the top-level decomposition steps.

Figure 4.16 shows the effect of the top-level decomposition steps on parallel performance. The data for this figure were taken from uniprocessor and 8-processor executions of the parallel placement and routing algorithm. The figure plots the percentage of the total execution time spent at each level of the hierarchy. The plot of the uniprocessor numbers show that due to the rapid expansion of the execution tree, the majority of execution time is spent near the bottom levels of the decomposition. However, the

Table 4.7. Process efficiency measurements

Circuit	Number of Processes	Process Efficiency		
		Min.	Max.	Ave.
z1	2	0.740	0.946	0.84
z1	4	0.501	0.867	0.63
z1	8	0.242	0.819	0.39
z2	2	0.911	0.981	0.95
z2	4	0.796	0.955	0.85
z2	8	0.549	0.799	0.66
z3	2	0.835	0.969	0.91
z3	4	0.614	0.789	0.73
z3	8	0.336	0.753	0.47
z4	2	0.754	0.979	0.86
z4	4	0.636	0.952	0.76
z4	8	0.379	0.861	0.51
p1	2	0.853	0.980	0.91
p1	4	0.725	0.908	0.79
p1	8	0.442	0.816	0.56
p2	2	0.884	0.915	0.90
p2	4	0.684	0.923	0.79
p2	8	0.423	0.782	0.56

