VLSI Design
1994, Vol. 2, No. 2, pp. 143–156
Reprints available directly from the publisher
Photocopying permitted by license only

Area Optimization of Slicing Floorplans in Parallel

CHENG-HSI CHEN
Dept. of Computer Science, The University of Texas at Dallas, P.O. Box 830688, EC31, Richardson, TX 75083-0688 U.S.A.

IOANNIS G. TOLLIS
Dept. of Computer Science, The University of Texas at Dallas, P.O. Box 830688, EC 31, Richardson, TX 75083-0688, U.S.A.

We first present a parallel algorithm for finding the optimal implementations for the modules of a slicing floorplan that respects a given slicing tree. The algorithm runs in $O(n)$ time and requires $O(n)$ processors, where n is the number of modules. It is based on a new $O(n^2)$ sequential algorithm for solving the above problem. We then present a parallel algorithm for finding a set of optimal implementations for a slicing floorplan whose corresponding slicing tree has height $O(\log n)$. This algorithm runs in $O(n)$ time using $O(\log n)$ processors. Our parallel algorithms do not need shared memory and can be implemented in a distributed system.

Key Words: *floorplan, parallel, slicing, optimization, area, useful.*

1 INTRODUCTION

Floorplan design is the first task in VLSI layout and perhaps the most important one. It is the problem of allocating space to a set of modules on the chip in order to minimize the area of the chip. A chip is a floor rectangle with the additional information about the relative positions of basic modules (circuits) such as registers, ALU, etc. The target of floorplanning is to partition the floor rectangle into smaller ones, called *basic rectangles,* and embed the basic modules into these small rectangles preserving the relative positions of the modules [7, 8]. A module is called *rigid* if its dimensions are given, otherwise it is called *flexible*. In this paper we assume that all modules are flexible. This flexibility allows the designer to manipulate the structure of the modules during floorplanning. For each module we are given a list of pairs (height, width), called *implementations*. Given the relative positions of the modules of a chip, we wish to find the best implementations of these modules, sometimes called *cells,* in order to minimize the total layout area of the chip. Notice that the objective function (layout area) to be minimized is nondecreasing. This problem has received considerable attention recently [1–2, 4–10].

A *floorplan* is a partition of the floor rectangle using vertical and horizontal line segments called *slices*. A floorplan is *slicing* if it is either a basic rectangle or there is a slice that partitions the enclosing rectangle into two slicing floorplans, see Figure 1. There are two ways to represent a slicing floorplan: (a) using series-parallel graphs [4], and (b) using a *slicing tree* [6]. A slicing tree T is a rooted binary tree that gives the natural hierarchical description of a slicing flooplan. Each nonleaf node of T is labeled either "H" or "V" specifying whether the corresponding slice is horizontal or vertical. Each leaf corresponds to a basic rectangle. In general, there are many slicing trees that describe a given slicing floorplan. Notice that a slicing tree with n leaves has $2n - 1$ nodes.

If each basic rectangle has c implementations, where c is a constant, then there are $O(c^n)$ possible sets of implementations for the floor rectangle. Stockmeyer [6] presented an algorithm for finding a set of optimal implementations of the cells of a slicing floorplan that requires that $O(n^2)$ time in the worst case. In fact, the algorithm runs in time $O(n \times l)$, where l is the height of the slicing tree. At the beginning, each leaf of the slicing tree has two pairs, corresponding to the two possible implementations

*Research supported in part by the Texas Advanced Research Program under Grant No. 3972.

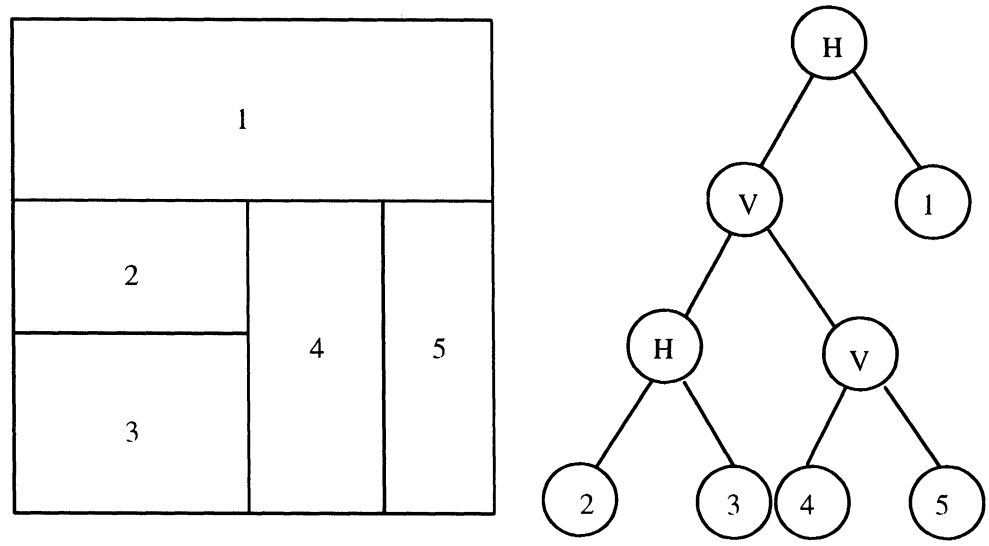

FIGURE 1 A slicing floorplan and its corresponding slicing tree.

of the basic rectangle. At each node, the lists of the children are merged in order to produce a new list containing all the implementations of the basic rectangles in the subtree that could give minimum total area. If the node is labeled "H" (resp. "V") then the lists of its children have to be sorted in decreasing order of widths (resp. heights), and the generated list (of the parent node) is also sorted in decreasing order of widths (resp. height). Therefore, if a parent node whose label is not compatible with the labels of its children, the lists of the children will not have the order that the parent node requires. In this case, the algorithm has to reverse them. This fact makes the algorithm hard to parallelize.

In this paper, we assume that each basic rectangle has c implementations, where c is a constant. We first present a new sequential algorithm that runs in $O(n^2)$ time and eliminates the need for reversing the lists when they are incompatible; therefore it is easily parallelizable. The main idea of the parallel algorithm is to use a processor for each nonleaf node of the slicing tree and propagate the pairs of the lists from the children to the parents in a pipeline fashion. The algorithm runs in $O(n)$ time and requires $O(n)$ processors (i.e., we achieve optimal speedup). However, there are slicing floorplans whose corresponding slicing trees have height $O(\log n)$. Such slicing floorplans are rather important, because in practice the slicing floorplans are obtained by recursively using circuit bipartitioning techniques. Hence, in most practical applications, the height of the slicing trees is $O(\log n)$. In the second part of this paper, we present a parallel algorithm to compute a set of optimal implementations (one for each basic rectangle) of such a floorplan in $O(n)$ time, using $O(\log n)$ processors. Moreover, our parallel algorithms do not need shared memory and can be implemented in a distributed system.

2 SLICING TREES AND FLOORPLANNING

A *vertical slice* is a vertical line segment that is enclosed in some rectangle and cuts the rectangle into two smaller ones. Similarly, a *horizontal slice* is a horizontal line segment that cuts the rectangle into two smaller ones. For each basic rectangle, there is a list of possible implementations of the form $\{h_1 \times w_1, h_2 \times w_2, \ldots, h_m \times w_m\}$, where h_i is the height and w_i is the width of the rectangle.

As discussed above, every node of T corresponds to a rectangle, and there is a list of pairs associated with each node. These pairs are ordered by their heights (resp. widths) either in decreasing or in increasing order. If they are ordered in decreasing order of height (resp. width) then we say that height (resp. width) is the *major-value,* otherwise if they are ordered in increasing order of height (resp. width) then we say that height (resp. width) is the *minor-value*. We will see later that if a list has height (resp. width) as its major- (resp. minor-) value, then it has width (resp. height) as its minor- (resp. major-) value.