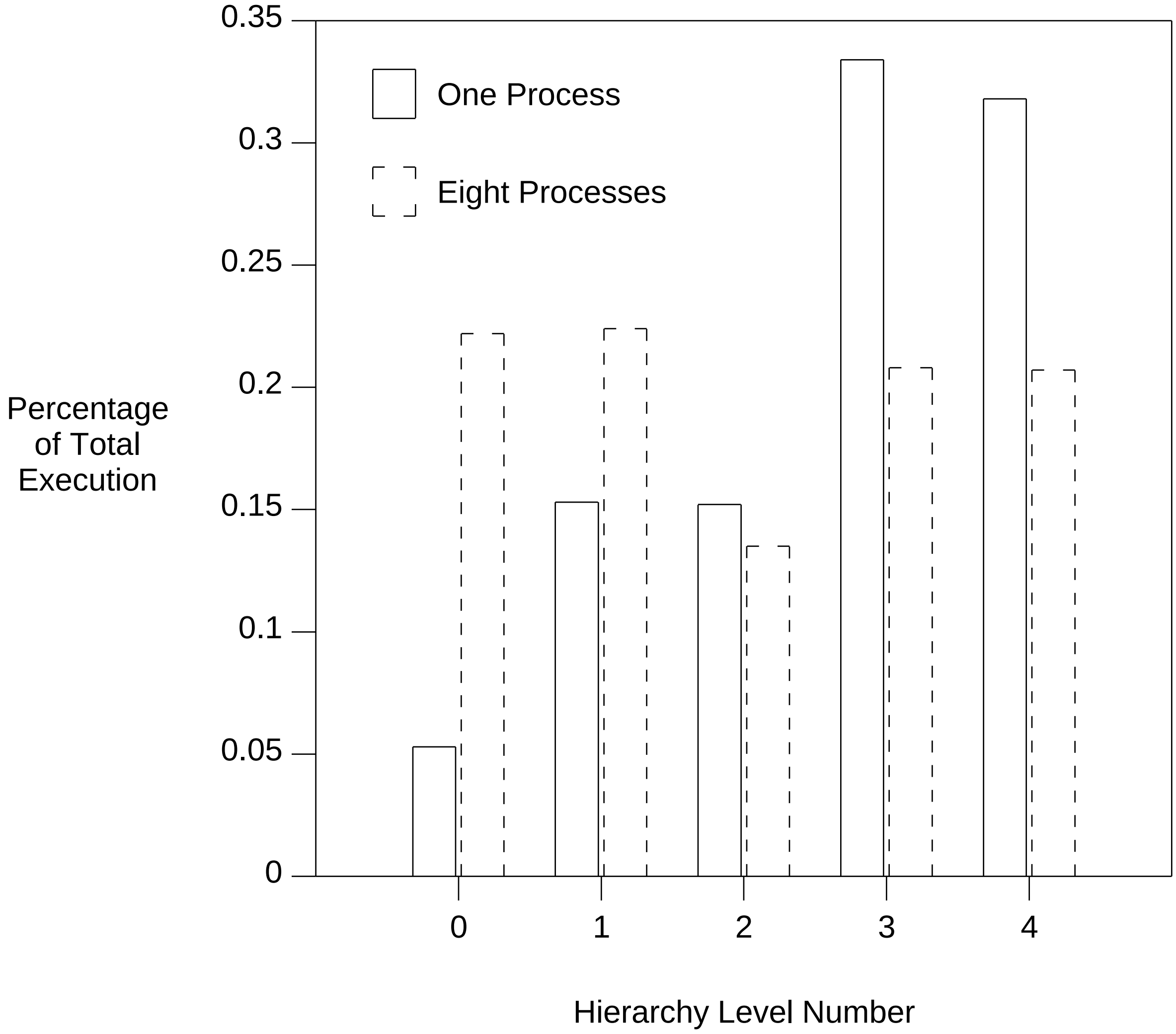


Figure 4.17. Execution time percentages

parallel processor numbers show that the top-level evaluations make up a large percentage of the execution time when the parallelisms at the lower decomposition levels are exercised. Although the top-level evaluations make up only five percent of the execution time for the uniprocessor case, they make up more than 20 percent of the execution time for eight processes; as the number of processes increases, the percentage will continue to grow.

4.7.9. Speedup evaluation

Table 4.8 provides information on the attainable speedups for the example circuits using a variable number of processors. The low speedup values in Table 4.8 are expected from the data presented in Table 4.7 and Figure 4.17. To improve the speedup and processor efficiency values for large numbers of processors, it is important to eliminate load imbalances among processors and to partition the tasks in the top-level decomposition steps into subtasks that may be executed in parallel. Another area of

Table 4.8. Speedup measurements

Circuit	Number of Processes	Runtime (s)	Speedup
z1	1	124	1.0
z1	2	69	1.8
z1	4	47	2.6
z1	8	37	3.4
z2	1	1648	1.0
z2	2	1016	1.6
z2	4	721	2.3
z2	8	599	2.8
z3	1	3062	1.0
z3	2	1900	1.6
z3	4	1318	2.3
z3	8	1078	2.8
z4	1	4215	1.0
z4	2	2676	1.6
z4	4	1811	2.3
z4	8	1496	2.8
p1	1	270	1.0
p1	2	160	1.7
p1	4	109	2.5
p1	8	95	2.8
p2	1	3988	1.0
p2	2	2655	1.5
p2	4	1857	2.1
p2	8	1555	2.6

further investigation is the effect of eliminating the scheduling barriers so that processes need not wait for other processes to finish before continuing execution.

CHAPTER 5.

CONCLUSIONS

5.1. Contributions

In this thesis we have presented a new parallel algorithm for global routing and a new parallel algorithm for simultaneous placement and routing which incorporates the first algorithm. We have demonstrated an algorithm for global routing which is not only fast and efficient, but also readily parallelizable. We have shown high processor utilization on a shared-memory multiprocessor and results that are competitive with well-known global routing programs.

We have also presented a new algorithm for simultaneous placement and global routing that is extremely well-suited for parallel processing. We have discussed enhancements over existing placement and routing algorithms and have demonstrated their effectiveness. Furthermore, we have verified the parallel properties of our algorithm with an implementation on a shared-memory multiprocessor.

5.2. Future Directions

In this thesis, we have laid the groundwork for further research into the placement and routing problems and methods for developing parallel algorithms to solve these problems. Our focus in this research has been to develop a hierarchical decomposition scheme so that the subproblems are completely independent of each other and can be evaluated in parallel. We have found that at the higher levels of any decomposition scheme, it may be necessary to develop ways to partition the relatively few tasks so that all processing resources may be fully utilized. To achieve this parallelism at the top levels, the algorithm may have to allow for concurrent evaluation of interdependent tasks.

We feel there are a number of enhancements and extensions to our algorithm which will provide substantial improvements in the quality of our results. Modifications to the capacity estimation algorithms and the methods used to make the net assignments following the linear program solution in the global router should improve the quality of the final routing. A bottom-up placement and routing adjustment phase can also be employed which (in parallel) considers small regions for local improvements. Following the local improvements, small regions are merged and the local improvement phase is repeated for the larger regions. This process would be repeated until the region includes the entire layout.