Procedure M_V merges a pair of two neighboring rectangles separated by a vertical slice into a larger rectangle. The new height is the greater height of the two and the new width is the sum of the widths.

Suppose (h_1, w_1) and (h_2, w_2) denote implementations of the two rectangles. The new rectangle is $(\max\{h_1, h_2\}, w_1 + w_2)$. *Procedure M_H* is defined similarly. If the slice is horizontal, the new rectangle is $(h_1 + h_2, \max\{w_1, w_2\})$. Figure 2 shows examples of running Procedures M_V and M_H.

Let (h, w) be a pair in list L. We say that (h, w) is *useless* if there is a (h', w') in L such that $h \geq h'$ and $w \geq w'$; if there is no such (h', w') in L then (h, w) is called *useful*.

Lemma 1 Let L be the list of node u of a slicing tree T that contains all the useful pairs and no useless pairs. The major-value of L is height if and only if the minor-value of L is width.

Proof: ($\Rightarrow$) Suppose that the major-value of L is height but the minor-value of L is not width. Let (h_j, w_j) be the first pair of L that violates the increasing order of width. So there is a pair (h_i, w_i) in L such that $h_i > h_j$ and $w_i > w_j$. This implies that (h_i, w_i) is useless, a contradiction. ($\Leftarrow$) Similar to the previous proof. □

Let u be a node in T with children u_1 and u_2. Assume that u is labeled "H." Let $L_1 = \{(h_i, w_i) | 1 \leq i \leq k\}$ and $L_2 = \{(h'_j, w'_j) | 1 \leq j \leq m\}$ be the lists of u_1 and u_2 respectively. Let both L_1 and L_2 have height as their major-value. A pair (h'_i, w'_i), $1 \leq i < m$, in L_2 is *redundant* (with respect to L_1) if $w_1 \geq w'_{i+1}$. The pair (h'_i, w'_i) is redundant because if it is merged with any pair in L_1 the resulting pair is always useless. Next, we give two examples to explain the redundant pairs. First, let $L_1 = \{(h_1, w_1), (h_2, w_2)\}$, $L_2 = \{(h'_1, w'_1), (h'_2, w'_2)\}$. Since we know that the major-value of both lists is height, $h_1 > h_2$, $w_1 < w_2$, and $h'_1 > h'_2$, $w'_1 < w'_2$. Now assume that $w_1 \geq w'_2 > w'_1$. Consider the pairs $s = M\_H((h_1, w_1), (h'_1, w'_1)) = (h_1 + h'_1, w_1)$, and $s' = M\_H((h_1, w_1), (h'_2, w'_2)) = (h_1 + h'_2, w_1)$. Since s and s' are in the same list and $h'_2 < h'_1$, s is useless. For the same reason, M_H$((h_2, w_2), (h'_1, w'_1)$ also generates a useless pair. Hence the pair (h'_1, w'_1) is redundant. In the next example, we give numerical values to the variables: Let $L_1 = \{(8, 5), (5, 8)\}$, $L_2 = \{(4, 3), (3, 4)\}$. Note that the width of (4, 3) is less than the width of (8, 5). M_H((8, 5), (4, 3)) = (12, 5), M_H((5, 8), (4, 3)) = (9, 8). But M_H((8, 5), (3, 4)) = (11, 5), and M_H((5, 8), (3, 4)) = (8, 8). We can see clearly that (4, 3) merges with any pair of L_1 only generates a useless pair. Hence (4, 3) is redundant.

Procedure Merge_V will merge all the useful implementations of two rectangles separated by a vertical slice in order to obtain all the useful implementations of the larger rectangle. Let u be a node of T labeled "V," with children u_1 and u_2. Assume that L, L_1 and L_2 are the lists of u, u_1 and u_2 respectively, and the major-value of L_1 and L_2 is height, where $L_1 = \{(h_i, w_i) | 1 \leq i \leq m\}$ and $L_2 = \{(h'_j, w'_j) | 1 \leq j \leq k\}$.

Procedure Merge_V
(1)$i := 1$; $j := 1$;
(2)**while** $i \leq m$ and $j \leq k$ **do**

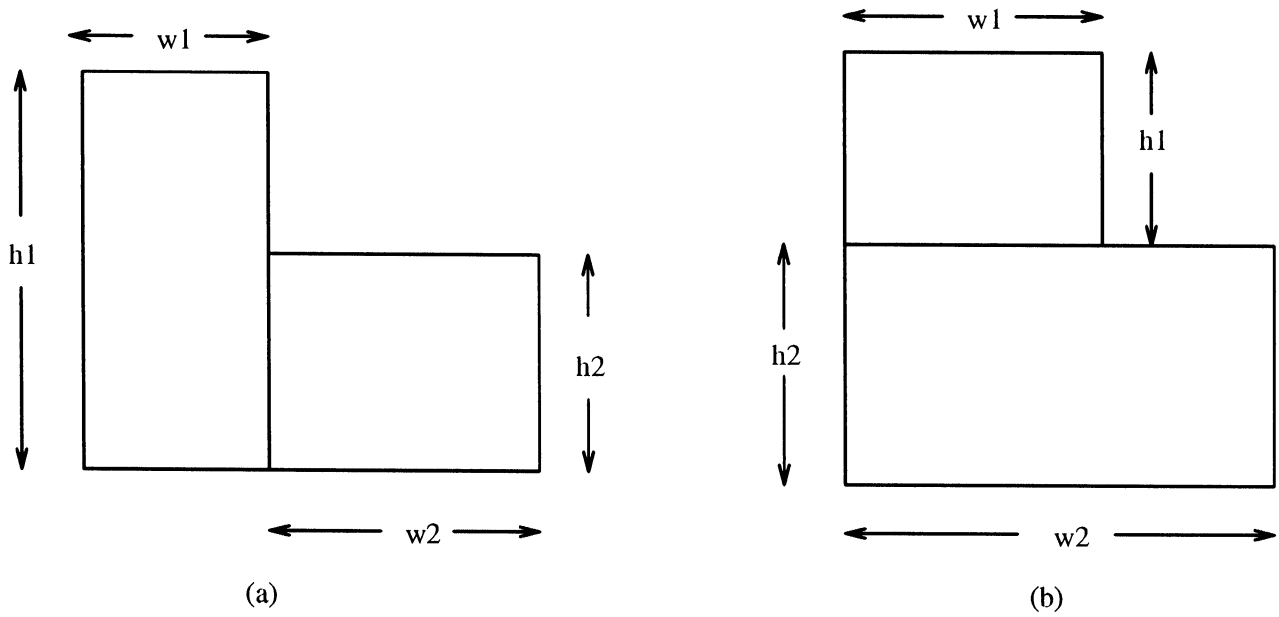

FIGURE 2 (a) M_V$((h_1, w_1), (h_2, w_2))$ (b) M_H$((h_1, w_1), (h_2, w_2))$.

(3)**begin**
(4)M_V{(h_i, w_i), (h_j', w_j')};
(5)**if** $h_i > h_j'$ **then** $i := i + 1$
(6)**else if** $h_i = h_j'$ **then begin**
(7) $i := i + 1$;
(8) $j := j + 1$;
(9) **end**
(10)**else** $j := j + 1$
(11)**end**;

Procedure *Merge_H* is symmetric to Procedure *Merge_V*. It merges all the useful implementations of two rectangles separated by a horizontal slice and obtains all the useful implementations of the enclosing rectangle. The parent node is labeled "H" and the major-value of L_1 and L_2 is width. These two procedures are variations of the ones described in [6] and are included here for completeness.

Lemma 2 Let u be a node of a slicing tree T with children u_1 and u_2. Assume that L_1 and L_2 are the lists of u_1 and u_2, respectively. If u is labeled "V" (resp. "H") and the major-value of L_1 and L_2 is height (width), then the list of u generated by Procedure Merge_V (Merge_H), L, has height (width) as its major-value and contains all the useful pairs and no useless pairs. Furthermore, both procedures run in $O(|L_1| + |L_2|)$ time, and produce at most $|L_1| + |L_2| - 1$ pairs.