Another direction of interest is to evaluate the algorithmic changes necessary for the solution of placement and routing problems for other design styles such as Macro Cell and Sea-of-Gates. Furthermore, a more completely interfaced final routing should be developed to complete the parallel package. The current implementation of the algorithms is intended for shared-memory multiprocessors. There are a number of issues to consider for implementation on different parallel architectures (e.g., message-passing multiprocessors and networks of workstations).

REFERENCES

[1] P. Banerjee, ‘‘The use of parallel processing in VLSI computer-aided design applications,’’ ICCAD-88 Tutorial, also Tech. Rep. no. CSG-104, Coordinated Science Laboratory, Univ. of Illinois, Urbana, IL, May 1988.

[2] R. Jayaraman and R. A. Rutenbar, ‘‘Floorplanning by annealing on a hypercube multiprocessor,’’ *Proc. Int. Conf. Computer-Aided Design*, pp. 346-349, Nov. 1987.

[3] K. P. Belkhale and P. Banerjee, ‘‘PACE2: An improved parallel VLSI extractor with parametric extraction,’’ *Proc. Int. Conf. Computer-Aided Design*, pp. 526-530, Nov. 1989.

[4] B. Tonkin, ‘‘Circuit extraction on a message-based multiprocessor,’’ *Proc. 27th Design Automat. Conf.*, pp. 260-265, June 1990.

[5] G. G. Hung, Y. C. Wen, K. Gallivan, and R. Saleh, ‘‘Parallel circuit simulation using hierarchical relaxation,’’ *Proc. 27th Design Automat. Conf.*, pp. 394-399, June 1990.

[6] G. C. Yang, ‘‘PARASPICE: A parallel circuit simulator for shared-memory multiprocessors,’’ *Proc. 27th Design Automat. Conf.*, pp. 400-405, June 1990.

[7] K. Subramanian and M. R. Zargham, ‘‘Distributed and parallel demand driven logic simulation,’’ *Proc. 27th Design Automat. Conf.*, pp. 485-490, June 1990.

[8] S. Patil and P. Banerjee, ‘‘A parallel branch and bound approach to test generation,’’ *IEEE Trans. Computer-Aided Design*, vol. 9, no. 3, pp. 313-322, Mar. 1990.

[9] T. Blank, ‘‘A survey of hardware accelerators used in computer-aided design,’’ *IEEE Design Test*, pp. 21-39, Aug. 1984.

[10] B. T. Preas and P. G. Karger, ‘‘Automatic placement: A review of current techniques,’’ *Proc. 23rd Design Automat. Conf.*, pp. 622-629, June 1986.

[11] M. R. Hartoog, ‘‘Analysis of placement procedures for VLSI standard cell layout,’’ *Proc. 23rd Design Automat. Conf.*, pp. 314-319, June 1986.

[12] M. Hanan and J. M. Kurtzberg, ‘‘Placement techniques,’’ in *Design Automation of Digital Systems: Theory and Techniques*. M. A. Breuer, Ed., Prentice-Hall, 1972, pp. 213-282.

[13] M. A. Breuer, ‘‘Min-cut placement,’’ *J. Design Automat. Fault Tol. Comp.*, vol. 1, pp. 343-382, Oct. 1977.

[14] B. W. Kernighan and S. Lin, ‘‘An efficient heuristic for partitioning graphs,’’ *Bell Syst. Tech. J.*, vol. 49, pp. 291-307, Feb. 1970.

[15] A. E. Dunlop and B. W. Kernighan, ''A procedure for placement of standard cell VLSI circuits,'' *IEEE Trans. Computer-Aided Design*, vol. CAD-4, no. 1, pp. 92-98, Jan. 1985.

[16] P. R. Suaris and G. Kedem, ''An algorithm for quadrisection and its application to standard cell placement,'' *IEEE Trans. Circuits Syst.*, vol. 35, no. 3, pp. 294-303, Mar. 1988.

[17] J. P. Blanks, ''Near-optimal placement using a quadratic objective function,'' *Proc. 21st Design Automat. Conf.*, pp. 602-615, June 1985.

[18] C. K. Cheng and E. S. Kuh, ''Module placement based on resistive network optimization,'' *IEEE Trans. Computer-Aided Design*, vol. CAD-3, no. 3, pp. 218-225, July 1984.

[19] Y. H. Hu and S. J. Chen, ''GM_Plan: A gate matrix layout algorithm based on artificial intelligence planning techniques,'' *IEEE Trans. Computer-Aided Design*, vol. 9, no. 8, pp. 836-845, Aug. 1990.

[20] R. S. Tsay, E. S. Kuh, and C. P. Hsu, ''PROUD: A fast sea-of-gates placement algorithm,'' *Proc. 25th Design Automat. Conf.*, pp. 318-323, June 1988.

[21] C. Sechen, *VLSI Placement and Global Routing using Simulated Annealing*. Boston: Kluwer Academic Publishers, 1988.

[22] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, ''Optimization by simulated annealing,'' *Science*, vol. 220, pp. 671-680, May 1983.

[23] P. Siarry, L. Bergonzi, and G. Dreyfus, ''Thermodynamic optimization of block placement,'' *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no. 2, pp. 211-221, Mar. 1987.

[24] L. K. Grover, ''Standard cell placement using simulated sintering,'' *Proc. 24th Design Automat. Conf.*, pp. 56-59, June 1987.

[25] R. M. Kling and P. Banerjee, ''ESP: Placement by simulated evolution,'' *IEEE Trans. Computer-Aided Design*, vol. CAD-8, no. 2, pp. 245-256, Mar. 1989.

[26] R. M. Kling, ''Optimization by simulated evolution and its application to cell placement,'' Tech. Rep. no. CRHC-90-7, Coordinated Science Laboratory, Univ. of Illinois, Urbana, IL, Aug. 1990.

[27] P. Banerjee, M. H. Jones, and J. S. Sargent, ''Parallel simulated annealing algorithms for standard cell placement on hypercube multiprocessors,'' *IEEE Trans. Parallel and Dist. Syst.*, vol. 1, no. 1, pp. 91-106, Jan. 1990.

[28] J. Sargent and P. Banerjee, ''A parallel row-based algorithm for standard cell placement with integrated error control,'' *Proc. 26th Design Automat. Conf.*, pp. 590-593, June 1989.

[29] C. P. Ravikumar and S. Sastry, ''Parallel placement on hypercube architecture,'' *Proc. Int. Conf. Parallel Process., Vol. III*, pp. 97-101, 1989.

[30] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, ''A parallel simulated annealing algorithm for the placement of macro-cells,'' *Proc. Int. Conf. Computer-Aided Design*, pp. 30-33, Nov. 1986.

[31] S. A. Kravitz and R. A. Rutenbar, ''Placement by simulated annealing on a multiprocessor,'' *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no 4, pp. 534-549, June 1987.

[32] A. Casotto and A. Sangiovanni-Vincentelli, ''Placement of standard cells using simulated annealing on the Connection Machine,'' *Proc. Int. Conf. Computer-Aided Design*, pp. 350-353, 1987.

[33] C. P. Wong and R. D. Fiebrich, ''Simulated annealing-based circuit placement algorithm on The Connection Machine System,'' *Proc. Int. Conf. Computer Design*, pp. 78-82, 1987.

[34] J. S. Rose, D. R. Blythe, W. M. Snelgrove, and Z. G. Vranesic, ''Fast, high quality VLSI placement on a MIMD multiprocessor,'' *Proc. Int. Conf. Computer-Aided Design*, pp. 42-45, Nov. 1986.

[35] K. Ueda, T. Komatsubara, and T. Hosaka, ''A parallel processing approach for logic module placement,'' *IEEE Trans. Computer-Aided Design*, vol. CAD-2, no. 1, pp. 39-47, Jan. 1983.

[36] R. M. Kling and P. Banerjee, ''Concurrent ESP: A placement algorithm for execution on distributed processors,'' *Proc. Int. Conf. Computer-Aided Design*, pp. 354-357, 1987.

[37] K. W. Lee and C. Sechen, ''A new global router for row-based layout,'' *Proc. Int. Conf. Computer-Aided Design*, pp. 180-183, Nov. 1988.