Proof: First assume that u is labeled "V." Let L be the list generated by Procedure Merge_V. It is clear that all the pairs in L are ordered in decreasing order of their height and L contains no useless pairs. The reason is as follows: Procedure Merge_V starts merging the first pair of L_1 with the first pair of L_2. When two pairs from L_1 and L_2 are merged, if the height of one pair is greater than or equal to the height of the other pair, then the pair with the greater height is discarded (if the height of both pairs are equal, then both pairs are discarded) and the next pair is considered. This process continues until the pairs in one of the lists are exhausted. Since the pairs in L_1 and L_2 have height as their major-value, we have that L are also has height as its major-value. Moreover, the reason that L contains no useless pairs is that all the pairs of L_1 and L_2 are ordered in increasing order of their width. As the process goes on, the generated pairs have larger width. Hence all the pairs in L are useful and ordered in increasing order of their width.

Next we will show that L contains all the useful pairs. Suppose it does not. Then there must be a pair (h, w) such that (h, w) is useful but is not in L. Let (h, w) be generated by merging pairs (h_i, w_i) and (h_j', w_j') from L_1 and L_2, respectively. Since (h, w) is not in L, either (a) (h_i, w_i) merges with some pair of L_2 other than (h_j', w_j'), or (b) (h_i, w_i) does not merge with any pair of L_2.

(a) First, assume that (h_1, w_i) is considered before (h_j', w_j') in Procedure Merge_V. Let (h_t', w_t') be the last pair in L_2 that merges with (h_i, w_i). Also let $(h', w') = \text{M\_V}((h_i, w_i), (h_t', w_t')) = (\max\{h_i, h_t'\}, w_i + w_t')$ and $(h, w) = (\max\{h_i, h_j'\}, w_i + w_j')$. Since (h_t', w_t') is the last pair of L_2 that merged with (h_i, w_i), we have $h_i \geq h_t'$. But $h_t' > h_j'$, hence $h' \leq h$. Since $w_t' < w_j'$, we have $w' < w$. This implies that (h, w) is useless, a contradiction.

Next assume that (h_j', w_j') is considered before (h_i, w_i) in Procedure Merge_V. Similarly, if there is a pair (h_t, w_t) that is the last pair in L_1 that merged with (h_j', w_j'), and $h_t > h_i$, then (h, w) is again useless, also a contradiction.

(b) If (h_i, w_i) does not merge with any pair of L_2 then, when (h_i, w_i) is considered, all the pairs in L_2 are exhausted. Similar to (a), this contradicts the fact that (h, w) is useful.

From the above discussion, we conclude that L contains all the useful pairs and no useless pairs. Since Procedure Merge_V moves one pair down each time at least in one of L_1 and L_2, it needs $O(|L_1| + |L_2|)$ time to generate L. In addition, since a pair of L is generated by merging a pair in L_1 with a pair in L_2, there are at most $|L_1| + |L_2|$ pairs of L generated by Procedure Merge_V. In fact the number of pairs generated is $|L_1| + |L_2| - 1$ since the last two pairs will generate at most one pair.

Assume that u is labeled "H." Let the major-values of L_1 and L_2 be width. Symmetrically (to Procedure Merge_V), Procedure Merge_H merges L_1 with L_2 to generate L, where L has width as its major-value, contains all the useful pairs and no useless pairs. Moreover, symmetrical to Procedure Merge_V, Procedure Merge_H takes $O(|L_1| + |L_2|)$ time, generates at most $|L_1| + |L_2| - 1$ pairs. □

Let u be a node of T and u_1, u_2 be its two children. The list of u can be obtained by applying Procedure Merge_V or Procedure Merge_H on the lists of u_1 and u_2 depending on the label of u. If the labels of u_1 and/or u_2 are not compatible with the label of u, the lists of u_1 and/or u_2 need to be reversed in advance. We can compute the list of the root by merging the lists of its children that have been computed in this bottom-up fshion. Notice that every pair p, generated by procedures Merge_V and Merge_H, has two pointers that point to the two pairs that generated p. Finally, performing a linear scan of the final list we can determine the implementation that gen-

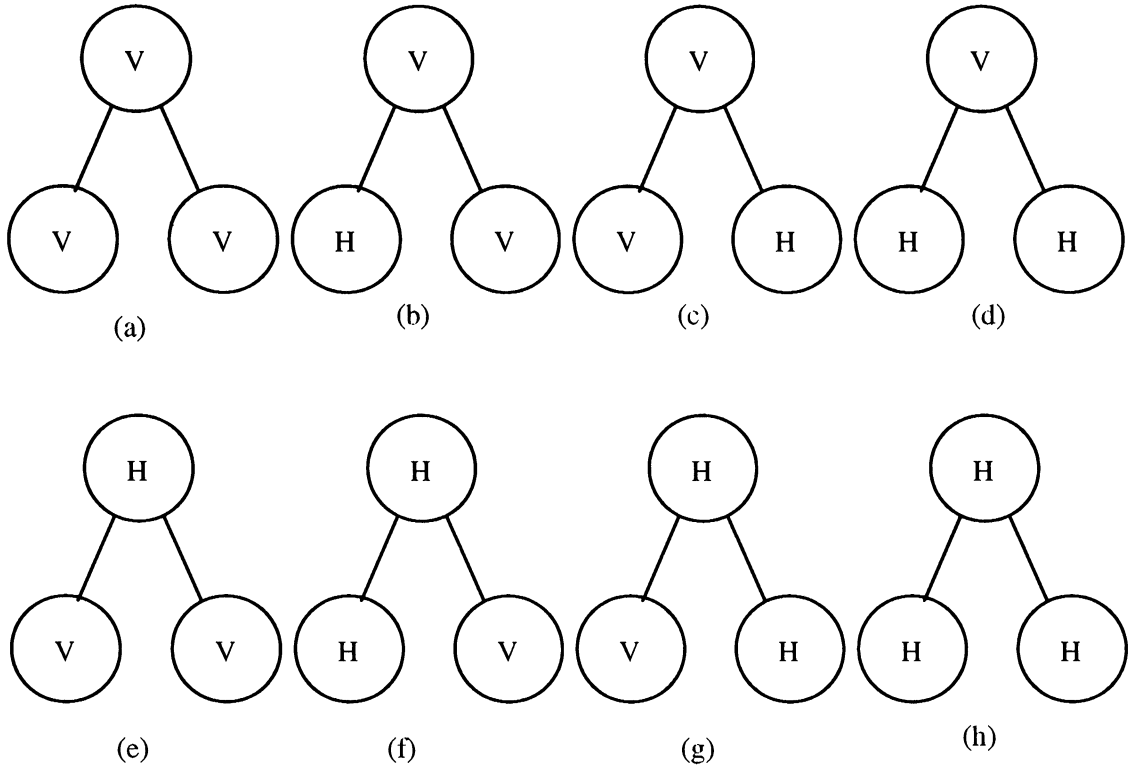

FIGURE 3 Eight different ways to label a node and its two children.

erates the minimum area of the floor rectangle, and using the pointers we can trace down to the leaf nodes and obtain the optimal implementations of the basic rectangles. There are eight different ways to label a node and its two children as shown in Figure 3. For the cases where the labels are not compatible (cases (c)–(f)), the list of the child (or both children) with different label has to be reversed. In order to reverse a list, we have to wait until all the pairs of the list are generated. Hence the list of the parent cannot be generated until both children's lists are completely generated. This makes the algorithm very hard to parallelize. Our new floorplanning algorithm does not need to reverse lists. The computation starts whenever there is a nonredundant pair available in each of the children's lists, and it continues until all the useful pairs of the parent are generated.

3 THE NEW FLOORPLANNING ALGORITHM

In this section we present a new procedure that generates the list of a parent without reversing the lists of its children. This procedure is perhaps the most important ingredient in parallelizing this floorplanning problem.