[38] J. Cong and B. Preas, ''A new algorithm for standard cell global routing,'' *Proc. Int. Conf. Computer-Aided Design*, pp. 176-179, Nov. 1988.

[39] G. Meixner and U. Lauther, ''A new global router based on a flow model and linear assignment,'' *Proc. Int. Conf. Computer-Aided Design*, pp. 44-47, Nov. 1990.

[40] R. Nair, ''A simple yet effective technique for global wiring,'' *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no. 2, pp. 165-172, Mar. 1987.

[41] M. P. Vecchi and S. Kirkpatrick, ''Global wiring by simulated annealing,'' *IEEE Trans. Comput.*, vol. 7, no. 4, pp. 215-222, Oct. 1983.

[42] N. Hasan and C. L. Liu, ''A force-directed global router,'' *Proc. Stanford Conf. Advanced Research in VLSI*, pp. 135-150, 1987.

[43] C. D. Hechtman and J. J. Lewandowski, ''A flux directed approach to a wire routing problem,'' *IEEE VLSI Tech. Bull.*, vol. 4, no. 3/4, pp. 124-138, Sept./Dec. 1989.

[44] M. Burstein and R. Pelavin, ''Hierarchical wire routing,'' *IEEE Trans. Computer-Aided Design*, vol. CAD-2, no. 4, pp. 223-234, Oct. 1983.

[45] M. Marek-Sadowska, ''Global router for gate array,'' *Proc. Int. Conf. Computer Design*, pp. 332-337, Oct. 1984.

[46] W. K. Luk, D. T. Tang, and C. K. Wong, ''Hierarchical global wiring for custom chip design,'' *Proc. 23rd Design Automat. Conf.*, pp. 481-489, June 1986.

[47] R. Nair, S. J. Hong, S. Liles, and R. Villani, ''Global wiring on a wire routing machine,'' *Proc. 19th Design Automat. Conf.*, pp. 224-231, June 1982.

[48] T. Watanabe, H. Kitazawa, and Y. Sugiyama, ''A parallel adaptable routing algorithm and its implementation on a two-dimensional array processor,'' *IEEE Trans. Computer-Aided Design*, vol. CAD-6, No 2, pp. 241-250, Mar. 1987.

[49] O. A. Olukotun and T. N. Mudge, ''A preliminary investigation into parallel routing on a hypercube computer,'' *Proc. 24th Design Automat. Conf.*, pp. 814-820, June 1987.

[50] Y. Won and S. Sahni, ''Maze routing on a hypercube multiprocessor computer,'' *Proc. Int. Conf. Parallel Process.*, pp. 630-637, Aug. 1987.

[51] Jonathan Rose, ''LocusRoute: A parallel global router for standard cells,'' *Proc. 25th Design Automat. Conf.*, pp. 189-195, June 1988.

[52] R. J. Brouwer and P. Banerjee, ''PHIGURE: A Parallel Hierarchical Global Router,'' *Proc. 27th Design Automat. Conf.*, pp. 650-653, June 1990.

[53] A. A. Szepieniec, ''Integrated placement/routing in sliced layouts,'' *Proc. Design Automat. Conf.*, pp. 300-307, 1986.

[54] E. Shragowitz, J. Lee, and S. Sahni, ''Placer-router for sea-of-gates design style,'' *Proc. Int. Conf. Computer Design*, pp. 330-335, 1987.

[55] W. M. Dai and E. S. Kuh, ''Simultaneous floor planning and global routing for hierarchical Building-Block Layout,'' *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no. 5, pp. 828-837, Sept. 1987.

[56] M. Igusa, M. Beardslee, and A. Sangiovanni-Vincentelli, ''ORCA: A sea-of-gates place and route system,'' *Proc. 26th Design Automat. Conf.*, pp. 122-127, June 1989.

[57] P. R. Suaris and G. Kedem, ''A quadrisection-based combined place and route scheme for standard cells,'' *IEEE Trans. Computer-Aided Design*, vol. 8, no. 3, pp. 234-244, Mar. 1989.

[58] R. S. Garfinkel and G. L. Nemhauser, *Integer Programming*. New York, NY: John Wiley and Sons, Inc., 1972, pp. 154-165.