Suppose u is a node of T and u_1, u_2 are its two children. Let L_1 and L_2 be the lists of u_1 and u_2, respectively. Now let the lists of all the leaves have height as their major-value. For those sibling leaves whose parents are labeled "V," the lists are merged using Procedure Merge_V, described in the previous section. For those sibling leaves whose parents are labeled "H," we use Procedure Mg_H, described below. This ensures that all the lists of the nodes of T will have height as their major-value.

We first present the ideas underlying Mg_H informally. Let u, the parent of u_1 and u_2, be labeled "H" and L be the list that is generated by merging the lists L_1 and L_2. Suppose $p_i = (h_i, w_i)$ and $p'_j = (h'_j, w'_j)$ are pairs in L_1 and L_2, respectively, that will be merged. If none of them is redundant, then we merge them. If none of them is the last pair of its list, we look at the pairs $p_{i+1} = (h_{i+1}, w_{i+1})$ and $p'_{j+1} = (h'_{j+1}, w'_{j+1})$. If $w_{i+1} > w'_{j+1}$ then p_i and p'_{j+1} are to be merged next; if $w_{i+1} = w'_{j+1}$ then p_{i+1} and p'_{j+1} are to be merged next; if $w_{i+1} < w'_{j+1}$ then p_{i+1} and p_j are to be merged next. The procedure terminates when both p_i and p'_j are the last pairs of their lists.

Procedure Mg_H

```
(1)begin
(2)i:=1; j:=1;
(3)while (h_i, w_i) is not the last pair of L_1 or (h'_j, w'_j) is not the last pair of L_2 do
(4) begin
(5) i = 1 and w'_j is the last pair of L_2 then M_H((h_i, w_i), (h'_j, w'_j))
    /*M_H((h_i, w_i), (h'_j, w'_j)) = (h_i + h'_j, max{w_i, w'_j})*/
(6) else if j = 1 and (h_i, w_i) is the last pair of L_1 then M_H((h_i, w_i), (h'_j, w'_j))
(7)   else if i = 1 and w'_j < w'_{j+1} ≤ w_i then j:=j + 1
      /*(h'_j, w'_j) is not the last pair of L_2, but it is redundant*/
(6)     else if j = 1 and w_i < w_{i+1} ≤ w'_j then i:=i + 1
        /*(h_i, w_i) is not the last pair of L_1 but is redundant*/
(7)       else if (h_i, w_i) is the last pair of L_1
(8)       then begin
(9)         M_H((h_i, w_i), (h'_j, w'_j));
(10)        j:=j + 1
(11)        end
(12)    else if (h'_j, w'_j) is the last pair of L_2
(13)      then begin
(14)        M_H((h_i, w_i), (h'_j, w'_j));
(15)        i:=i + 1
(16)        end
(17)      else begin
        /*(h_i, w_i) and (h'_j, w'_j) are not the last pairs of L_1 and L_2*/
(18)        M_H((h_i, w_i), (h'_j, w'_j));
(19)        if w_{i+1} > w'_{j+1} then j:=j + 1
(20)        else if w_{i+1} = w'_{j+1} then begin
(21)          i:=i + 1;
(22)          j:=j + 1;
(23)          end
(24)        else i:=i + 1
(25)      end
(26) end;/*while loop*/
(27) M_H((h_i, w_i), (h'_j, w'_j));
(28)end.
```

This procedure needs constant time to generate a new pair of the list of u, L, We will show that a pair p is useful in L if and only if p is generated by Procedure Mg_H.

Let u be a node of the slicing tree T with children u_1 and u_2. Assume that L_1 and L_2 are the lists of u_1 and u_2, and their major-value is height. Let u be labeled "H," and L be the list of u. We have the following lemma:

Lemma 3 Let $p = (h, w)$ and $p = \text{M\_H}(p_i, p_j)$ where $p_i = (h_i, w_i)$ and $p_j = (h'_j, w'_j)$ are pairs in L_1 and L_2, respectively. If p is a useful pair for u then p is generated by Procedure Mg_H.

Proof: We distinguish the following cases:

Case 1. p_i is the last pair of L_1 and p_j is the last pair of L_2. By Procedure Mg_H, p is generated.

Case 2. p_j is the last pair of L_2 and p_i is not the last pair of L_1. Obviously, p_j is not the first pair of L_2 because L_2 contains at least two pairs. Also, (h'_{j+1}, w'_{j+1}) does not exist. Hence none of the Steps (5) and (6) are executed for p_i and p_j. Now suppose p_i does not merge with p_j. According to Procedure Mg_H, this can happen only if p_i merges with (h'_{j-1}, w'_{j-1}) and $w_{i+1} \leq w'_j$. If $w_{i+1} = w'_j$, then (h_{i+1}, w_{i+1}) will merge with p_j and generate the pair $(h_{i+1} + h'_j, \max\{w_{i+1}, w'_j\})$. Since the major-value of L_1 is height, by Lemma 1, $h_{i+1} < h_i$ and $w_{i+1} > w_i$. Hence the generated pair makes p useless, a contradiction. If $w_{i+1} < w'_j$, again, by Procedure Mg_H, there exists a pair $p_t = (h_t, w_t)$ in L_1 such

that $h_t < h_i$ and p_t merges with p_j. Similar to the discussion above, p becomes useless, a contradiction.

Case 3. p_i is the last pair of L_1 and p_j is not the last pair of L_2. Similar to Case 2.

Case 4. p_i and p_j are not the last pairs of L_1 and L_2, respectively. Apparently, both p_i and p_j can not be redundant, otherwise p is useless. Suppose p_i does not merge with p_j. Similar to the previous case, if this is true, then p is useless, a contradiction.

Considering all the possible cases for p_i and p_j, we conclude that if p is a useful pair for u, then p is generated by Procedure Mg_H. □

Next, we will show that Procedure Mg_H does not generate any useless pairs. Lemma 4 shows that for any two consecutive pairs (h, w) and (h', w') generated by running Procedure Mg_H on L_1 and L_2, we have $h > h'$ and $w < w'$. We need this property in order to show that Procedure Mg_H does not generate useless pairs.

Lemma 4 Let $p = (h, w)$ and $p' = (h', w')$ be two consecutive pairs generated by running Procedure Mg_H on L_1 and L_2, where the major-value of L_1 and L_2 is height. If p is generated before p' then $h > h'$ and $w < w'$.

Proof: Let $p = \text{M\_H}(p_i, p_j)$ where $p_i = (h_i, w_i)$ and $p_j = (h'_j, w'_j)$ are pairs in L_1 and L_2 respectively. We distinguish the following cases:

Case 1. p_i is the last pair of L_1. Obviously, p_i is not the last pair of L_2, otherwise there is no p' generated. Let $p'_{j+1} = (h'_{j+1}, w'_{j+1})$, if $w_i \geq w'_{j+1}$, according to Procedure Mg_H, p_i does not merge with p_j, a contradiction. So $w_i < w'_{j+1}$. By Procedure Mg_H, $p' = \text{M\_H}(p_i, p'_{j+1})$. Hence, $p = (h_i + h_j, \max\{w_i, w'_j\})$ and $p' = (h_i + h'_{j+1}, w'_{j+1})$. Since the major-value of L_1 is height, we have $h'_j > h'_{j+1}$, which implies $h > h'$. Because the minor-value of L_1 is width (i.e., $w'_j < w'_{j+1}$) we have $w > w'$.

Case 2. p_j is the last pair of L_2. Similar to Case 1.

Case 3. Neither p_i nor p_j is the last pair. In this case, p' is decided by comparing w_{i+1} and w'_{j+1}. $p' = (h_i + h'_{j+1}, \max\{w_i, w'_{j+1}\})$ if $w_{i+1} > w'_{j+1}$. As in Case 1, $w_i < w'_{j+1}$, otherwise p is not generated. Comparing p and p', $h > h'$ and $w < w'$. Similarly, we can prove that for $w_{i+1} = w'_{j+1}$ and $w_{i+1} < w'_{j+1}$, also we have $h > h'$ and $w < w'$. □

A simple induction on Lemma 4 proves the following:

Lemma 5 Procedure Mg_H does not generate useless pairs. □

Let L^* be the list that contains all the useful implementations and no useless implementations for u. From Lemma 3 and Lemma 5 we have the following:

Theorem 1 For any pair p, p is in L^* if and only if p is generated by Procedure Mg_H. Furthermore, Procedure Mg_H runs in $O(|L_1| + |L_2|)$ time and generates at most $|L_2| + |L_2| - 1$ pairs.

Proof: Suppose p is in L^*. From Lemma 3, p is generated by Procedure Mg_H.

Suppose that p is not in L^*. Then there must be a useful pair q in L^* that makes p useless. But q is also generated by Procedure Mg_H. Using Lemma 5, the fact that q is generated by Mg_H implies that p is not generated by Procedure Mg_H, a contradiction.

Since Procedure Mg_H moves one pair down each time at least in one of L_1 and L_2, it takes time $O(|L_1| + |L_2|)$. Moreover, similar to Lemma 2, Procedure Mg_H generates at most $|L_1| + |L_2| - 1$ pairs. □

After selecting the best implementation of the floor rectangle, we need to trace down to the basic rectangles in order to obtain the optimal implementations of the cells. This is done easily by keeping two pointers for each pair in each list that point to the pairs of the children that generated the pair. The Algorithm FP below uses Procedures Merge_V and Mg_H to find the optimal implementations of the cells.

Algorithm FP

(1) **begin**
(2) Prepare the lists of all the leaves so that the major-value is height.
(3) From the second to the bottom level to the root level **do**
(4) From the leftmost node to the rightmost node **do**
(5) **if** this node is labeled "V" **then** call Procedure Merge_V
(6) **else** call Procedure Mg_H;
(7) Let L_r be the list of the root. Scan all the pairs in L_r and select the one with minimum area, i.e., height × width.
(8) Using the pointers of the selected pair, trace down to the cells and return the optimal implementations of the cells.
(9) **end**.

If T is a skewed slicing tree with internal nodes labeled "V" and "H" alternately, the height of T is

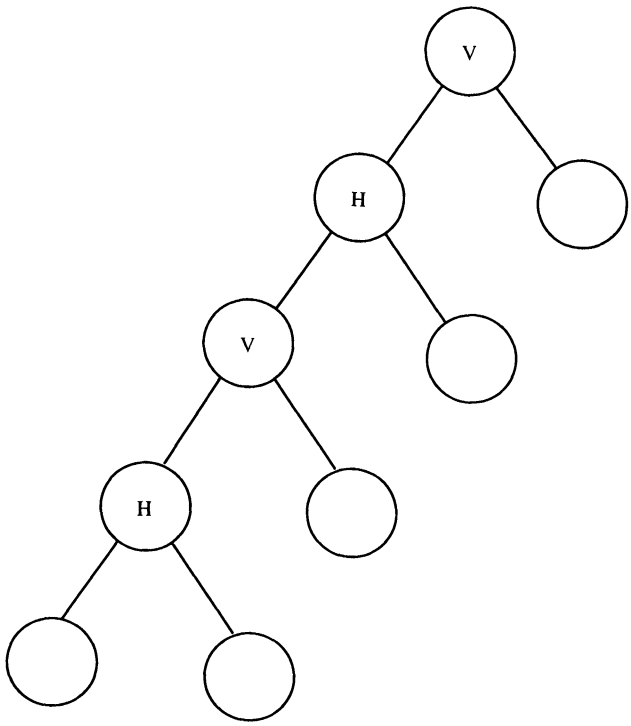

FIGURE 4 A skewed slicing tree.

$O(n)$, see Figure 4. Because there are $O(n)$ cells, the time needed for generating L is $O(n^2)$. Hence we have the following Theorem:

Theorem 2 Let T be a slicing tree with n leaves. Algorithm FP computes the optimal implementations of the cells in time $O(n^2)$. □

4 THE PARALLEL ALGORITHM

Let r be the root of a slicing tree T, and assume that the lists of all the leaves are prepared such that the major-value is height. In this section we present an algorithm that uses $O(n)$ processors and computes the list of r in $O(n)$ time. To simplify our description, we assign a processor to each internal node. If the node is labeled "V," and for each of its children's lists there is at least one pair available, then we apply Procedure Merge_V to merge them; if it is labeled "H" and for each of its children's lists there are at least two pairs available, then we apply Procedure Mg_H to merge them. As soon as a pair is generated by a processor, it is sent to the processor of its parent. A processor stops computing when Procedure Merge_V or Mg_H is finished.

There is a way to embed an arbitrary binary tree into a hypercube with dilation three (i.e., such that any two neighboring processors in the tree are at distance at most three in the hypercube). Furthermore, an arbitrary binary tree can be embedded into its *optimal* hypercube with dilation five [3]. Hence, we can embed a slicing tree into a hypercube so that the communication delay of our parallel algorithms are only multiplied by three or five.

Notice that we do not need to assign one processor per internal node. The exact number of processors depends on the structure of the slicing tree. One can use a pool of available processors. When a processor is needed then it is taken from the pool. Similarly, when a processor finishes its computation, it is returned to the pool. However, in this model, we assume that the processors are fully interconnected.

Algorithm PFP;
(1) **begin**
(2) **for** all the internal nodes the corresponding processors **do** the following in parallel:
/*each processor contains two lists corresponding to the children's lists*/
while there are pairs available in their children's lists **do**
begin
if the node is labeled "V" and there is a pair available in each list of its children **then**
begin
merge the children's lists (according to Procedure Merge_V)
for each generated pair, it is passed to the processor of the parent node;
end
else if the node is labeled "H" and
there are at least two pairs available in both children's lists **then**
begin
merge the children's lists (according to Procedure Mg_H);
for each generated pair, it is passed to the processor of the parent node;
end;
end

(3) Let L_r be the list of the root. Scan all the pairs in L_r and select the one with minimum area.
(4) Using the pointers of the selected pair, trace down to the cells and return the optimal implementations of the cells.
(5) **end.**

Before we continue, we need the following simple observations:

(1) Each pair is generated in constant time. The time taken to generate a pair is called a *basic step*.
(2) In Procedure Mg_H the redundant pairs will not generate pairs in their parent's list, because if a pair is known to be redundant, it will not be merged with pairs of the other list.
(3) If (h_i, w_i) is not redundant then (h_{i+1}, w_{i+1}) is not redundant because $w_{i+2} > w_{i+1} > w'_1$.

Let u be a node of slicing tree T with children u_1 and u_2, and L, L_1 and L_2 be the lists of u, u_1 and u_2 respectively. Also let u be labeled "H" and the major-value of L_1 and L_2 be height. Suppose that the first k pairs of L_2 are redundant. Using Procedure Mg_H to merge L_1 and L_2, the processor in u generates the first pair of L $k + 2$ basic steps after it started computing. This is because Procedure Mg_H will skip the first k pairs of L_2, and start merging the first pair of L_1 and the $(k + 1)$th pair of L_2 at basic step $k + 1$. Notice that we assumed that a processor needs one *basic step* to check and skip a redundant pair, although in fact it takes less time than generating a pair.

Lemma 6 Let u be a node of T, and T' be the subtree rooted at u. Assume that the height of T' is l. If there are k redundant pairs generated by the nodes of T', then the processor at u starts generating pairs no later than basic step $l + k$.

Proof: If u is labeled "V" then there are no redundant pairs to be considered. So we only need to consider the case that u is labeled "H." Let T_1, T_2 be the subtrees of T rooted at u_1 and u_2. We use induction on l.