[59] C. M. Fiduccia and R. M. Mattheyses, ''A linear-time heuristic for improving network partitions,'' *Proc. 19th Design Automat. Conf.*, pp. 175-181, 1982.

VITA

Randall Brouwer received the B.S. degree in Engineering from Calvin College, Grand Rapids, Michigan in 1985. He received the M.S. degree in Electrical Engineering from the University of Illinois, Urbana, Illinois in 1988. He is currently a candidate for the Ph.D. degree in Electrical Engineering at the University of Illinois, Urbana, Illinois.

From 1984 to 1985, Mr. Brouwer was a laboratory assistant for the Engineering Department at Calvin College. During the summer of 1985, he worked at Smith's Industries in Grand Rapids, MI (formerly Lear Siegler, Inc.), developing software for testing memory boards. During the Spring of 1985, Mr. Brouwer worked as a teaching assistant for the Department of Electrical Engineering at the University of Illinois. From the Fall of 1985 to the present, he has been working as a research assistant in the Coordinated Science Laboratory at the University of Illinois.

His research interests include Computer-Aided Design of Integrated Circuits, parallel processing and the development of parallel algorithms for various applications, and high-performance multiprocessor systems.

PARALLEL ALGORITHMS FOR PLACEMENT AND ROUTING IN VLSI DESIGN

Randall J. Brouwer, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois, Urbana-Champaign, 1991

The computational requirements for high quality synthesis, analysis, and verification of VLSI designs have rapidly increased with the fast growing complexity of these designs. Research in the past has focused on the development of heuristic algorithms, special purpose hardware accelerators, or parallel algorithms for the numerous design tasks to decrease the time required for solution. In this thesis, we propose two new parallel algorithms for two VLSI synthesis tasks, standard cell placement and global routing.

The first algorithm, a parallel algorithm for global routing, uses hierarchical techniques to decompose the routing problem into independent routing subproblems that are solved in parallel. Results are then presented which compare the routing quality to the results of other published global routers and which evaluate the speedups attained.

The second algorithm, a parallel algorithm for cell placement and global routing, hierarchically integrates a quadrisection placement algorithm, a bisection placement algorithm, and the previous global routing algorithm. Unique partitioning techniques are used to decompose the various stages of the algorithm into independent tasks which can be evaluated in parallel. Finally, we present results which evaluate the various algorithm alternatives and compare the algorithm performance to other placement programs, and we present measurements on the parallel speedups available.

DEDICATION

TO: Janine

ACKNOWLEDGEMENTS

I would like to thank most of all my advisor, Professor Prithviraj Banerjee, for his constant advice, support, and encouragement throughout my work on this project. I would also like to thank the other members of my thesis committee for not only their comments and suggestions on my work, but also their flexibility in scheduling the various examinations. I am thankful to the professors in the Center for Reliable and High-Performance Computing for the excellent facilities to which I have had access.

There are too many other students and staff members with whom I have worked that have been great sources of help, encouragement, and enjoyment to list here, but I would like to specifically mention my past and present officemates who survived sharing an office with me and let me bounce ideas off them: Ralph Kling, A.L.N. Reddy, Mike Peercy, and Kaushik De.

Finally, I would like to express my appreciation to my wife, Janine, for her endless love and support, and my parents, in-laws, relatives, and friends for all of their encouragement throughout my studies.