Assume that $l = 1$. This implies that u_1 and u_2 are both leaves. Assume that there are c_1 and c_2 pairs in L_1 and L_2, respectively, where c_1 and c_2 are constants. Suppose there are k redundant pairs for T', then those k redundant pairs must be either in L_1 or L_2. W.l.o.g., let the redundant pairs be in L_1. Hence the processor at u is checking and skipping the redundant pairs until the $(k + 1)$th pair of L_1 is considered, and starts merging the pairs at basic step $l + k$.

Now assume that for $t < l$, where t is the height of T', the processor at u starts generating pairs at basic step $t + k$ where k is the number of redundant pairs generated by the nodes of T'.

Let the height of T' be l. Assume that the number of redundant pairs generated by the nodes of T_1 and T_2 is k_1 and k_2, respectively. Also assume that there are m redundant pairs either in L_1 or in L_2. Since the height of both T_1 and T_2 is less than l, u_1 starts generating pairs at basic step $l - 1 + k_1$ and u_2 starts generating pairs at basic step $l - 1 + k_2$. Hence, u will start generating pairs no later than basic step $l + \max\{k_1, k_2\} + m$. But the nodes of T' generate $k = k_1 + k_2 + m$ redundant pairs and $\max\{k_1, k_2\} \leq k_1 + k_2$. Therefore, the processor at u starts generating pairs no later than basic step $l + k$. □

Lemma 7 Let each leaf of T have at most c implementations. The processor at the root of T will start generating pairs no later than $(c + 1)n$ basic steps.

Proof: Let l be the height of T. Since each redundant pair does not generate any pair for upper nodes, if at some level i there are p pairs totally and q of them are redundant, then there are at most $p - q$ pairs for every level higher than i. By Lemma 2 and Theorem 1, we know that there are totally no more than cn pairs generated by the nodes in the same level of T. Hence, there are at most cn pairs generated by the root of T. This implies that there are at most cn redundant pairs generated by the nodes of T. By Lemma 6, the processor at the root starts generating pairs no later than basic step $l + cn \leq (c + 1)n$. □

In the next lemma we show that for any processor, once it starts generating pairs, it will always have enough pairs in its children's lists so that it generates a new pair for each basic step until the pairs of one of its children's list are exhausted.

Lemma 8 Once a processor at some node u of T starts generating pairs it will generate a new pair for each basic step until one of the lists of its children is exhausted.

Proof: Let the height of u be l. Let u_1 and u_2 be the children of u, and L_1, L_2 be the lists of u_1 and u_2, respectively. We use induction on l.

Assume that $l = 1$. Clearly, u_1 and u_2 are both leaves. The processor at u does not need to wait for the new pairs arriving at u_1 and u_2. Hence it will not be interrupted.

Next, assume that for $t < l$, where t is the height of u, the processor at u will generate a new pair for each basic step until one of its children's list is exhausted.

Now consider the processor at u that has height l. Let the processor at u start generating pairs at basic step j. This implies that there are enough non-redundant pairs of L_1 and L_2 available at basic step $j - 1$ (by Algorithm PFP). So after basic step j, by induction hypothesis, new pairs will be generated at u_1 and u_2. If u is labeled "V," there are no redundant pairs need to be considered at u. Again, by induction hypothesis, at each basic step, there are new pairs generated at u_1 and u_2, respectively, until one of L_1 or L_2 is exhausted. If u is labeled "V," then there are no redundant pairs need to be considered. If u is labeled "H," by Observation 3, the new generated pairs at u_1 and u_2 are not redundant. Hence, there are always enough pairs available for merging at u_1 and u_2, thus the processor at u will not be interrupted until no more pairs arrive at u_1 or u_2. □

Combining the above results we obtain the following:

Theorem 3 Let T be a slicing tree with n leaves. Algorithm PFP computes the optimal implementations of the cells in $O(n)$ time using $O(n)$ processors.

Proof: Since T has n leaves, there are $2n - 1$ nodes in T. Hence $O(n)$ processors are enough for the parallelization.

By Lemma 7, the processor at the root will start generating pairs no later than $(c + 1)n$ basic steps; by Lemma 8, once it starts generating pairs, it will not stop until all the pairs are generated. Furthermore, since there are at most cn pairs in L_r, the list of the root, so the time for generating L_r can be calculated as follows:

Total generating time = Broadcasting time + Waiting time + Execution time of the processor at the root $\leq (c + 1)n + 0 + cn = O(n)$. For steps (3) and (4), only $O(n)$ time is needed to select the pair and trace down to each cell. Hence the whole process can be done in $O(n)$ time. □

5 THE PARALLEL ALGORITHM FOR ALMOST BALANCED SLICING TREES

As we discussed in the Introduction, the target of floorplanning is to determine a set of implementations (one for each basic rectangle) such that the relative positions of the basic rectangles are preserved, and the total layout area of the chip is minimized. The *topology,* i.e., the relative positions of basic rectangles, of a slicing floorplan is obtained by recursively using circuit bipartitioning techniques. A *bipartition* divides a given circuit into two parts such that: (1) the sizes (the number of basic rectangles) of the two parts are as equal as possible, and (2) the number of nets connecting the two parts is minimized. Hence, clearly, the slicing trees generated by bipartitioning techniques have height $O(\log n)$.

The result of Theorem 3 achieves optimal speedup in the worst case, i.e., when the slicing tree has height $O(n)$. However, if the height of the tree is $O(\log n)$, the sequential algorithm takes only $O(n \log n)$ time. Here we will show how to compute the optimal implementations in $O(n)$ time using $O(\log n)$ processors.

Algorithm FBT;
(1) **begin**
(2) Find some level i of T that contains $O(\log n)$ nodes.
(3) The nodes at level i are the roots of $O(\log n)$ subtrees. Dedicate one processor to each subtree.
(4) For each such subtree, use Algorithm FP (except for the last two steps) to compute the lists of the roots sequentially. Of course all $O(\log n)$ lists are computed in parallel independently.
(5) Consider the subtree of T consisting of the root and all the nodes down to level i, call it T'. T' has $O(\log n)$ leaves (the roots of the previous subtrees), whose lists were computed in Step 3. Call Algorithm PFP on T', using a processor for each non-leaf node of T'.
(6) **end**.

Figure 5 shows an example of a balanced slicing tree with height 4. During the first step of Algorithm FBT, T is partitioned into T' and $O(\log n)$ subtrees. Each such subtree has $O(n/\log n)$ leaves and height $O(\log n)$. If we use Algorithm FP the list of the root of each subtree can be clearly obtained in $O(n)$ time. Also, T' is a balanced tree with $O(\log n)$ leaves whose lists contain $O(n/\log n)$ pairs. Since there are at most cn pairs in each level of T', Algorithm PFP will compute the best implementations of the cells in $O(n)$ time. Hence we have the following:

Theorem 4 Let T be a slicing tree with n leaves and height $O(\log n)$. Algorithm FBT computes the optimal implementations of the cells in $O(n)$ time using $O(\log n)$ processors. □

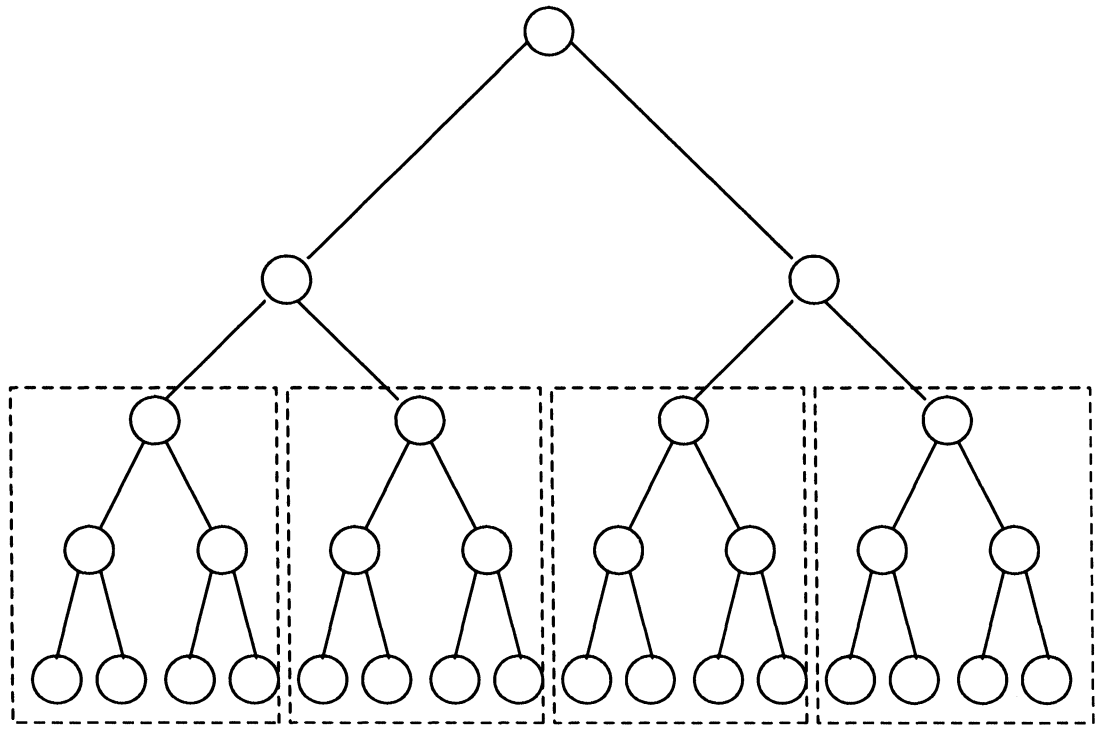

FIGURE 5 A balanced slicing tree with height = 4.

6 EXPERIMENTAL RESULTS

We simulated Algorithm PFP to generate all the useful implementations of F in Pascal on a Sun 4 workstation under the UNIX operating system. We compared the number of steps needed for the sequential algorithm with the number of steps needed for the parallel algorithm. In EXP1, we tested 10 skewed slicing trees. In EXP2, we tested 10 non-skewed slicing trees. The trees are randomly generated. The results are shown in Table I and Table II, respectively. For the parallel algorithm, we also computed the number of processors used. The number of processors to be used is obtained as the following: We use a pool of available processors. When a processor is needed then it is taken from the pool. When a processor finishes its computation, it is returned to the pool. We can see that the number of processors needed in both EXP1 and EXP2 is close to 70% of the number of leaves of the slicing trees. The results are also plotted in Figure 6 and Figure 7. These two figures verify that the sequential algorithm needs $O(n^2)$ time and the parallel algorithm needs $O(n)$ time to find the optimal implementations of the basic

TABLE I
The Experimental Results of EXP1

No. of Leaves	Seq. Algorithm	Parallel Algorithm	
	No. of Steps	No. of Steps	No. of Processors
3	7	7	2
4	12	10	3
8	42	20	5
12	87	29	8
14	116	34	10
16	147	38	11
18	184	44	12
20	225	49	14
22	270	54	15
26	369	63	18

TABLE II
The Experimental Results of EXP2

No. of Leaves	Seq. Algorithm	Parallel Algorithm	
	No. of Steps	No. of Steps	No. of Processors
3	7	7	2
4	12	10	3
6	24	13	4
8	32	14	6
10	45	17	7
12	58	21	9
16	90	27	10
20	118	31	12
24	158	36	14
26	178	38	16

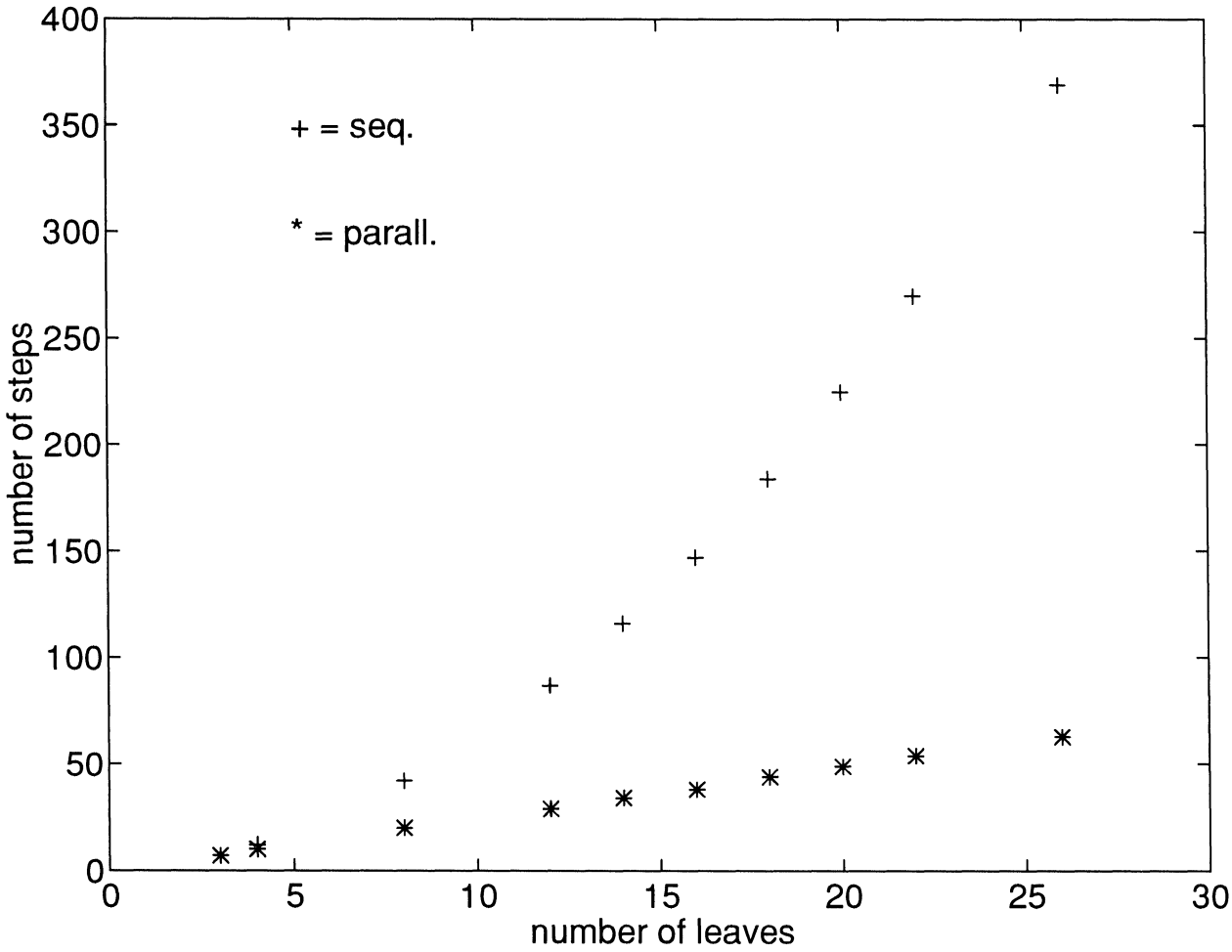

FIGURE 6 The experimental results of EXP1.

rectangles for F. Also we note that the time needed for skewed slicing trees is more than the time needed for non-skewed slicing tree.

7 CONCLUDING REMARKS

In this paper, we presented a parallel algorithm to compute the optimal implementations of the basic rectangles for any slicing floorplan. Our algorithm runs in $O(n)$ time with $O(n)$ processors, in the worst case, where n is the number of basic rectangles. We also presented a more efficient algorithm that solves the area optimization problem for floorplans whose corresponding slicing trees have height of $O(\log n)$. Namely, our algorithm runs in $O(n)$ time and uses $O(\log n)$ processors. Both parallel algorithms achieve optimal speedup. In addition, our algorithms do not need shared memory and can be implemented in a distributed system.