TABLE OF CONTENTS

CHAPTER PAGE

1. INTRODUCTION 1

    1.1. Parallel Processing for CAD 1

    1.2. Parallel Processing Architectures 3

    1.3. Thesis Outline 5

2. CELL PLACEMENT AND GLOBAL ROUTING PROBLEMS 7

    2.1. Introduction 7

    2.2. Row-Based Layouts 7

    2.3. Uniprocessor Cell Placement Algorithms 9

    2.4. Parallel Cell Placement Algorithms 11

    2.5. Uniprocessor Global Routing Algorithms 14

    2.6. Parallel Global Routing Algorithms 16

    2.7. Combined Placement and Routing Algorithms 17

    2.8. Parallel Combined Placement and Routing Algorithm 19

3. PARALLEL GLOBAL ROUTING 20

    3.1. Global Routing Model 20

        3.1.1. Feedthrough insertion and channel width expansion 25

        3.1.2. Hierarchical decomposition 26

            3.1.2.1. Maximal boundary determination 26

3.1.2.2. Minimal boundary determination 28

3.2. Parallel Algorithm Overview 31

3.2.1. Exploitation of coarse-grained parallelism 32

3.2.1.1. Maximal boundary determination 32

3.2.1.2. Minimal boundary determination 33

3.2.2. Exploitation of fine-grained parallelism 35

3.2.3. Task complexity 37

3.2.4. Experimental results on 2X2 routing task complexity 39

3.3. Implementation 42

3.4. Results 44

3.5. Conclusions 46

4. PARALLEL PLACEMENT AND ROUTING 48

4.1. Overview 48

4.2. Floorplanning Step 49

4.3. Placement and Routing 50

4.3.1. Quadrisection-based placement 51

4.3.2. Routing of the quadrisection 60

4.3.3. Initial placement for quadrisection 61

4.3.3.1. The X-dimension restricted global bisection 61

4.3.3.2. The Y-dimension restricted global bisection 64

4.3.3.3. Combining X- and Y-dimension bisectioning 64

4.3.4. Two-by-N global routing 66

4.4. Algorithm Outline 66

4.5. Detailed Routing 72

4.6. Parallelisms and Algorithm Complexities 72

4.6.1. Complexity evaluation 73

4.7. Results 77

4.7.1. Implementation 77

4.7.2. Benchmark circuits 80

4.7.3. Evaluation of net cost function 80

4.7.4. Evaluation of initial placement by global bisection 82

4.7.5. Evaluation of cell swapping 82

4.7.6. Route-based placement evaluation 83

4.7.7. Comparison to TimberWolf 5.4 84

4.7.8. Process efficiency 86

4.7.9. Speedup evaluation 89

5. CONCLUSIONS 91

5.1. Contributions 91

5.2. Future Directions 91

REFERENCES 93

VITA 97

LIST OF TABLES

Table Page

3.1. Routing quality comparison 44

3.2. Uniprocessor runtime comparison 45

3.3. Parallel algorithm results 46

4.1. Benchmark statistics 80

4.2. Comparison of net cost parameters 81

4.3. Initial placement alternatives comparison 82

4.4. Effect of cell swapping 83

4.5. Route-based vs. standard cost functions 83

4.6. Comparison to TimberWolf 84

4.7. Process efficiency measurements 87

4.8. Speedup measurements 89

LIST OF FIGURES

Figure Page

1.1. The Hipercad Project overview 2

1.2. Multiple Instruction-Multiple Data multiprocessors 3

1.3. Distributed-memory multiprocessor 4

1.4. Shared-memory multiprocessor 4

1.5. Single Instruction-Multiple Data multiprocessors 5

2.1. Example of a gate array design 8

2.2. Example of a standard cell design 10

2.3. Wire length estimation by bounding box 11

2.4. Cell congestion estimation by net cut count 12

2.5. The global routing model 14

3.1. Routing block model 20

3.2. (a) Axes capacities of 2x2 bin array (b) Example 22

3.3. Net types and possible routings 23

3.4. Maximal boundary determination 27

3.5. Example of maximal boundary determination 29

3.6. Minimal boundary determination 30

3.7. Plot of projected speedup vs. number of processes 35

3.8. Percentage of tasks in startup phase 36

3.9. Net setup time vs. iteration number 39
3.10. LP solution time vs. iteration number 40
3.11. Net assignment time vs. iteration number 41
3.12. Total time vs. iteration number 42
3.13. Parallel global routing flowchart 43
4.1. Overview of the placement and routing algorithm 49
4.2. Determination of layout block array 50
4.3. Min-cut partitioning 51
4.4. Quadrisection-based partitioning 53
4.5. Partition and cut lines for different quadrisection levels 54
4.6. Simple quadrisection net cost function 55
4.7. Improved quadrisection net cost function 56
4.8. Quadrisection gain tables 57
4.9. Gain table data structure 58
4.10. Partitioning for X-dimension restricted global bisection 62
4.11. Bisection cost function example 63
4.12. Y-dimension restricted global bisection 65
4.13. Two-by-N routing of a bisection region 67
4.14. Placement and routing decomposition 68
4.15. Decomposition example 70
4.16. Parallel placement and routing pseudocode 79
4.17. Execution time percentages 88