We provided experimental results that verify the theoretical speedup of Algorithm PFP, and showed that we use about $0.7n$ processors.

When the height, l, of the slicing tree is more than $O(\log n)$ and less than $O(n)$, the sequential algorithm of Section 3 takes $O(ln)$ time to find the optimal implementations. There is an interesting question: Is it possible to achieve optimal speed up in this case? Our preliminary results indicate that it is possible to solve the problem in $O(n)$ time using $O(l)$ processors. However the technique seems to be rather complicated and not very realistic, because we need a powerful parallel machine with shared memory that allows concurrent reads and concurrent writes. Another interesting question is the following: Can this problem be solved in poly-log time using a polynomial number of processors or is the problem P-complete?

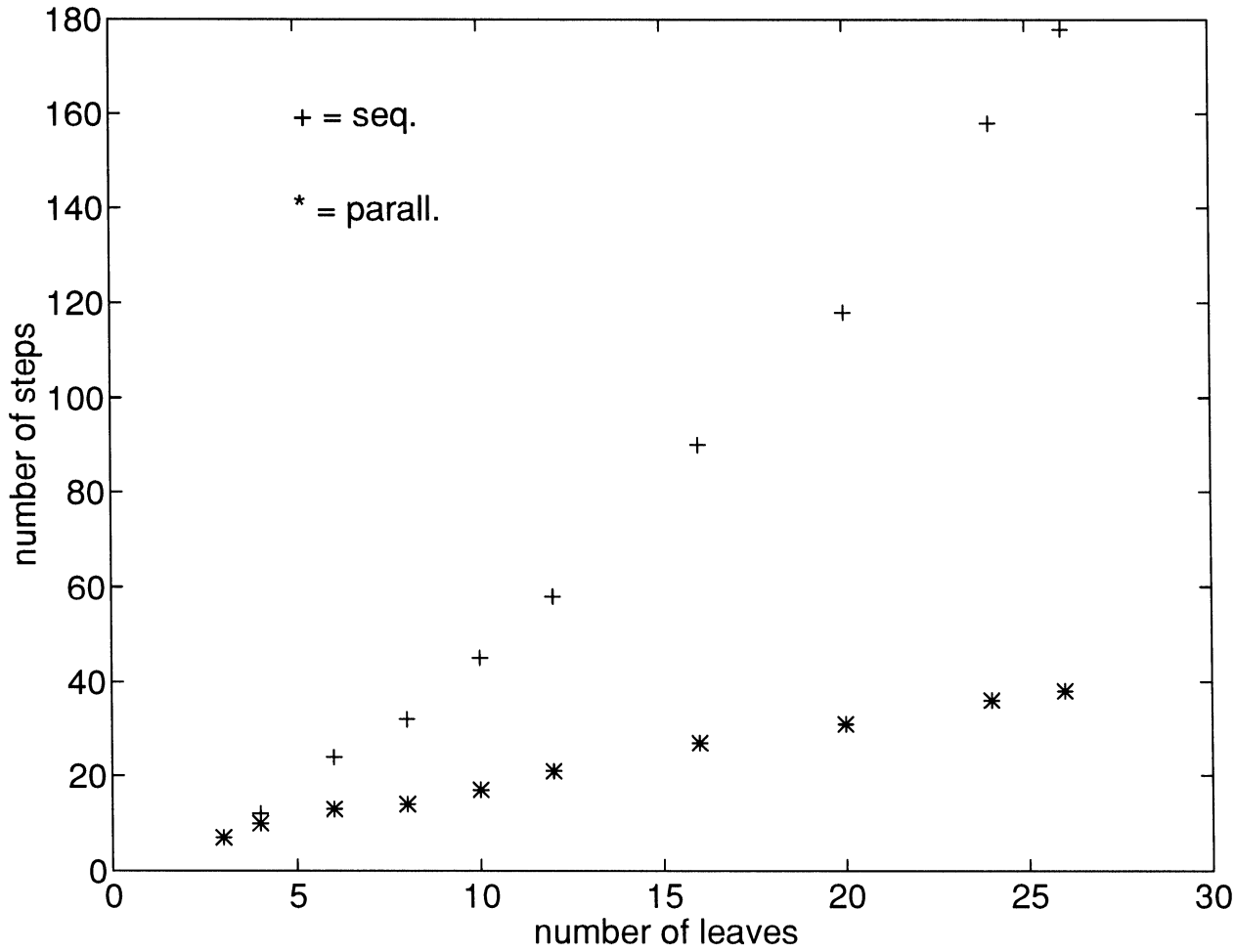

FIGURE 7 The experimental results of EXP2.

References

[1] E.S. Kuh and T. Ohtsuki, "Recent Advances in VLSI Layout," *Proceedings of the IEEE,* Vol. 78, No. 2, pp. 237–263, 1990.

[2] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout,* John Wiley & Sons, 1990.

[3] B. Monien and I.H. Sudborough, "Simulating Binary Trees on Hypercubes," VLSI Algorithms and Architectures, Lecture Notes in Computer Science 319, AWOC 88, pp. 170–180.

[4] R.H.J.M. Otten, "Automatic Floorplan Design," in *Proceedings, 19th Design Automation Conf.,* pp. 261–267, 1982.

[5] R.H.J.M. Otten, "Efficient Floorplan Optimization," *ICCD83-IEEE Int. Conf. on Computer Design,* pp. 499–502, 1983.

[6] L. Stockmeyer, "Optimal Orientations of Cells in Slicing Floorplan Designs," *Information and Control* 57, pp. 91–101, 1983.

[7] S. Wimer, I. Koren, and I. Cederbaum, "Optimal Aspect Ratios of Building Blocks in VLSI," in *25th ACM/IEEE Design Automation Conference,* pp. 66–72, 1988.

[8] S. Wimer, I. Koren, and I. Cederbaum, "Floorplans, Planar Graphs and Layouts," in *IEEE Trans. on Circuits and Systems,* pp. 267–278, 1988.

[9] D.F. Wong and C.L. Liu, "A New Algorithm for Floorplan Design," *Proc. 23rd ACM/IEEE Design Automation Conference,* pp. 101–107, 1986.

[10] D.F. Wong and C.L. Liu, "Floorplan Design for Rectangular and L-shaped Modules," *Proc. IEEE Intl. Conference on Computer-Aided Design,* pp. 520–523, 1987.

Biographies

DR. CHENG-HSI JESSE CHEN received his M.S. and Ph.D. degree in Computer Science from the University of Texas at Dallas in December 1985 and August 1993, respectively. He received his Master degree in Business Administration from Tarleton State University, Stephenville, Texas, in December 1983, and B.S. degree in Chemistry from National Tsing Hua University, Hsing Chu, Taiwan, in June, 1978. From 1986 to 1987, he worked in business company as a programmer and manager. Since September, 1992, he became a Postdoctoral Fellow in the Dept. of Electrical and Computer Engineering, Concordia Univeristy.

His research interests include VLSI/CAD, Parallel Computation, Graph Algorithms, and Computer Architectures. He is currently working on Parallel Processing for VLSI/CAD.

DR. IOANNIS G. TOLLIS received his Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in November 1987. He received his Diploma degree in Mathematics from the National University of Athens, Greece and his M.S. degree in Computer Science from Vanderbilt University, Nashville, Tennessee. He joined the faculty of The University of Texas at Dallas in December 1987, where he is currently Associate Professor of Computer Science. During the summer of 1985 he held a visiting engineer position at the Computer Research Laboratory, Tektronix Laboratories, Beaverton, Oregon.

His research interests include Computer Aided Design for VLSI, Layout Algorithms, Graph Algorithms, Computational Geometry, and Parallel and Distributed Processing. He served in the Program Committee of the Second IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, Dec. 1990, and is a permanent member of the steering committee of the same conference. Also, he served as a member of the Program Committee of the 1993 IEEE International Symposium on Circuits and Systems. Dr. Tollis is a member of ACM, IEEE and EATCS.